BLT: Exact Bayesian Inference with Distribution Transformers

Charles Yuan and Jan Hoffmann

Carnegie Mellon University, Pittsburgh, PA 15213, USA
{charlesyuan,jhoffmann}@cmu.edu

Abstract. This paper presents a static analysis for solving the Bayesian inference problem for finite-state probabilistic programs featuring categorical random variables and looping control flow. The results of the analysis are transformations on state distributions, which efficiently compute output distributions from input distributions. The inference algorithm generates and composes probability transformations, and translates them to a system of constraints solvable by a combination of linear algebra and linear programming. To improve the efficiency of inference queries on output distributions, a dataflow analysis computes, for each program fragment, the relevant variables for an inference query. The inference is exact and proved to be sound with respect to a denotational semantics. The analysis has been implemented in the tool BLT, which successfully infers output distributions for probabilistic programs with possibly non-terminating loops. An experimental evaluation with existing and new benchmarks like Bayesian networks shows that BLT's performance is comparable to state-of-the-art solvers for exact inference.

Keywords: Probabilistic programming · Bayesian inference · Program analysis · Software verification

1 Introduction

Probabilistic programming languages (PPLs) precisely describe probabilistic processes using general-purpose programming languages that are extended with probabilistic features that manipulate random variables and their distributions. For example, variables may be initialized by sampling from some distribution, affecting control flow and introducing nondeterministic execution. The semantics of a probabilistic program describe a distribution of possible outcomes. Numerous popular languages [26][12][17][29] as well as domain-specific languages [10][16] now possess probabilistic programming capabilities.

PPLs are useful for a wide variety of applications in computer science, statistics, machine learning, computer vision, and other fields [11][14][23]. They are powerful enough to express techniques such as probabilistic graphical models, which are used to model complex systems as a graph consisting of random factors

An artifact has been made available as part of the submission of this paper.

and interactions [18]. One prominent graphical model that PPLs may describe is Bayesian networks, graphs of random variables where directed edges indicate conditioning between two variables [22]. Bayesian networks have been used to analyze diverse phenomena such as disease markers and patterns within meteorological data [32].

1.1 Probabilistic Inference

The *inference problem* for probabilistic programs is the determination, given a set of inputs, of the output probability distribution of a program over all of its possible variable states. That is, given some of the variables present in the program, and some desired assignments to those variables, inference computes the probability that the desired state is realized at program termination. Inference subsumes many useful analyses of a program, such as the expected values of each variable, the expected time of termination, and the conditional or marginal distributions for subsets of variables.

Traditional approaches to inference largely consist of sampling and statistical techniques, such as the Metropolis-Hastings and Gibbs sampling algorithms [5][28][2]. These methods are *dynamic* in nature, meaning that they require simulated execution of the program, often over a large number of samples.

Such dynamic techniques may take an indeterminate amount of time and resources to complete, and generally there is little assurance that the results are sound: that output figures are sufficiently accurate, or that if an analysis fails to converge, the input program can also be deemed to not terminate. Sampling inherently relies on a source of (pseudo-)randomness, which may introduce systematic error and uncertainty into the process.

By contrast, *static* methods for inference apply techniques from program analysis and compiler construction to avoid sampling. Techniques such as symbolic execution [5] and path exploration [30] have been used to represent program states symbolically. Static methods have the potential to be *exact*, not incurring the errors and approximations normally required by sampling. They can also be proven sound on entire classes of programs, giving confidence for their outputs.

However, exactness can come at a cost in terms of increased computational complexity. In particular, exact inference on PPLs with unbounded data types such as integers is undecidable [21]. Even after restricting to only finite states, exact inference is **#P**-complete [7]. Any attempt at exact inference must therefore acknowledge a possible exponential-time worst-case performance, and strive to make common cases more computationally feasible.

Another challenge in exact inference is accounting for loops and nontermination. Most exact systems account for at most bounded loops in programs, and cannot interpret programs that are not almost certainly terminating. Of those that handle unbounded loops, one common limitation is iterative, numeric convergence of accumulated facts around loops [5], and others include restrictions on loop structure [1] and requirements for annotated invariants [28].

In this paper we develop BLT, the Bayesian Loop Transformer system, a novel approach to performing static, exact Bayesian inference on imperative

finite-state probabilistic languages featuring categorical random variables and unbounded looping constructs. In particular, our developments are:

- an efficient algorithm for *inference* that reduces probabilistic programs to linear algebra and linear programming operations;
- a *compositional* analysis and proof of *soundness* aligning naturally with the semantics of the language;
- successful accounting for *nontermination*, including possibly unbounded loops, without sacrificing guaranteed convergence;
- identification of relevant program variables through dataflow analysis, facilitating *optimization* of analysis;

2 Imperative Probabilistic Programs

The probabilistic language we study is a variant of the guarded command language and possesses syntactic expressions and statements. For our initial consideration, only Boolean random variables and expressions of Boolean algebra are present.

$$
\begin{aligned}
\mathsf{Typ}\ \tau &::= \texttt{bool} \\
\mathsf{Exp}\ e &::= x \mid e_1 \wedge e_2 \mid e_1 \vee e_2 \mid \neg e_1 \mid e_1 = e_2 \mid e_1 \neq e_2 \\
\mathsf{Stm}\ s &::= \texttt{skip} \mid x \leftarrow e \mid x \sim \mathcal{B}(p) \\
&\quad \mid \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2 \mid \texttt{while}\ (e)\ s_1 \mid \texttt{observe}\ (e) \mid s_1;\ s_2
\end{aligned}
$$

Fig. 1: Syntax for imperative probabilistic language.

Two constructs are unique to probabilistic programs: the *Bernoulli sampling* operation, which nondeterministically grants a variable a truth value with some fixed probability $0 \leq p \leq 1$, and the *observation* operation, which acts as a probabilistic assertion statement. That is, the statement `observe` (e) checks if the expression e evaluates to true, and if so continues the program with no effect. Otherwise, we take the standard interpretation that program execution ceases; the run is considered rejected, which we characterize the same way as it having failed to terminate, or having *diverged*.

Based on this characterization of observation, we simply define it in terms of the loop construct:

$$\texttt{observe}\ (e) \triangleq \texttt{while}\ (\neg e)\ \texttt{skip}$$

The issue of nontermination is central to inference. Should a run of a program fail to terminate, we still wish to recover useful information about it. Our analysis must treat program nontermination similarly to any valid termination state, so that we may ask for e.g. the probability that a program terminates, or the final state given that a program has terminated.

2.1 Example Programs

The following programs demonstrate the capabilities of our language. In Example 2, we specify that there is a 10% chance of rain and that in the event of rain, there is a 75% chance of having brought an umbrella. Variables are initialized to false, so the probability of bringing an umbrella is $0.1 \times 0.75 = 0.075$. Likewise, the probability of rain without an umbrella is $0.1 \times (1 - 0.75) = 0.025$.

```
bool raining = Bernoulli(0.1);
bool brought_umbrella;
if (raining) {
    brought_umbrella = Bernoulli(0.75);
}
```

Ex. 2: Conditioning the probability of an event on another.

```
bool coin;
while (!coin) {
    coin = Bernoulli(0.1);
}
```

Ex. 3: Flipping a biased coin.

```
bool b1 = Bernoulli(0.25);
bool b2 = Bernoulli(0.5);
observe(b1 || b2);
```

Ex. 4: Observing a condition.

In Example 3, the loop that flips a coin until its value is true is guaranteed to terminate with probability 1, though after an unbounded number of iterations. At termination, the only possible state is `coin` being certainly true.

The program in Example 4 terminates only if one of `b1` or `b2` is sampled to true. The resulting distribution from this program is that both are true with probability $0.25 \times 0.5 = 0.125$, `b1` alone with probability 0.125, `b2` alone with probability 0.375, and the program is divergent with probability 0.375.

