Language-Agnostic Static Analysis For Probabilistic Programs

Markus Böck
markus.h.boeck@tuwien.ac.at
TU Wien
Vienna, Austria

Michael Schröder
michael.schroeder@tuwien.ac.at
TU Wien
Vienna, Austria

Jürgen Cito
juergen.cito@tuwien.ac.at
TU Wien
Vienna, Austria

ABSTRACT

Probabilistic programming allows developers to focus on the modeling aspect in the Bayesian workflow by abstracting away the posterior inference machinery. In practice, however, programming errors specific to the probabilistic environment are hard to fix without deep knowledge of the underlying systems. Like in classical software engineering, static program analysis methods could be employed to catch many of these errors. In this work, we present the first framework to formulate static analyses for probabilistic programs in a language-agnostic manner: LASAPP. While prior work focused on specific languages, all analyses written with our framework can be readily applied to new languages by adding easy-to-implement API bindings. Our prototype supports five popular probabilistic programming languages out-of-the-box. We demonstrate the effectiveness and expressiveness of the LASAPP framework by presenting four language-agnostic probabilistic program analyses that address problems discussed in the literature.

CCS CONCEPTS

• **Mathematics of computing** → *Bayesian computation*; • **Theory of computation** → **Program analysis**; • **Software and its engineering** → Abstraction, modeling and modularity; Error handling and recovery.

KEYWORDS

probabilistic programming, program analysis, language-agnostic

ACM Reference Format:
Markus Böck, Michael Schröder, and Jürgen Cito. 2024. Language-Agnostic Static Analysis For Probabilistic Programs. In *Proceedings of ACM Conference (Conference'24).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In Bayesian inference, we have a set of latent variables, observed data, and a statistical model which typically encodes how the observations are believed to be generated from the latents. Domain knowledge about the latents can be incorporated by specifying informative prior distributions. Posterior inference is concerned with finding or approximating the posterior probability distribution of the model—the distribution over the latent variables given the observed data. Probabilistic programming systems provide an intuitive means to specify Bayesian models as simple programs and to automatically perform posterior inference on them. These systems decouple modeling from inference by implementing general-purpose inference algorithms. While previously the analysis of complex Bayesian models was reserved for experts, probabilistic programming opens it up for users without substantial knowledge of statistics and inference techniques.

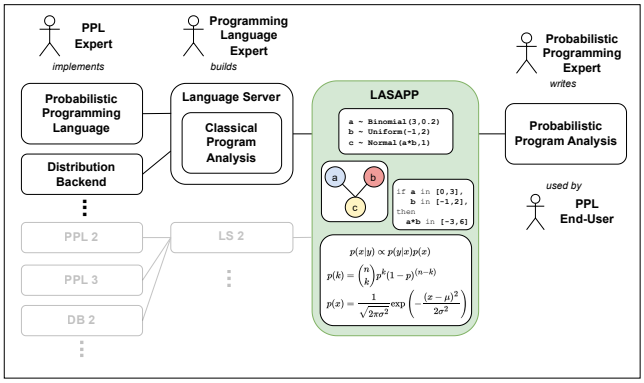


Figure 1: Overview of the LASAPP framework: Analyses are formulated in a high-level language-agnostic API. Internally, a language server performs classic analyses like data and control flow analysis. Servers have to be built only once per host language and facilitates easy-to-implement bindings for multiple embedded probabilistic programming languages.

While writing probabilistic programs seems intuitive, it also leads to many counter-intuitive situations. For instance, a programmer modeling a generative process may think about it sequentially, but the execution order of sample statements during inference need not be sequential and in general depends on the applied algorithm. Also, due to the stochastic nature of probabilistic programming, bugs may not occur during every run of the program and could be hard to reproduce. Furthermore, probabilistic programming systems abstract away thousands of lines of code and one may have to trace bugs deep into the inference libraries. For these reasons, debugging probabilistic programs still requires extensive knowledge of inference algorithms and the system at hand.

Program analysis can serve as a powerful technique to catch bugs statically and to improve the usability of probabilistic programming languages in general. In fact, recent research realizing this idea includes SlicStan, which relaxes the syntax of Stan [22], automatic guide generation for Pyro [33], improved debugging with probabilistic assertions [36], and slicing of probabilistic programs for improved inference [28]. However, a shared attribute of all of these

Conference'24, July 2024, Washington, DC, USA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

methods is their limitation to a single probabilistic programming language.