2.2 Categorical Random Variables

Categorical, or discrete, random variables may be added to the language as a type of bounded integers with equality comparison. As an example, the following program randomly selects one of three options, then observes either the first or third choice:

```
cat[3] choice = Categorical(0.1, 0.8, 0.1);
observe(choice == 1 || choice == 3);
```

In our approach, categorical variables are encoded as sequences of Boolean variables. An n-way categorical variable may be represented using $\lceil \log_2 n \rceil$ bits, each of which becomes a Boolean variable in the program. Categorical assignment and comparison may be reduced to assignment and comparison of the associated bitvectors. Sampling may be reduced to sequential binomial sampling, iterating through each of the specified probabilities and at each branch storing a bit sequence into the associated variables.

2.3 Semantics

We now describe a denotational semantics of the language that maps each statement to a mathematical transformation on program state distributions. Let Var be the set of variables in the program.

Definition 1 (State). *A program state $\sigma \subseteq \mathsf{Var}$ contains variables set to true. The set Σ of all states is $\mathcal{P}(\mathsf{Var})$.*

Definition 2 (Distribution). *A state distribution $\Sigma \to [0,1]$ provides for each state the probability that it contains exactly the true variables.*

The sum of probabilities across all states is nonnegative and at most 1. When the sum is less than 1, we have a *state sub-distribution*, in which there is some residual probability of nontermination. Point-wise addition and scalar multiplication are defined on distributions in the usual manner.

Definition 3 (Transformer). *A state transformer $\Sigma \to \Sigma \to [0,1]$ represents transitions from states to other states with certain probabilities.*

Applying a transformer to a state yields a distribution, so a transformer may also be seen as a function from source state to destination distribution.

For a set X define the operator $(=) : X \to X \to [0,1]$ to be 1 if its arguments are equal and 0 otherwise. The following is an instantiation of a well known probability-theoretic construction [15].

Definition 4 (Giry monad). *The Giry monad $\mathcal{D}$ acts on probability distributions such that $\mathcal{D}(X)$ is defined to be $X \to [0,1]$. Define unit and bind operators:*

$$\begin{aligned} \delta &: X \to \mathcal{D}(X) \\ \delta(x) &\coloneqq \lambda x'.x = x' \\ (-)^\dagger &: (X \to \mathcal{D}(Y)) \to \mathcal{D}(X) \to \mathcal{D}(Y) \\ f^\dagger(d) &\coloneqq \lambda y. \sum_{x \in X} f(x)(y) \cdot d(x) \end{aligned}$$

The monad gives us a means of relating transformers and functions from distributions to distributions. We will use the notation $X \Rightarrow Y$ to denote $X \to \mathcal{D}(Y)$ or equivalently $X \to Y \to [0,1]$ throughout this paper. X is denoted the *domain* and Y the *range* of the transformer.

The idea of taking marginal distributions on random variables generalizes to taking *marginal transformers*. Marginalization produces a new transformer with a restricted domain and range, through summation over rows and columns that are no longer differentiated under the new restricted state sets. In other words, it is the meaning of the transformer ignoring effects on a subset of state variables.

The full denotational semantics follow in Figure 5. The semantics for expressions are omitted and give meaning to states as functions from expressions to truth values. $\sigma[x \leftarrow t]$ denotes state $\sigma \cup \{x\}$ if t is true, or $\sigma \setminus \{x\}$ if t is false. Each statement has a denotation as a transformer: $\Sigma \Rightarrow \Sigma$.

$$
\begin{aligned}
[\![-]\!] &: \mathsf{Stm} \to \Sigma \Rightarrow \Sigma \\
[\![\texttt{skip}]\!] &= \delta \\
[\![x \leftarrow e]\!](\sigma) &= \delta(\sigma[x \leftarrow \sigma(e)]) \\
[\![x \sim \mathcal{B}(p)]\!](\sigma) &= p \cdot \delta(\sigma[x \leftarrow \texttt{true}]) + (1-p) \cdot \delta(\sigma[x \leftarrow \texttt{false}]) \\
[\![s_1;\ s_2]\!](\sigma) &= [\![s_2]\!]^\dagger\,([\![s_1]\!](\sigma)) \\
[\![\texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2]\!](\sigma) &= \begin{cases} [\![s_1]\!](\sigma) & \text{if } \sigma(e) \\ [\![s_2]\!](\sigma) & \text{otherwise} \end{cases} \\
[\![\texttt{observe}\ (e)]\!] &= [\![\texttt{while}\ (\neg e)\ \texttt{skip}]\!] \\
[\![\texttt{while}\ (e)\ s]\!] &= \mu X.\lambda\sigma. \begin{cases} X^\dagger([\![s]\!](\sigma)) & \text{if } \sigma(e) \\ \delta(\sigma) & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 5: Denotational semantics for statements.

The `skip` operator simply sends the input to the output. Assignment updates the state with the value of the expression, and sampling returns a weighted sum of the two program distributions with the variable updated. Sequencing applies the first statement, then binds its output to the second. The conditional branch selectively executes one of the statements depending on the condition, and observation is simply translated to the equivalent loop.

Finally, the loop construct is recursively defined as a transformer which has no effect if the condition is false, or executes the body and repeats itself otherwise. As in [1], we express this recursion with the least fixed point operator μ and use the ordering between transformers on the weight of the output distributions. As each of the transformers is $\Sigma \Rightarrow \Sigma$ where Σ is discrete, this ordering is pointwise: we say that $T_1 \leq T_2$ when for all $\sigma, \tau \in \Sigma$, we have $T_1(\sigma)(\tau) \leq T_2(\sigma)(\tau)$. The definition ensures that $\leq$ is a directed complete partial order, and since f is monotone, Kleene's fixed point theorem guarantees existence of the fixed point.

3 Bayesian Inference

Given a program s, its denotation $[\![s]\!]$ is its corresponding transformer, meaning that given an initial state σ, $[\![s]\!](\sigma)$ is the distribution of output states. We now have a Bayesian interpretation of the inference problem, where given a prior distribution of inputs we may infer the posterior distribution of outputs. The composition of transformers represents the iterated application of Bayes' rule.

So, the core inference algorithm is simply an implementation of the denotational semantics, and performs a syntax-directed conversion of all program constructs into concrete data structures for distributions and transformers.

3.1 Compositional Matrix Transformers

With only Boolean variables present, the most direct approach to encoding distributions $\mathcal{D}(\Sigma)$ is representing them as vectors of length 2^n, where $n = |\mathsf{Var}|$.

A transformer $\Sigma \Rightarrow \Sigma$ can then be represented as a matrix $\mathbf{M}$ of dimensions $2^n \times 2^n$, where the row index σ_1 is the source state and the column index σ_2 is the destination state, so that $\mathbf{M}_{\sigma_1,\sigma_2}$ is the probability of transition from σ_1 to σ_2. The rows of the matrix are thus output distributions. Since each row of a matrix sums to at most 1, the matrix can be characterized as a right sub-stochastic (Markov) matrix.

Denote the matrix implementation of a distribution or transformer T as $\mathbf{M}_T$. The action of applying a state to a transformer is then vector-matrix multiplication, and the bind operator is matrix multiplication. The return operator lifts a state to a distribution by returning its point mass, which is the identity matrix.

$$\begin{aligned} \mathbf{M}_{T(\sigma)} &\coloneqq \sigma \cdot \mathbf{M}_T \\ \mathbf{M}_{\delta} &\coloneqq \mathbf{I} \\ \mathbf{M}_{T_1^\dagger \circ T_2} &\coloneqq \mathbf{M}_{T_2} \cdot \mathbf{M}_{T_1} \end{aligned}$$

The choice operator present in the conditional branch and the loop is implemented by matrix addition. Let $\mathbf{\Gamma}_e$ be a *guard matrix* such that $(\mathbf{\Gamma}_e)_{\sigma,\tau} = 1$ if $\sigma = \tau$ and $\sigma(e)$, or 0 otherwise. Then the implementation of the choice between T_1 if $\sigma(e)$, or T_2 otherwise is

$$\mathbf{\Gamma}_e \cdot \mathbf{M}_{T_1} + (\mathbf{I} - \mathbf{\Gamma}_e) \cdot \mathbf{M}_{T_2}$$