In this paper, we present LASAPP (Figure 1)—the first approach to writing static analyses for probabilistic programs in a language-agnostic way. The key insight lies in the fact that while probabilistic programming languages differ in their implementation and expressiveness, the underlying meaning of these programs remains the same: a probabilistic model. By using existing program analysis techniques, like data and control flow analysis, we can put an abstraction layer on top of classical program properties and formulate analyses of probabilistic properties in the language of these abstractions at a high level. Our approach makes it easy to add support for new languages and thus existing analyses written in LASAPP are immediately available for them. Furthermore, the clear separation of classical program analyses entails that one can write probabilistic analysis without having to be a programming languages expert. We demonstrate that our framework allows the formulation of many practical analyses that greatly improve the usability of a multitude of probabilistic programming languages at once.

The main contributions of this work are:

- The first language-agnostic static analysis framework for probabilistic programs (LASAPP).
- Four language-agnostic program analyses for improved usability of probabilistic programming systems:
  - Statistical Dependency Analysis (Section 4.1)
  - HMC Assumptions Checker (Section 4.2)
  - Parameter Constraint Verifier (Section 4.3)
  - Model-Guide Validation (Section 4.4)
- Prototype LASAPP bindings for the popular languages Pyro [8], PyMC [39], BeanMachine [42], Turing [20], and Gen [13], available at https://github.com/lasapp/lasapp.git.

2 MOTIVATION

2.1 Probabilistic Programming

Probabilistic programming is an intuitive means to specify Bayesian models as programs. We start by giving a general definition:

Definition. *A probabilistic program is a (non-deterministic) program, for which an (unnormalized) weight can be assigned to each possible execution.*

If the program returns a number, we can think of it as an (unnormalized) probability distribution over the return values. However, in probabilistic programming the local random variables of a program are also of great interest as they typically encode how observed data is believed to be generated. For each run of the program, the values of these variables are captured in an *execution trace*. Thus, probabilistic programs can also be viewed as an (unnormalized) joint probability distribution over all random variables, or as an (unnormalized) probability distribution over execution traces [2, 11, 21, 41, 44]. This distribution may also be referred to as *program density*.

In most probabilistic programming languages (PPLs), random variables of a model and their prior distributions are declared with special syntax in the form of *sample statements*. Relating these variables with program constructs implicitly defines the likelihood of the model. Furthermore, in probabilistic programming systems, there is a mechanism to constrain the values of random variables to observed data. This results in a probabilistic model *conditioned* on data. All variables that are not constrained to any observed values are called *latent variables*.

```
def linear_model(x, y):
  a = pyro.sample("a", dist.Normal(0,10))
  b = pyro.sample("b", dist.Normal(0,10))
  s2 = pyro.sample("s2", dist.InverseGamma(1,1))
  for i in range(len(x)):
    pyro.sample(f"y_{i}", dist.Normal(a*x[i]+b,s2), obs=y[i])
```

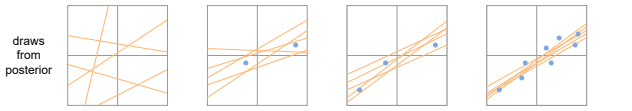


Figure 2: Linear model implemented in Pyro on the top. Results from posterior inference with decreasing uncertainty on the bottom.

In Figure 2, you can see a classic example of a probabilistic model, namely a linear model, where the univariate data y is believed to have a noisy linear relationship with the input values x. The variables y_i are constrained to observed data and we are interested in the posterior distribution over the latents `a`, `b`, and `s2`—the distribution over the slope, intercept, and noise parameter given the data. We can inspect the posterior distribution to find the line that best fits the data and quantify the uncertainty about our conclusions.

What makes probabilistic programming special is that across different languages the semantic meaning of the programs remains the same. The languages differ in syntax and implementations, making trade-offs between the efficiency of inference algorithms and generality in terms of which models are expressible. However, the programs written in any language can all be interpreted as mathematical probabilistic models. Thus, the same probabilistic program analysis is meaningful in several languages and it is desirable to be able to apply the analysis directly without having to build up an analysis framework for each language.

2.2 Static Analysis of Probabilistic Programs

Consider a simple Bayesian model where a system can be in two states—5 or 6. Noisy measurements X of the state are available and with posterior inference we wish to find the most probable state of the system given the measurements and quantify the uncertainty. In Figure 3, you can see implementations of this model in Turing, BeanMachine, and Pyro.