3.2 Transformers for Loops

A main contribution of this work is the effective derivation of transformers for both converging and diverging loops. Our partial order translates in meaning to matrix transformers, where $\mathbf{M}_1 \leq \mathbf{M}_2$ if their elements are so ordered pointwise. Then, letting $\mathbf{\Gamma}_e$ be the guard matrix as above and $\mathbf{S} = \mathbf{M}_{[\![s]\!]}$, the matrix transformer corresponding to $[\![\texttt{while}\ (e)\ s]\!]$ is a solution of the functional

$$f(\mathbf{X}) = (\mathbf{I} - \mathbf{\Gamma}_e) + \mathbf{\Gamma}_e \cdot \mathbf{S} \cdot \mathbf{X} \tag{1}$$

Now if $\mathbf{\Gamma}_e \cdot \mathbf{S} - \mathbf{I}$ is invertible, the fixed point of f may be computed simply by

$$\mathbf{X} = (\mathbf{\Gamma}_e \cdot \mathbf{S} - \mathbf{I})^{-1}(\mathbf{\Gamma}_e - \mathbf{I})$$

However, in general the system is under-constrained and there are multiple solutions, of which we must select the least. Our innovation is that in the under-constrained case we solve the following linear program, the unrolled definition of matrix multiplication and addition, that minimizes the fixed point:

$$\begin{aligned} \text{minimize} \quad & \sum_{\sigma,\tau \in \Sigma} \mathbf{X}_{\sigma,\tau} \\ \text{subject to} \quad & \mathbf{X}_{\sigma,\tau} = (\mathbf{I} - \mathbf{\Gamma}_e)_{\sigma,\tau} + \sum_{\upsilon \in \Sigma} (\mathbf{\Gamma}_e \cdot \mathbf{S})_{\sigma,\upsilon} \cdot \mathbf{X}_{\upsilon,\tau} && \sigma, \tau \in \Sigma \\ & 0 \leq \mathbf{X}_{\sigma,\tau} \leq 1 && \sigma, \tau \in \Sigma \\ & \sum_{\tau \in \Sigma} \mathbf{X}_{\sigma,\tau} \leq 1 && \sigma \in \Sigma \end{aligned}$$

Certainly, the least fixed point of f is indeed minimal for the objective above, since if there existed $\mathbf{X}$ with a smaller value for the objective it would necessarily be lesser in the point-wise ordering. Furthermore, the linear program is always feasible since the fixed point of f exists and satisfies equation (1). Finally, because loops have a nested structure within programs, it is always possible to numerically determine $\mathbf{S}$ before solving loops that contain s. Loops may be solved in an inside-out traversal, with a linear system generated and solved for a child loop before being substituted into a parent loop. Thus, only the $\mathbf{X}_{\sigma,\tau}$ are variables in the LP, and all other identifiers are constants.

3.3 Inference Example

We now demonstrate the framework on the following program, which by manual inspection should have 0.5 probability of terminating with `b1` false and `b2` true, and 0.5 probability of nontermination:

```
bool b1, b2;
b1 = Bernoulli(0.5);
while (b1 || !b2) {
    b2 = Bernoulli(0.5);
}
```

By the analysis, the transformer of the overall program is the composition of the first statement, $\texttt{b1} \sim \mathcal{B}(0.5)$, and the `while`-loop. All matrices in the analysis are given in Figure 6. Enumerating the four states of the program with `b1` being the low-order bit, the matrix for the sample statement is $\mathbf{M}_1$. The guard of the loop, $\texttt{b1} \vee \neg\texttt{b2}$, has a corresponding matrix $\mathbf{\Gamma}$, and the body of the loop, $\texttt{b2} \sim \mathcal{B}(0.5)$, has matrix $\mathbf{S}$. The matrix $\mathbf{\Gamma} \cdot \mathbf{S} - \mathbf{I}$ is singular, so we solve the linear program as given in Section 3.2. Its numeric least solution is $\mathbf{M}_2$, and the product $\mathbf{M}_1 \cdot \mathbf{M}_2$ gives the overall transformer for the program.

Finally, the first row d of that transformer gives the output distribution assuming that variables are initialized to false (as they are, in this language), which confirms our manual analysis of the program: 0.5 probability of `b2` only being true, and the remainder probability, 0.5, being that of nontermination.

4 Optimized and Incremental Queries

In the inference algorithm, the data structures for distributions and transformers can quickly expand to be infeasible for time and memory. In the worst case, the number of LP variables generated is $O(2^{2n}m)$ where m is the number of statements in the program. To reduce the complexity, we reduce the size of intermediate data by means of information from a dataflow analysis, storing at each step only the data strictly necessary to analyze a given subprogram. We assume that the user interacts with the inference system by providing the probabilistic program and a query consisting of the variables in which they are interested.

$$\mathbf{M}_1 = \begin{bmatrix} 0.5 & 0.5 & 0 & 0 \\ 0.5 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \end{bmatrix} \quad \mathbf{\Gamma} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{S} = \begin{bmatrix} 0.5 & 0 & 0.5 & 0 \\ 0 & 0.5 & 0 & 0.5 \\ 0.5 & 0 & 0.5 & 0 \\ 0 & 0.5 & 0 & 0.5 \end{bmatrix}$$

$$\mathbf{\Gamma} \cdot \mathbf{S} - \mathbf{I} = \begin{bmatrix} -0.5 & 0 & 0.5 & 0 \\ 0 & -0.5 & 0 & 0.5 \\ 0 & 0 & -1 & 0 \\ 0 & 0.5 & 0 & -0.5 \end{bmatrix} \quad \mathbf{M}_2 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{M}_1 \cdot \mathbf{M}_2 = \begin{bmatrix} 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0.5 & 0 \end{bmatrix}$$

$$d = \begin{bmatrix} 0 & 0 & 0.5 & 0 \end{bmatrix}$$

Fig. 6: Matrices used in the example analysis.

The central idea of the optimization is that not all variables are relevant to the analysis of every subprogram. Consider, for example, a statement (program fragment) including the variables `b1`, `b2`, and `b3`, and which ends with the following line:

```
b3 = Bernoulli(0.25);
```

Suppose the user issues a query for the effect of the statement on `b3`, given the old values of all variables as input. In Bayesian form, this query could be written as a set of posterior probabilities, each conditioned on a possible input from the prior. One element of the set would be $\mathbb{P}(\texttt{b3} \mid \texttt{b1} \wedge \texttt{b2})$. In total, there are $2^3 = 8$ conditional probabilities to evaluate. However, if the user only wishes to know information about `b3`, we may not need to consider the other two variables at all. That is, here the last line informs us that after the statement executes, $\mathbb{P}(\texttt{b3}) = 0.25$ unconditionally, saving the work of enumerating the entire state space, and allowing us to skip all but the last statement altogether.

By contrast, variables that are conditionally dependent require more resources to store than variables that are independent. Suppose the statement instead ended with this line:

```
b3 = b1 || b2;
```

For the first program, to represent the effect on `b3`, a matrix of size 2×2 suffices. But for the second program, since `b3` is conditionally dependent on both other variables, a matrix of size $2^3 \times 2^3$ is required, capturing all combinations of the three variables. Conditional dependencies thus introduce a potentially exponential blowup in the size of intermediate data.

With detailed information about which variable dependencies, it becomes possible to take an incremental approach to inference where a query that examines few components of a program need not waste work in analyzing the remainder. This optimization is enabled by the dependent variable analysis that follows.

4.1 Dependent Variable Analysis

The goal of the *dependent variable analysis* is to compute, for each program fragment, a fine-grained picture of its variable interactions. Given a statement and a set of variables whose distributions we wish to infer, this analysis informs us which variables are necessary as inputs and what the dependency relationships between variables are.

We formulate a backward dataflow analysis, presented through inference rules that unroll loops in the program until saturation of variable relationships is reached. Some of the following definitions are similar to the notion of use-def chains in compiler construction, but with alterations for the probabilistic context.