Even though Hamiltonian Monte Carlo (HMC) [25] is a very robust and frequently recommended inference algorithm, it would be ill-advised to apply it for the described model. In fact, for all of the three implementations, it leads to cryptic runtime errors which need to be traced deep into the inference library. The trained eye will immediately spot the declarations of the discrete random variable `state` and find it incompatible with the gradient-based inference algorithm. However, we argue that this is a non-trivial bug, because the error messages provide little help, it requires knowledge of the assumptions of the inference algorithm to fix, and the same source code would be correct for different algorithms.

```
@model function model(X)
  state ~ Categorical([0.5, 0.5])
  if state == 1        ⇒ ERROR: InexactError:
    mu = 5.              Int64(0.13392275765318448)
  else
    mu = 6.
  end
  for i in eachindex(X)
    X[i] ~ Normal(mu, 1.)
  end
end
```

```
@bm.random_variable
def state():
  return dist.Categorical(torch.tensor([0.5,0.5]))
@bm.random_variable                    ⇒ RuntimeError: only
def model(n):                            Tensors of a floating
  if state() == 1:                       point and complex dtype
    mu = 5.                              can require gradients
  else:
    mu = 6.
  return dist.Normal(mu * torch.ones(n), 1.)
```

```
def model(X):
  state = pyro.sample("state", dist.Categorical([0.5,0.5]))
  if state == 1:                          ⇒ RuntimeError: Boolean
    mu = 5.                                 value of a Tensor with
  else:                                     more than one value is
    mu = 6.                                 ambiguous.
  pyro.sample("X",
  dist.Normal(mu * torch.ones(len(X)), 1.), obs=X)
```

Figure 3: Simple model implemented in Turing, BeanMachine, and Pyro (top to bottom). Applying the HMC algorithm to these models results in different cryptic bugs. The root cause is identical for all of them: the declaration of the discrete variable `state` is not supported by HMC which requires continuous variables.

Having observed the differences in syntax and disregarding the implementation details of HMC, we note that the underlying root cause of the runtime error is the same for all implementations. Even more, the bug can be caught *statically*. However, implementing such a static analysis separately for each PPL would require a separate parser for each language's unique syntax, extracting and traversing different abstract syntax trees (ASTs) to find the respective sample statements, extracting the underlying distribution and looking up its properties based on the PPL's particular semantics, and finally raising a warning if the type is discrete.

In this paper, we propose a framework where we abstract away much of the classical program analysis details like traversing the AST and parsing sample statements. As a result, the same static analysis can to be written at a higher level in just three steps:

(1) Find all random variable declarations.
(2) Iterate over all random variables.
(3) If a variable has a discrete distribution, raise a warning.

In terms of the concrete LASAPP API, these high-level steps translate almost one-to-one to Python code:

```
1 import lasapp
2 program = lasapp.ProbabilisticProgram(path)
3 random_variables = program.get_random_variables()    # (1)
4
5 for rv in random_variables:                          # (2)
6   props = lasapp.infer_distribution_properties(rv)
7   if props.is_discrete():                            # (3)
8     raise Warning("...")
```

What makes our approach powerful is that adding support for new languages is easy and all existing program analysis written in the framework are immediately available. The above analysis is written in a completely language-agnostic way and can be applied to prevent all bugs from Figure 3. In fact, it also prevents two issues reported by developers in the Turing and Pyro forum, which where the inspiration for the discussed examples [19, 29].

3 LASAPP

Figure 1 presents an overview of LASAPP, our framework for **L**anguage-**A**gnostic **S**tatic **A**nalysis for **P**robabilistic **P**rogramming. LASAPP targets PPLs that are embedded in a host language, like Pyro (which is embedded in Python), or Turing (embedded in Julia). Focusing on these types of PPLs allows us to separate probabilistic analysis from the classical program analysis machinery. LASAPP makes it possible for PPL experts to write high-level probabilistic program analyses in a language-agnostic way.

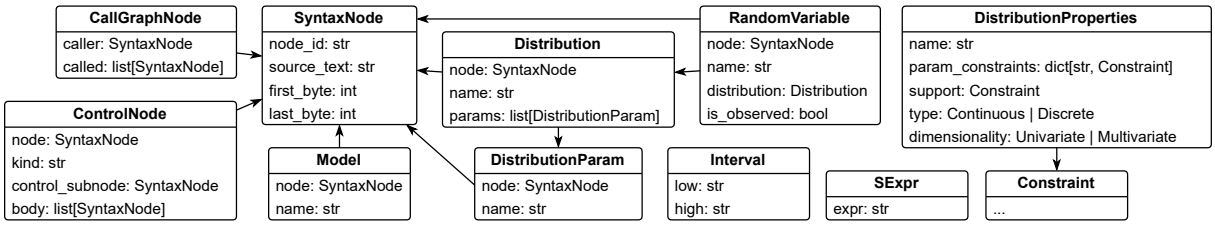
For each host language, a programming language expert implements a *language server*, which is responsible for parsing source code and representing it as an abstract syntax tree (AST), performing data- and control-flow analysis, abstract interpretation, or symbolic execution. Language servers need to be implemented only once per host language, and can be built using a plethora of well-established methods (Section 3.2). Once a host language is supported, adding support for new PPLs embedded in that language is simple. Mostly, one has to write bindings for a PPL's special sample syntax (Section 3.1.1) and for its probabilistic distribution back-end (Section 3.1.2). For our prototype, we implemented language servers for Python and Julia, and added support for five PPLs, including two shared probability distribution back-ends (Table 1).

Table 1: Lines of code needed to add LASAPP support for various PPLs and probability distribution back-ends.

	Back-end / PPL	LOC	
Python	Language Server	1131	
	PyMC [39]	153	custom back-end
	torch.distributions [37]	68	shared back-end
	Pyro [8]	63	
	BeanMachine [42]	134	
Julia	Language Server	1822	
	Distributions.jl [6]	110	shared back-end
	Gen [13]	192	
	Turing [20]	74	

3.1 Probabilistic Analysis API

LASAPP provides a Python API to implement static analyses of probabilistic programs. Figure 4 shows the main data types and most important API methods. We arrived at this API design by considering the common characteristics of popular PPLs to find a shared set of universal abstractions that would serve the needs of real-world probabilistic analyses (see Section 4).




```
class ProbabilisticProgram(filename: str)
  def get_model() → Model
  def get_random_variables() → list[RandomVariable]                                                    # see Section 3.1.1
  def get_data_dependencies(node: SyntaxNode) → list[SyntaxNode]                                       # see Section 3.2.1
  def get_control_parents(node: SyntaxNode) → list[ControlNode]                                        # see Section 3.2.1
  def get_call_graph(node: SyntaxNode) → list[CallGraphNode]                                           # see Section 3.2.1
  def estimate_value_range(expr: SyntaxNode, assumptions: dict[RandomVariable, Interval]) → Interval   # see Section 3.2.2
  def get_path_condition(node: SyntaxNode, root: SyntaxNode, assumptions: dict[SyntaxNode, SExpr]) → SExpr  # see Section 3.2.3
def infer_distribution_properties(rv : RandomVariable) → Optional[DistributionProperties]             # see Section 3.1.2
```

Figure 4: Main types and methods of the LASAPP API.

3.1.1 Sample Statements. Different PPLs use different syntax to declare random variables. In these sample statements, we require a user-defined random variable name and an expression corresponding to the distribution of the variable. The variable name is also referred to as *address*—a (dynamically) unique string, with which the value of a variable is stored in the execution trace.

We give a few examples of different sample syntaxes, where we highlight the addresses, distributions, and special syntax:

- Turing has a special syntax which includes a tilde character, which is resolved with a macro before compilation. Values are assigned to program variables denoted on the left-hand side, which also serve as addresses:

```
mu ~ Normal(0., 1.)
x[i] ~ Normal(mu, 1.)
```

- In BeanMachine, a special decorator is added to function definitions to declare a random variable, which can then be referenced from anywhere with a regular function call. The above example can be rewritten in BeanMachine as:

```
@bm.random_variable
def mu() :
  return dist.Normal(0., 1.)
@bm.random_variable
def x(i) :
  return dist.Normal(mu(), 1.)
```

- In Pyro, sample statements are calls to the `pyro.sample` method, with addresses and distributions passed as arguments. In Pyro the example becomes:

```
mu = pyro.sample( "mu" , dist.Normal(0., 1.) )
x[i] = pyro.sample( f"x_{i}" , dist.Normal(mu, 1.) )
```

LASAPP abstracts all of these into the same `RandomVariable` data type, preserving the semantics of the original syntaxes.