Definition 5 (Use). *An expression e uses a variable x, denoted $\mathsf{use}(x, e)$, if x appears in e.*

The used variables of an expression e, $\mathsf{used}(e)$, are all x such that $\mathsf{use}(x, e)$. In the probabilistic setting, sampling and assignment are not the only means of modifying a variable. A loop that may diverge can affect a distribution, since it reduces the total termination probability of the program. Also, even if the marginal distribution of a variable does not change through a statement, since we are interested in conditional dependencies, we must account for changes to joint distributions with other variables.

Definition 6 (Def). *A statement s defines a variable x, denoted $\mathsf{def}(s, x)$, if the execution of s may affect the normalized marginal distribution of x, or the normalized joint distribution of x with any other variable.*

$$\frac{}{\mathsf{def}(x \sim \mathcal{B}(p), x)}\ (\textsc{Def-Sample})$$

$$\frac{\mathsf{def}(s_1, x) \vee \mathsf{def}(s_2, x)}{\mathsf{def}(\texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2, x)}\ (\textsc{Def-C}) \qquad \frac{\mathsf{def}(s, x)}{\mathsf{def}(\texttt{while}\ (e)\ s, x)}\ (\textsc{Def-L})$$

$$\frac{\mathsf{def}(s_1, x) \vee \mathsf{def}(s_2, x)}{\mathsf{def}(s_1;\ s_2, x)}\ (\textsc{Def-Seq}) \qquad \frac{\mathsf{dep}(x', s, x)}{\mathsf{def}(s, x)}\ (\textsc{Def-Dep})$$

Fig. 7: Rules for definition judgment.

As expected, sampling defines a variable, and also statements define variables defined in sub-statements. Rules are not present for assignment, conditionals, and loops, because the next relation, *dependency*, provides information about those constructs. The last rule states that the target of a *dependency chain* is considered defined by a statement.

Definition 7 (Dependency). *A statement s introduces a dependency by defining variable x using information in a preceding variable x', denoted $\mathsf{dep}(x', s, x)$.*

$$\frac{\mathsf{use}(x', e)}{\mathsf{dep}(x', x \leftarrow e, x)}\ (\textsc{Dep-Assign})$$

$$\frac{\mathsf{dep}(x', s_1, x) \vee \mathsf{dep}(x', s_2, x)}{\mathsf{dep}(x', \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2, x)}\ (\textsc{Dep-C}) \qquad \frac{\mathsf{use}(x', e) \qquad \mathsf{def}(s_1, x) \vee \mathsf{def}(s_2, x)}{\mathsf{dep}(x', \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2, x)}\ (\textsc{Dep-C-F})$$

$$\frac{\mathsf{def}(s_1, x) \qquad \neg\, \mathsf{def}(s_2, x)}{\mathsf{dep}(x, \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2, x)}\ (\textsc{Dep-C-Id}_1) \qquad \frac{\mathsf{def}(s_2, x) \qquad \neg\, \mathsf{def}(s_1, x)}{\mathsf{dep}(x, \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2, x)}\ (\textsc{Dep-C-Id}_2)$$

$$\frac{\mathsf{dep}(x', \texttt{if}\ (e)\ \{s;\ \texttt{while}\ (e)\ s\}, x)}{\mathsf{dep}(x', \texttt{while}\ (e)\ s, x)}\ (\textsc{Dep-L})$$

$$\frac{\mathsf{dep}(x', s_1, x)}{\mathsf{dep}(x', s_1;\ s_2, x)}\ (\textsc{Dep-Head}) \qquad \frac{\neg\, \mathsf{def}(s_1, x') \qquad \mathsf{dep}(x', s_2, x)}{\mathsf{dep}(x', s_1;\ s_2, x)}\ (\textsc{Dep-Tail})$$

$$\frac{\mathsf{dep}(x'', s_1, x') \qquad \mathsf{dep}(x', s_2, x)}{\mathsf{dep}(x'', s_1;\ s_2, x)}\ (\textsc{Dep-Chain}) \qquad \frac{\mathsf{dep}(x', s, x)}{\mathsf{dep}(x', s, x')}\ (\textsc{Dep-Fwd})$$

Fig. 8: Rules for dependency judgment.

The notion of dependency is information flow from one variable into another. Assignments cause information flow from the source to the destination. Conditionals propagate the dependencies of their bodies, and also introduce dependencies between the condition expression and the variables defined in the bodies. If a variable is defined in only one branch of a conditional, then the other branch preserves its existing value, so the conditional as a whole relies on the old value of that variable. Loops are unrolled to conditional statements, and the (Dep-L) rule must be iterated until saturation, when additional unrollings do not introduce any new dependencies.

Next, dependencies are propagated for the first statement of a sequence. However, we conditionally propagate dependencies in the second position, only if the source of the dependency is not defined by the first statement. This design ensures that intermediate variables remain local to a statement. On the other hand, if s_1 defines an intermediate variable used by s_2 but requires an input to do so, we transitively chain that dependency into s_2. Finally, any input for a statement is also forwarded to an output, so that information is not lost.

Definition 8 (Available). *The available set of variables of a statement s, $\mathsf{avail}(s)$, contains all x such that $\mathsf{def}(s, x)$.*

The inference algorithm must query a statement for exactly the variables that it makes available that are relevant to the analysis. This ensures that every possible distribution update is eventually seen by the inference.

Definition 9 (Needed). *The needed variables of a statement s for a variable set $v \subseteq \mathsf{avail}(s)$, $\mathsf{need}(s, v)$, is the smallest set of variables needed to give meaning to the variables in v, such that $x' \in \mathsf{need}(s, v)$ iff $\mathsf{dep}(x', s, x)$ for some $x \in v$.*

4.2 Dependency-Sensitive Queries

Given a program fragment s and a set of variables v, we now have a precise characterization of how to query the statement for the variables: as long as $v \subseteq \mathsf{avail}(s)$, we only need $\mathsf{need}(s, v)$ as inputs to obtain the distribution of v.

We now show that the analysis is sound, that is, it faithfully describes a program fragment's inputs and outputs. Given a statement s and variable x, let $s(x)$ be the normalized distribution of x after the execution of s.

Theorem 1 (Soundness of Dependency Analysis). *For any statement s and variable x, if $x \in \mathsf{avail}(s)$, then every variable $x' \notin \mathsf{need}(s, x)$ is conditionally independent of $s(x)$ given x. Otherwise, $s(x) = x$.*

The full proof is in Appendix 1.

4.3 Domain-Sensitive Transformers

So far, transformers have all been assumed to take Σ as their domain, but the optimized analysis will now produce transformers on subsets of the variables. To account for this, we revisit the idea of composition of two transformers X and Y, now with generalized sets of input and output variables:

$$X : S_1 \Rightarrow T_1, \quad Y : S_2 \Rightarrow T_2$$

To represent the composition $Y \circ X$, we will still use matrix multiplication $\mathbf{M}_X \cdot \mathbf{M}_Y$, which is only mathematically defined if $T_1 = S_2$. Otherwise, we must reshape the matrices so that they become compatible. In particular, we may need to supply Y with more inputs than are made available in X, and preserve all outputs of X even if they are not relevant to Y. Written in set notation, the expected dimensions of the new composition are:

$$Y \circ X : S_1 \cup (S_2 \setminus T_1) \Rightarrow T_2 \cup (T_1 \setminus S_2)$$

Likewise, for the choice operator $X+Y$, if one operand makes a variable available that the other does not, then the variable is preserved along one branch, meaning it must be obtained as input and forwarded to the output. We must therefore obtain as inputs the symmetric difference $\triangle$ of the outputs. In set notation, the dimensions are:

$$X + Y : S_1 \cup S_2 \cup (T_1 \triangle T_2) \Rightarrow T_1 \cup T_2$$

In the underlying representation of these transformers, we must physically reshape operand matrices by adding rows and columns, and then apply the conventional matrix multiplication or addition. A reshape operation cannot fundamentally alter the probabilistic meaning of the transformer, so to add a set of

variables $v \subseteq \mathsf{Var}$ as outputs, they must also be added as inputs. No change must occur on the distributions of these new variables, which is a property only held by the identity transformer. Therefore, $\delta(v)$, corresponding to an identity matrix of size $2^{|v|} \times 2^{|v|}$, is combined with the original matrix by the Kronecker product, producing a transformer equal in meaning to the original, but over more variables. Likewise, a set of variables v may be added purely as inputs, and not outputs, by the operation of a Kronecker product with a column vector of all ones with length $2^{|v|}$. Effectively, this product duplicates rows of the original matrix without changing its probabilistic meaning. These two forms of sound matrix reshape allow us to implement the modified composition and choice transformers.

4.4 Query Implementation

We now specify the dependency-sensitive query algorithm. Given an expression e and a set of variables $v \subseteq \mathsf{Var}$, define the transformer

$$\texttt{guard}\ e\ v : \mathsf{used}(e) \Rightarrow v \cap \mathsf{used}(e)$$

to be the guard transformer $\mathbf{\Gamma}_e$ as in Section 3.2, with outputs marginalized to those within v. Now given a statement s and a set of query variables $v \subseteq \mathsf{avail}(s)$, define the query transformer

$$\texttt{query}\ s\ v : \mathsf{need}(s, v) \Rightarrow v$$

which computes the transformer for s efficiently, with input states restricted to variables on which the output depends. We will also use the following macros:

$$\mathsf{of}(v, s) \coloneqq v \cap \mathsf{avail}(s)$$
$$\mathsf{with}(v, s) \coloneqq v \cup \mathsf{need}(s, \mathsf{of}(v, s))$$

The first restricts variables to only those available in a statement, and the second expands a set of variables to include additional ones that a statement needs.

The full implementation of the query is shown in Figure 9. In the algorithm, the sequencing rule communicates the minimization of the state space. Successor statements are queried for each of the variables that are available; other variables have their query forwarded to a predecessor statement. Also, only the needed variables are propagated backward, and only the variables that are queried and available forward. This propagation performs a form of dead code elimination, where unused definitions are skipped during the analysis altogether. With the loop case, we must expand the query to include all variables needed within the loop, so that we may continue iterating the loop.

When the query is executed on an empty variable set, since $|\emptyset| = 0$, we would expect to represent $\emptyset \Rightarrow \emptyset$ as a 1×1 matrix, which is simply a scalar. For statements that terminate almost surely, the value of this scalar is equal to 1, and has no effect when composed with any other transformer. The query on the empty set may be elided in such a case. If a statement may possibly not terminate (i.e. it structurally contains a loop or observation), then the value of the scalar may be less than one, in which case it cannot be elided.

$$
\begin{aligned}
\texttt{query}\ (x \leftarrow e)\ (\{x\} \cup v) &\coloneqq \lambda\sigma.\ \lambda\tau.\ \tau = v[x \leftarrow \sigma(e)] \\
\texttt{query}\ (x \sim \mathcal{B}(p))\ \{x\} &\coloneqq \lambda\sigma.\ \lambda\tau.\ \texttt{if}\ \tau = \{x\}\ \texttt{then}\ p\ \texttt{else}\ 1-p \\
\texttt{query}\ (\texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2)\ v &\coloneqq \texttt{guard}\ e\ \mathsf{with}(v, s_1) \cdot \texttt{query}\ s_1\ \mathsf{of}(v, s_1) \\
&\quad + \texttt{guard}\ (\neg e)\ \mathsf{with}(v, s_2) \cdot \texttt{query}\ s_2\ \mathsf{of}(v, s_2) \\
\texttt{query}\ (\texttt{while}\ (e)\ s)\ v &\coloneqq \texttt{let}\ n \coloneqq \mathsf{need}(\texttt{while}\ (e)\ s, v)\ \texttt{in} \\
&\quad \mu\ (X : n \Rightarrow v) \\
&\quad\ \ \texttt{guard}\ e\ \mathsf{with}(n, s) \cdot \texttt{query}\ s\ \mathsf{of}(n, s) \cdot X \\
&\quad\ \ + \texttt{guard}\ (\neg e)\ v \cdot \delta(v) \\
\texttt{query}\ (s_1;\ s_2)\ v &\coloneqq \texttt{query}\ s_1\ (\mathsf{of}(\mathsf{need}(s_2, \mathsf{of}(v, s_2)), s_1) \cup (v \setminus \mathsf{avail}(s_2))) \\
&\quad \cdot \texttt{query}\ s_2\ \mathsf{of}(v, s_2)
\end{aligned}
$$

Fig. 9: Dependency-sensitive query algorithm implementation. The query for every unspecified combination of s and v is the identity transformer.

4.5 Soundness

Since the optimized query algorithm is an adaptation of the algorithm from Section 3 to become dependency-sensitive, the soundness of the algorithm rests largely on the correctness of the variable analysis. Combined with Theorem 1, the next statement shows that the optimized query yields a transformer that correctly specifies the output distributions of the query variables. In the following, transformers are considered equal point-wise (giving equal distributions on equal inputs).

Theorem 2 (Soundness of Query). *Given statement s and variables $v \subseteq \mathsf{avail}(s)$, $\texttt{query}\ s\ v$ is equal to $[\![s]\!]$, marginalized to dimensions $\mathsf{need}(s, v) \Rightarrow v$.*

The full proof is in Appendix 2.

5 Implementation

We have implemented the Bayesian Loop Transformers inference algorithm in about 2000 lines of OCaml. The implementation supports input in the form of programs as well as Bayesian networks in the Bayesian Interchange Format [8]. We use the CLP linear program solver [6] and the Owl linear algebra package [33].

With a Bayesian network as input, we perform a simple transformation into the PPL. Nodes in the network are converted to categorical random variables, which in turn are encoded bitwise into Boolean variables. Initial variable distributions are converted to a program prologue that samples all known variables according to their specified distributions. Edges are converted to conditionals that test the inputs to a node, with body statements that resample the node being defined according to the weights assigned to the edge.

5.1 Matrix Implementation

Of central importance in the implementation is the choice of matrix data structure. A simple choice would be representing transformers as dense 2D matrices. An alternative is sparse storage of the factors of a matrix Kronecker product, exploiting our dependent variable analysis. As described previously, a transformer contains information about output distributions as conditioned on inputs. For example, suppose at some point in the program, we wish to compute the transformer that takes as input variables x_1, x_2, x_3, x_4 and outputs variables x_5, x_6. The dense matrix representation of $\{x_1, x_2, x_3, x_4\} \Rightarrow \{x_5, x_6\}$ would require a matrix of size $2^4 \times 2^2$. Absent finer-grained information about variable dependencies, we can do no better.

However, suppose that we know x_5 is dependent only on x_1 and x_3, and x_6 only on x_2 and x_4. As suggested in Section 4, when variables in the output are conditionally independent, their joint distributions need not be parameterized with respect to each other. Instead, we can compute a separate transformer for each set of mutually dependent variables, joined by Kronecker product. Crucially, we will leave the Kronecker product unevaluated for as long as possible and only store the factors, until a later stage of the program where independence is lost. In the example, we may represent the transformer as the Kronecker product of two distributions $\{x_1, x_3\} \Rightarrow \{x_5\}$ and $\{x_2, x_4\} \Rightarrow \{x_6\}$, which is two matrices of size $2^2 \times 2^1$, and a large savings in time and memory usage.

In practice, this proposed Kronecker representation has the potential to greatly improve performance of certain analyses. However, there are two drawbacks. The first is the increased complexity of the implementation, which must identify when to explicitly compute the products when matrices are no longer disjoint over their domains and ranges. The simpler dense matrix design was eventually adopted for our BLT implementation as it was easier to verify. Furthermore, an important property of Kronecker products is that they commute with matrix multiplication, or sequential composition in our inference, but not with matrix addition, or conditional choice. Conditional branches in the analysis therefore force the sparsity to be lost whenever they are encountered.