In this work, we only consider languages where sample statements can be extracted statically from the source code. To be able to perform classical static analyses, like data flow analysis, we also require that random variables are always bound to program identifiers, e.g., variables or functions.

In addition to its distribution, it is also interesting to know whether a random variable is observed or not. Mechanism to declare an observed variable range from adding an optional "observed" parameter to the sample call, passing the observed value as a function parameter, or collecting the observations in a hash-map. For the sake of brevity we refer to the prototype implementation and PPL documentations for more details.

Lastly, probabilistic programs usually have a model entry-point or a program construct which encapsulates the sample statements. For instance, in Pyro and Turing a program function defines a model while in PyMC sample statements are listed in the body of a `with` statement.

3.1.2 Distribution Back-ends. Probabilistic programming systems either have their own implementation of common probability distributions or use a package of their host language. For instance, PyMC has its own distribution library, while Pyro and BeanMachine share the PyTorch `torch.distributions` back-end [37] and Turing and Gen share the `Distributions.jl` back-end [6]. Sharing back-ends between PPLs allows for further modularisation and reusability but introduces an additional layer of variation.

LASAPP needs to map back-end-specific distribution and parameter names to a common vocabulary. For instance, consider the `Gamma` object in the `Distributions.jl` package, where the default parameters map to a parametrization in terms of shape and rate:

$$\texttt{Gamma}(\overset{\alpha}{1.5}, \overset{\beta}{2.0}) \quad \rightsquigarrow \quad p(x) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} \exp(-\beta x)$$

In PyTorch, however, the `Gamma` distribution is expressed in terms of shape and scale:

$$\texttt{Gamma}(\overset{k}{1.5},\ \overset{\theta}{2.0}) \quad \rightsquigarrow \quad p(x) = \frac{1}{\Gamma(k)\theta^k} x^{k-1} \exp\left(-\frac{x}{\theta}\right)$$

LASAPP represents distributions with unique names for both parameters and the distribution itself.

Note that a distribution expression in a sample statement need not be a single distribution object, but could be an arbitrary expression evaluating to some distribution at runtime. LASAPP currently maps such expressions to an "Unknown" distribution type. In future work, we plan on parsing these arbitrary expression to support more distribution classes, such as mixture distributions.

3.2 Classical Analysis Foundation

LASAPP's probabilistic analysis API is built on a foundation of classical static analysis methods. This layered approach allows PPL experts to focus on implementing high-level language-agnostic analyses of probabilistic properties, while programming language experts can provide the necessary classical analysis foundations for each host language.

While all the classical techniques we discuss in this section are of general importance to PPL analysis, they were specifically chosen to aid in the implementation of the probabilistic analyses we present in Section 4. Other probabilistic analyses may require the support of additional classical techniques. By virtue of its modular design, LASAPP can be extended to incorporate additional static analysis methods in its language servers and expose them via the high-level API, accommodating the needs of new probabilistic analyses as they arise.

3.2.1 Data/Control Flow Analysis. Data and control flow analysis is concerned with finding the relationship between program variables and the order in which program statements are executed. We begin by informally defining data dependencies following Hennessy and Patterson [24].

Definition (Data Dependency). *A statement S_1 is* data dependent *on statement S_2, if S_2 produces a result that may be used by S_1.*

This definition is related to the notion of *reaching definitions* [9]. Note that if statement S_2 is data dependent on statement S_3, then S_1 is also data dependent on S_3 by transitive closure.

Next, we give a general definition of a control-flow graph [9]. For a more formal definition, see for instance Cytron et al. [14].

Definition (Control-Flow Graph). *A* control-flow graph *is a directed graph that connects basic blocks of a program (sequence of operations always executed together) and encodes all paths that might be traversed through a program during its execution.*

Lastly, we consider a special type of control-flow graph, namely the call graph [9].

Definition (Call Graph). *A* call graph *is a directed graph that represents the calling relationships among the procedures in a program.*

Data dependencies, control flow, and the call graph are particularly important in the analysis of probabilistic programs, because they are used to find dependencies between sample statements. In fact, knowing the dependencies of random variables allows us to represent programs as *Bayesian networks* or more generally as *probabilistic graphical models*. Graphical models are well-understood mathematical objects [31] which allows us to leverage existing methods to reason about probabilistic programs. In Section 4.1, we will show how to statically extract the model graph of a probabilistic programs following the data and control flow.

3.2.2 Abstract Interpretation. Abstract interpretation is a technique for approximating the behaviour of a program by replacing its concrete semantics with an *abstract* semantics [10]. An abstract domain is usually based on (partially) ordered sets or lattices. As an illustrative example, consider an analysis to determine whether a numerical variable is positive or negative. We can replace the concrete variables with an abstract representation of just their signs, i.e., whether the variable is positive (+), negative (−), or zero (0). To achieve soundness, the concrete semantics are over-approximated: we may need to accept that we cannot always tell the sign of a variable and have to assign it an abstract "don't know" value ($\top$).

For our purposes, we are interested in more than just the signs of program variables. As probabilistic programs are stochastic by nature and the values of random variables may be different from one execution to another, we would rather like to know the possible *range of values* of any particular variable. To achieve this, we have implemented an interval abstract domain in LASAPP to approximate value ranges with intervals. Intervals are propagated through the program by following interval arithmetic [1, 35]. See some of the rules below.

$$\begin{aligned}
[a,b] + [c,d] &= [a+c, b+d] \\
[a,b] - [c,d] &= [a-d, b-c] \\
[a,b] \times [c,d] &= [\min(ac,ad,bc,bd), \max((ac,ad,bc,bd)] \\
\exp([a,b]) &= [\exp(a), \exp(b)]
\end{aligned}$$

In Section 4.3, we analyse the value range of distribution parameters to verify that they do not violate the imposed constraints in any possible program execution.

3.2.3 Symbolic execution. Symbolic execution is similar to abstract interpretation, in the sense that here too the concrete semantics of a program are abstracted in order to prove certain properties. Program variables or inputs are exchanged with symbols representing arbitrary values. The syntax of the programming language remains the same, but the execution now follows the symbolic semantics [30]. In contrast to abstract interpretation, symbolic execution does not necessarily explore all possible program paths, but it has the advantage that it can often avoid over-approximation.

A particular application of symbolic execution relevant for this work is concerned with so-called *path conditions*. Depending on the values of variables appearing in conditional if-statements, either the "then" or the "else" branch will be taken during execution. By representing the conditional symbolically, we can derive constraints that need to be satisfied for a certain program branch to be executed. This is especially interesting for probabilistic programs that exhibit stochastic branching. For these types of programs, random variables may appear in conditional statements where a certain fraction p of executions are expected to take the "then" branch, while the

"else" branch will be taken with probability $1 - p$. Such properties are especially relevant for validating so-called guide programs, a practical analysis we describe in Section 4.4.

3.3 Implementation

For our prototype, we have implemented a Julia and a Python server by building on the default parsers and predominantly following a visitor pattern for the analysis of a program's abstract syntax tree. The bindings for a probabilistic programming language effectively have to match abstract syntax nodes to the corresponding sample statement syntax (see Section 3.1.1). This includes parsing the distributions and its parameters when possible (see Section 3.1.2) and checking whether the corresponding random variable is observed.

From Table 1, we can see that the initial implementation of the language servers in our prototype requires an order of magnitude more lines of code than the PPL bindings. This demonstrates that the heavy lifting of writing the classical program analysis backbone of LASAPP has to be done only once for each host language and support for new probabilistic programming languages can be added easily. Note that we have implemented basic versions of the classic static analyses described in Section 3.2 (e.g. no loops for abstract interpretation or symbolic execution) from scratch to make the LASAPP prototype self-contained. In future work, they can be replaced with more sophisticated implementations.

The LASAPP front-end Python package for writing probabilistic static analyses communicates with the language servers via JSON RPC, a simple remote procedure protocol implemented in many programming languages. In Figure 4, you can see the most important methods of this API. Note that only get_model and get_random_variables interface with the PPL bindings. All other methods can be viewed as general purpose static analysis endpoints and can be developed independently from all probabilistic aspects of the framework by a programming languages expert.

Our prototype implementation of the LASAPP framework can be found at https://github.com/lasapp/lasapp.git.

4 EVALUATION

We evaluate our approach by presenting four language-agnostic analyses formulated in our framework. These analyses are partly inspired by existing research. We show that our analyses cover real-world use cases and work well on examples examined in prior publications. For the sake of brevity, we will only give an overview description of the analyses and simplified versions of the implementations in LASAPP's API. The full implementations can be found in the replication package.