We can generalize the data structure question further to other possible transformer representations that leverage dependency information. In symbolic model checkers, the algebraic decision diagram (ADD), a generalization of the binary decision diagram, has been used to represent symbolic program states and probability distributions for inference and related problems [5] [10]. In the best case, ADDs can enable an exponential order of resource saving over dense matrices, by sharing repeated substructures [25]. One factor that greatly impacts the performance of ADD-based algorithms is the choice of ordering of variables, since a pessimal ordering results in no efficiency gain over dense matrices. Determining the optimal ordering of variables is **NP**-hard [25], so in practice various heuristics are used. Though we focused on the fundamental inference mechanism and did not implement ADDs, our dependent variable analysis may aid the determination of variable ordering, aiding potential ADD-based implementations.

6 Related Work

Many inference tools for probabilistic programs and modeling exist, including Stan [2], Hakaru [27], and Infer.NET [26]. These three, representative of real-world implementations, utilize suites of approximate inference algorithms with different performance characteristics. For example, Stan uses a modified Hamiltonian Monte Carlo sampler that adapts continuously during execution. Hakaru uses Metropolis-Hastings sampling in conjunction with computer algebra to convert programs to distributions that can be sampled efficiently. Finally, Infer.NET supports a wide range of approximate inference algorithms, including expectation propagation, variational message passing, and Gibbs sampling. The sheer number of different approximate techniques with different advantages can make a full comparison overwhelming. In general, though these tools perform reasonably in practice, unlike BLT they make few formal guarantees of the quality of their convergence. Very few, among which are Hur et al. [20], attempt to formalize the correctness of the sampling with respect to program semantics.

Within the realm of exact inference, PSI [13] is a prominent work that is built around symbolic distributions. PSI relies extensively on symbolic integration and algebraic optimization, and achieves a fully symbolic inference output. By comparison, while BLT also performs exact inference, it relies on linear algebra and linear programming, so increased precision requires support from the underlying numerical methods. However, PSI does not support unbounded loops, except through repeated iteration of the loop-solving procedure until an externally-specified tolerance threshold is met. BLT is able to statically generate a fixed set of constraints for arbitrary unbounded loops, and the solution of the resulting constraints is delegated to a heavily optimized LP solver. No developer intervention is necessary to perform loop inference, and performance is not impacted by the probability of termination.

Support for unbounded loops and nonterminating programs is relatively uncommon in systems for inference. R2 [28], a Metropolis-Hastings sampler, supports unbounded loops in theory but requires annotations of loop invariants by the user. The implementation performs unrolling of loops to a fixed number of iterations, thereby approximating the result, unless the invariant is available. Automatic inference of the invariant is not yet possible in their system. By contrast, BLT fully solves loops without user intervention and requires no unrolling in its theoretical basis or implementation. In a related work, Claret et al. [5] presented an approach to exact static inference for loopy programs using dataflow analysis and symbolic execution. While they maintain a symbolic program state distribution that is repeatedly updated during symbolic execution, BLT instead converts program statements to concrete transformers that are solved together. Consequently, their loop evaluation is also by iteration of the loop until a threshold is met, whereas BLT requires no extrinsic threshold and solves loops exactly as linear programs.

Other work influenced the design of BLT, especially regarding loop inference. Batz et al. [1] showed a means of determining the expected sampling time of a Bayesian network using weakest pre-expectations, which is closely related to

inference. However, they limit loops to the top level of the program and focus on loops with independence properties between iterations. Sankaranarayanan et al. have analyzed probabilistic programs with loops from the perspective of invariant synthesis through the generation of martingales, which allows them to reason about the nontermination probability of programs [3]. They have also developed a technique to synthesize expectation invariants of loops that relies on classic methods in abstract interpretation [4], as well as a static analysis that chooses a subset of paths through a program to explore and then establish bounds on inference queries across the program as a whole [30]. Probabilistic NetKAT [10], though not designed as a direct mechanism for inference, also encounters nonterminating probabilistic programs and represents them using Markov chain constructions.

Finally, a large challenge for BLT is the representation and optimization of large state spaces. Here, symbolic model checkers such as PRISM [24] and Storm [9] share similar challenges to inference, even though their focus is typically not probabilistic programming but rather checking of properties on Markov chains and similar structures. They use algebraic decision diagrams and other compact data structures for large problem instances but fall back to sparse and dense 2D matrices to avoid large constant overhead factors when instances are smaller. Also, Holtzen et al. [19] recently discussed efficient compilation of probabilistic programs, exploiting variable independence similarly to our analysis.

7 Evaluation

We executed the inference procedure on a collection of Bayesian networks from the Bayesian Network Repository [32]. We evaluated the time to complete inference for BLT against two state-of-the-art inference tools, PSI [13] and R2 [28]. PSI operates by exact symbolic inference and R2 by optimized Markov Chain Monte Carlo sampling. We also compared BLT with the dependent variable optimization to a baseline BLT implementation without. Experiments were performed on a system with 16 GB of RAM and a 2.5 GHz Intel Core i7 processor.

The timing information we gathered is displayed in Table 1. A total of 22 networks were available for testing, but most were sufficiently large such that no reasonably written query could terminate for any of the three tools, and so they are not displayed in the table. We found that on the set of small networks in the repository, with up to 11 nodes, BLT significantly outperformed both PSI and R2. In particular, we were able to infer 4 of the 5 networks nearly instantaneously. For larger networks, PSI and R2 did not complete inference in a reasonable period of time and so no timing information for them is available. BLT, however, was able to handle limited queries on networks of up to 441 nodes in a few seconds, due to its incremental inference capability.

Finally, though we focused on comparing to tools with similar organization and feature sets, alternative tools specific to Bayesian networks generally outperform inference on networks embedded into probabilistic programs. The package `bnlearn` [31], for example, takes a large number of fast samples to compute

Table 1: Inference time for Bayesian networks in seconds. Column n represents the number of total nodes in the network, s the number of edges, and v the number of nodes in the query issued. $\times$ denotes a trial that did not terminate or produced an error.

Network	n	s	v	base	BLT	PSI	R2
`cancer`	5	4	5	0.17	0.01	1.03	4.33
`earthquake`	5	4	5	0.13	0.01	1.16	3.50
`survey`	6	6	6	0.79	0.02	2.30	5.50
`sachs`	11	17	3	$\times$	5.97	$\times$	6.50
`asia`	8	8	8	49.70	0.05	4.59	3.86
`alarm`	37	46	4	$\times$	2.30	$\times$	$\times$
`insurance`	27	52	3	$\times$	3.80	$\times$	$\times$
`hepar2`	70	123	5	$\times$	8.13	$\times$	$\times$
`win95pts`	76	112	2	$\times$	1.97	$\times$	$\times$
`andes`	223	338	2	$\times$	4.49	$\times$	$\times$
`pigs`	441	592	1	$\times$	1.61	$\times$	$\times$

approximate distributions to arbitrary user-specified precision, and its default setting outperforms all three of the benchmarked tools by one to two orders of magnitude. Such a comparison is not entirely fair since many optimizations exist for Bayesian networks specifically, but we mention it here for completeness.

8 Conclusion and Future Work

We have presented Bayesian Loop Transformers, a general approach to performing exact static inference over probabilistic programs with categorical variables and unbounded loops. It includes a fruitful dependent variable analysis that enables optimization of inference based on the structure of programs and user queries. The analysis also supports nonterminating programs via linear programming with strong guarantees of convergence.

A natural generalization of BLT allows analysis on procedural languages with first-order functions. A function may be treated as a program fragment with a set of fixed argument variables and a single return variable, which has a corresponding transformer that is substituted whenever the function is called. Nonterminating recursive functions are also related to the fixed point construction for loops, as transformers that refer to themselves at recursive call sites. So-called linear recursive programs, where each path through any function makes at most one recursive call, are similar to loops, and the existing LP solution mechanism may perform inference on them without modification. The analysis of more complex recursive programs and higher-order functional programs is ongoing work.

9 Acknowledgements

The authors wish to thank Di Wang for contributions and feedback throughout development, and David Kahn for solutions to the fixed-point equation.

References

1. Batz, K., Kaminski, B.L., Katoen, J.P., Matheja, C.: How long, O Bayesian network, will I sample thee? In: Ahmed, A. (ed.) Programming Languages and Systems. pp. 186–213. Springer International Publishing, Cham (2018)
2. Carpenter, B., Gelman, A., Hoffman, M., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., Riddell, A.: Stan: A probabilistic programming language. Journal of Statistical Software, Articles **76**(1), 1–32 (2017). https://doi.org/10.18637/jss.v076.i01, https://www.jstatsoft.org/v076/i01
3. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: CAV (2013)
4. Chakarov, A., Sankaranarayanan, S.: Expectation invariants for probabilistic program loops as fixed points. In: SAS (2014)
5. Claret, G., Rajamani, S., Nori, A., Gordon, A., Borgström, J.: Bayesian inference using data flow analysis. pp. 92–102. ACM (August 2013), https://www.microsoft.com/en-us/research/publication/bayesian-inference-using-data-flow-analysis-2/
6. COIN-OR Foundation: COIN-OR linear programming solver (2019), https://projects.coin-or.org/Clp
7. Cooper, G.F.: The computational complexity of probabilistic inference using Bayesian belief networks. Artificial Intelligence **42**(2), 393 – 405 (1990). https://doi.org/10.1016/0004-3702(90)90060-D, http://www.sciencedirect.com/science/article/pii/000437029090060D
8. Cozman, F.G.: Bayesian Interchange Format (1998), http://www.cs.cmu.edu/afs/cs/user/fgcozman/www/Research/InterchangeFormat/
9. Dehnert, C., Junges, S., Katoen, J.P., Volk, M.: A storm is coming: A modern probabilistic model checker (02 2017)
10. Foster, N., Kozen, D., Mamouras, K., Reitblatt, M., Silva, A.: Probabilistic NetKAT. In: Thiemann, P. (ed.) Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9632, pp. 282–309. Springer (2016). https://doi.org/10.1007/978-3-662-49498-1_12
11. Freer, C.E., Roy, D.M., Tenenbaum, J.B.: Towards common-sense reasoning via conditional simulation: legacies of turing in artificial intelligence. CoRR **abs/1212.4799** (2012), http://arxiv.org/abs/1212.4799
12. Ge, H., Xu, K., Ghahramani, Z.: Turing: Composable inference for probabilistic programming. In: Storkey, A.J., Pérez-Cruz, F. (eds.) International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain. Proceedings of Machine Learning Research, vol. 84, pp. 1682–1690. PMLR (2018), http://proceedings.mlr.press/v84/ge18b.html
13. Gehr, T., Misailovic, S., Vechev, M.: Psi: Exact symbolic inference for probabilistic programs. vol. 9779, pp. 62–83 (07 2016). https://doi.org/10.1007/978-3-319-41528-4_4
14. Ghahramani, Z.: Probabilistic machine learning and artificial intelligence. Nature **521**(7553), 452 (2015)
15. Giry, M.: A categorical approach to probability theory, pp. 68–85 (11 2006). https://doi.org/10.1007/BFb0092872

16. Goodman, N.D., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: a language for generative models. CoRR **abs/1206.3255** (2012), http://arxiv.org/abs/1206.3255
17. Goodman, N.D., Stuhlmüller, A.: The Design and Implementation of Probabilistic Programming Languages. http://dippl.org (2014), accessed: 2019-4-5
18. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: Proceedings of the on Future of Software Engineering. pp. 167–181. FOSE 2014, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2593882.2593900, http://doi.acm.org/10.1145/2593882.2593900
19. Holtzen, S., Qian, J., Millstein, T., Van den Broeck, G.: Factorized exact inference for discrete probabilistic programs. In: Languages for Inference (2009), http://starai.cs.ucla.edu/slides/LAFI19-exactdiscrete.pdf
20. Hur, C.K., Nori, A., Rajamani, S.: A provably correct sampler for probabilistic programs. In: Foundations of Software Technology and Theoretical Computer Science (FSTTCS). Leibniz International Proceedings in Informatics (December 2015), https://www.microsoft.com/en-us/research/publication/a-provably-correct-sampler-for-probabilistic-programs/
21. Kaminski, B.L., Katoen, J.: On the hardness of almost-sure termination. CoRR **abs/1506.01930** (2015), http://arxiv.org/abs/1506.01930
22. Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning. The MIT Press (2009)
23. Kulkarni, T.D., Kohli, P., Tenenbaum, J.B., Mansinghka, V.: Picture: A probabilistic programming language for scene perception. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (June 2015)
24. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proc. 23rd International Conference on Computer Aided Verification (CAV'11). LNCS, vol. 6806, pp. 585–591. Springer (2011)
25. Miner, A., Parker, D.: Symbolic representations and analysis of large probabilistic systems. vol. 2925, pp. 296–338 (01 2004). https://doi.org/10.1007/978-3-540-24611-4_9
26. Minka, T., Winn, J., Guiver, J., Zaykov, Y., Fabian, D., Bronskill, J.: Infer.NET 0.3 (2018), Microsoft Research Cambridge. http://dotnet.github.io/infer
27. Narayanan, P., Carette, J., Romano, W., Shan, C., Zinkov, R.: Probabilistic inference by program transformation in hakaru (system description). In: International Symposium on Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings. pp. 62–79. Springer (2016). https://doi.org/10.1007/978-3-319-29604-3_5
28. Nori, A., Hur, C.K., Rajamani, S., Samuel, S.: R2: An efficient MCMC sampler for probabilistic programs. AAAI (July 2014), https://www.microsoft.com/en-us/research/publication/r2-an-efficient-mcmc-sampler-for-probabilistic-programs/
29. Salvatier, J., Wiecki, T.V., Fonnesbeck, C.: Probabilistic programming in python using PyMC3. PeerJ Computer Science **2**, e55 (apr 2016). https://doi.org/10.7717/peerj-cs.55, https://doi.org/10.7717/peerj-cs.55
30. Sankaranarayanan, S., Chakarov, A., Gulwani, S.: Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths (June 2013), https://www.microsoft.com/en-us/research/publication/static-analysis-probabilistic-programs-inferring-whole-program-properties-finitely-many-paths/
31. Scutari, M.: Learning bayesian networks with the bnlearn R package. Journal of Statistical Software **35**(3), 1–22 (2010). https://doi.org/10.18637/jss.v035.i03

32. Scutari, M.: Bayesian Network Repository (2017), http://www.bnlearn.com
33. Wang, L.: Owl: A general-purpose numerical library in OCaml. CoRR **abs/1707.09616** (2017), http://arxiv.org/abs/1707.09616

A Proofs

A.1 Proof of Theorem 1

We first show that if s may affect the normalized marginal distribution of x, or the normalized joint distribution of x with any other variable, then $x \in \mathsf{avail}(s)$.

Proof. Proceed by induction on the structure of s.

- Case `skip`: `skip` may not affect the distribution of any variable, so this case is vacuous.
- Case $x \leftarrow e$: The assignment affects the marginal distribution of x. By rule (Dep-Assign), $x \in \mathsf{avail}(s)$. It also affects the joint distribution of every $x' \in \mathsf{used}(e)$ with x. By rules (Dep-Assign) and (Dep-Fwd), $x' \in \mathsf{avail}(s)$.
- Case $x \sim \mathcal{B}(p)$: The sample affects only the marginal distribution of x. By rule (Def-Sample), $x \in \mathsf{avail}(s)$.
- Case `if` (e) s_1 `else` s_2: The conditional affects every variable that is affected by one branch. By rule (Def-C), if $x \in \mathsf{avail}(s_1)$ or $x \in \mathsf{avail}(s_2)$ then $x \in \mathsf{avail}(s)$. It also affects the joint distribution of each variable used by e with variables affected by the branches. By rules (Dep-C-F) and (Dep-Fwd), if $x \in \mathsf{used}(e)$ and s_1 or s_2 affect any variables, then $x \in \mathsf{avail}(s)$.
- Case `while` (e) s_1: The meaning of the while loop is completely defined by its unrolling. If the guard e is true, the loop executes its body and repeats. This behavior is characterized by rule (Dep-L).
- Case s_1; s_2: The variables that may be affected by s_1 or s_2 are exactly those affected by s. Rule (Def-Seq) characterizes this behavior.

We next show that if $x \in \mathsf{avail}(s)$ and x' is required for s to give meaning to x, then $x' \in \mathsf{need}(s, \{x\})$.

Proof. Proceed by induction on the structure of s. Throughout each of the cases, $x \in \mathsf{avail}(s)$ is always needed to give meaning to itself, which is reflected in rule (Dep-Fwd).

- Case `skip`: Vacuous.
- Case $x \leftarrow e$: Those $x' \in \mathsf{used}(e)$ are exactly those required to give meaning to x. By rules (Dep-Assign), $x' \in \mathsf{need}(s, \{x\})$.
- Case $x \sim \mathcal{B}(p)$: No variables are required to give meaning to x, so vacuous.
- Case `if` (e) s_1 `else` s_2: By rule (Def-C), if $x' \in \mathsf{need}(s_1, \{x\})$ or $x' \in \mathsf{need}(s_2, \{x\})$, then $x' \in \mathsf{need}(s, \{x\})$. Other dependencies introduced are those between all variables in e and all variables defined in s. They are characterized by rule (Dep-C-F). Finally, the only dependencies remaining are that x is required to give meaning to itself if x is only defined in one of the branches and not both. This is reflected by rules (Dep-C-Id).

- Case `while` (e) s_1: As before, the meaning of the while loop is completely defined by its unrolling by rule (Dep-L).
- Case s_1; s_2: If some variable x' is required to give meaning to x, then a definition of x must be in s_1 or s_2. If s_1 requires x' to define x, then rule (Dep-Head) carries the dependency to s. If s_2 requires x', then a definition of x' either is or is not present in s_1. If not, then rule (Dep-Tail) carries the dependency to s_2. If so, then s_1 may require any number of dependencies x'' to give meaning to x'. For each x'', s requires x'' to give meaning to x, reflect in rule (Dep-Chain).

The combination of both shows the overall claim.

A.2 Proof of Theorem 2

Proof. We show that the operations performed by the optimized analysis correspond to the original algorithm taking all needed variables into account at every subexpression, and that all transformer operations have compatible dimensions. Proceed by induction on structure of the statement s. Let $v \subseteq \mathsf{avail}(s)$ be the variable set being queried.

- Case `skip`:
  s makes no variables available, so $v = \emptyset$. We have $\mathsf{need}(s, \emptyset) = \emptyset$, and the only total transformer from $\emptyset \to \emptyset$ is $\delta(\emptyset)$.
- Case $x \leftarrow e$:
  Either $x \in v$ or not. If $x \in v$, then $\mathsf{need}(s, v) = \mathsf{used}(e)$. Then, taking $\sigma \subseteq \mathsf{used}(e)$ and $\tau \subseteq v$, we have $\lambda\sigma.\ \lambda\tau.\ \tau = v[x \leftarrow \sigma(e)] : \mathsf{used}(e) \Rightarrow v$. If $x \notin v$, then $\mathsf{need}(s, v) = v \subseteq \mathsf{used}(e)$ and $\delta(v) : v \Rightarrow v$.
- Case $x \sim \mathcal{B}(p)$:
  Either $x \in v$ or not. If $x \in v$, then $\mathsf{need}(s, v) = \emptyset$. Also, $v = \{x\}$ since $\mathsf{avail}(s) = \{x\}$. Then, taking $\sigma = \emptyset$ and $\tau \subseteq v$, we have $\lambda\sigma.\ \lambda\tau.\ \texttt{if}\ \tau = \{x\}\ \texttt{then}\ p\ \texttt{else}\ 1 - p : \emptyset \Rightarrow v$. If $x \notin v$, then $v = \mathsf{need}(s, v) = \emptyset$ and $\delta(\emptyset) : \emptyset \Rightarrow \emptyset$.
- Case `if` (e) s_1 `else` s_2:
  Let $v_1 = \mathsf{of}(v, s_1)$, the subset of v available in s_1. Let $v_2 = \mathsf{of}(v, s_2)$. We query s_1 for v_1 and s_2 for v_2, the maximal set of variables that each makes available in v.
  So `query` $s_1\ v_1 : \mathsf{need}(s_1, v_1) \Rightarrow v_1$ and `query` $s_2\ v_2 : \mathsf{need}(s_2, v_2) \Rightarrow v_2$, and $v \cup \mathsf{need}(s_1, v_1) = \mathsf{with}(v, s_1)$ and $v \cup \mathsf{need}(s_2, v_2) = \mathsf{with}(v, s_2)$.
  Also, we have `guard` $e\ \mathsf{with}(v, s_1) : \mathsf{used}(e) \to \mathsf{used}(e) \cap \mathsf{with}(v, s_1)$ and `guard` $\neg e\ \mathsf{with}(v, s_2) : \mathsf{used}(e) \to \mathsf{used}(e) \cap \mathsf{with}(v, s_2)$.
  The composition of the guard and query for s_1 therefore takes as input every variable in $\mathsf{used}(e) \cup \mathsf{need}(s_1, v_1)$, since the output of the guard is a subset of its input. The output is $v \cap \mathsf{used}(e) \cup v_1$, since the guard contributes $v \cap \mathsf{used}(e)$, and the query contributes v_1 and any $x \in v$ is either in v_1 or not in $\mathsf{need}(s_1, v_1)$ since the query makes available all of its needed variables.

Likewise, the type of the second branch is $\mathsf{used}(e) \cup \mathsf{need}(s_2, v_2) \Rightarrow v \cap \mathsf{used}(e) \cup v_2$. The type of the overall sum is therefore $\mathsf{used}(e) \cup \mathsf{need}(s_1, v_1) \cup \mathsf{need}(s_2, v_2) \cup (v_1 \triangle v_2) \Rightarrow v \cap \mathsf{used}(e) \cup v_1 \cup v_2$.
Any variable made available by the conditional is made available by one of its branches or is in the guard. Furthermore, any variable needed by the conditional is in the guard, or needed by one branch, or defined only in one branch. Therefore, this type is equal to $\mathsf{need}(s, v) \Rightarrow v$.

– Case `while` (e) s_1:
Let $n = \mathsf{need}(s, v)$ as in the algorithm and assume $X : n \Rightarrow v$. We know n contains v, $\mathsf{need}(s_1, v)$, and $\mathsf{used}(e)$ as subsets since all of those variables will enter the needed set upon unrolling the loop.
We query s_1 for all variables of n it makes available. We similarly obtain every available input to s_1, $\mathsf{with}(n, s_1)$, from the guard for e.
We have `query` s $\mathsf{of}(n, s_1) : \mathsf{need}(s, \mathsf{of}(n, s_1)) \Rightarrow \mathsf{of}(n, s_1)$, and
`guard` e $\mathsf{with}(n, s_1) : \mathsf{used}(e) \Rightarrow \mathsf{used}(e) \cap \mathsf{with}(n, s_1)$.
So the input of the first branch is n, as n subsumes each of the other inputs. The output is v, as n subsumes each of the other outputs.
In the second branch, the type is $\mathsf{used}(e) \cup v \Rightarrow v$. The type of the sum is therefore $n \Rightarrow v$, since n subsumes $\mathsf{used}(e)$ and v. This is consistent with the type of the fixed point.

– Case s_1; s_2:
Let $v_2 = \mathsf{of}(v, s_2)$. We query s_2 for all of the variables in v it makes available, v_2. We must then query s_1 for the variables in v that were not made available by s_2 as well as for the available inputs to s_2. This is exactly the union of $v \setminus v_2$ and $\mathsf{of}(\mathsf{need}(s_2, v_2), s_1)$.
The input of the overall composition is therefore
$(\mathsf{need}(s_2, v_2) \setminus \mathsf{of}(\mathsf{need}(s_2, v_2), s_1)) \cup \mathsf{need}(s_1, v \setminus v_2)$, which is inputs to s_2 not available in s_1, and inputs to s_1 for variables not available in s_2, which overall is $\mathsf{need}(s, v)$. The output of the composition is simply v since all intermediate variables are cancelled, so the type of the overall transformer is $\mathsf{need}(s, v) \Rightarrow v$.