4.1 Statistical Dependency Analysis

Finding dependencies between sample statements is a building block of many analyses. In fact, representing a probabilistic program graphically as a Bayesian network allows us to apply algorithms designed specifically for probabilistic graphical models for further analysis [31]. The high-level overview of the algorithm to find these dependencies is as follows:

(1) Initialise an empty statistical dependency graph $\mathcal{G}$.
(2) Find all sample statements.
(3) Filter for random variables that are reachable from the model entry point.
(4) For each random variable, traverse the data dependency graph until we encounter another sample statement. For each data dependency d, we also need to traverse the data dependencies of every control parent of d.
(5) Add an edge between the two sample statements to $\mathcal{G}$.

You can see a concrete LASAPP implementation of the steps above in Listing 1 (simplified and shortened for clarity).

```
dependencies = [] # == G                                  # (1)
all_variables = program.get_random_variables()            # (2)
model = program.get_model()
cg = program.get_call_graph(model.node)
random_variables = filter(all_variables, model, cg)       # (3)
for rv in random_variables:                               # (4)
  queue = [rv.node]
  while len(queue) > 0:
    node = queue.popleft()
    data_deps = program.get_data_dependencies(node)       # (4)
    for dep in data_deps:
      if not already_processed(dep):
        if dep in random_variables:
          dependencies.append((dep, rv))                  # (5)
        else:
          queue.append(dep)
      mark_processed(dep)

  for dep in program.get_control_parents(node):           # (4)
    queue.append(dep.control_subnode)
```

Listing 1: Model graph extraction written in LASAPP.

In Figure 5, we illustrate how the algorithm operates and why we also need to traverse the data dependencies of control nodes. To find the statistical dependencies of the random variable `x`, we start by finding the data dependencies of the distribution parameter `m`. We notice that the value of `m` depends on the conditional `b == 1` and thus, also `x` implicitly depends on the conditional. We continue to find the only data dependency of `b == 1` in form of the sample statement of `b`, stop the traversal and add the edge `b` $\rightarrow$ `x` to the statistical dependency graph.

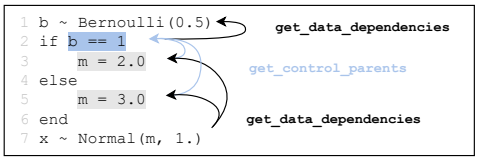


Figure 5: Repeated application of `get_data_dependencies` and `get_control_dependencies` to find statistical dependencies in a Turing program.

We note that the algorithm above over-approximates the amount of dependencies. To see this, consider again the program in Figure 5, but replace line 5 with `m = 2.0`. This modification does not change the behaviour and result of the algorithm, however, it is clear that now `m` does not really depend on `b` as it takes the same value in both branches. This makes `m` and `b` statistically independent which is not captured by the algorithm.

There are also programs where it is not that obvious that two variables are independent even though they are connected by program constructs. In fact, it is known that in general, the graphical representation of a probabilistic model (which we approximate for a program by applying the algorithm above) does not necessarily capture all independence properties [31]. In contrast to the dependencies, the independencies detected graphically are guaranteed to be true independencies in the model.

```
@model function slicing(g)
  d ~ Bernoulli(0.6)
  i ~ Bernoulli(0.7)

  if (!i && !d)
    g_prob = 0.3
  else
    g_prob = (i && !d) ? 0.9 : 0.5
  end
  g ~ Bernoulli(g_prob)
  s ~ Bernoulli(i ? 0.95 : 0.2)
  l ~ Bernoulli(g ? 0.5 : 0.1)
end
```

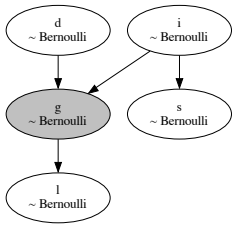


Figure 6: Statistical dependency graph of a Turing program extracted with LASAPP.

Fortunately, the graphical representation is a so-called perfect independence map for many real world models. Consider the program in Figure 6, which was adapted from Hur et al. [28]. The graphical representation on the right immediately reveals the correct variable relationships and aids in program comprehension. For example, the shaded node g represents an observed variable which implies that the parent nodes d and i are statistically dependent in the posterior distribution. This fact may be counter-intuitive for novice PPL users as the corresponding sample statements are seemingly independent in the program on the left.

Furthermore, if the graphical representation extracted via program analysis is a perfect independence map, then one can apply algorithms designed for probabilistic graphical models based on d-separation to automatically determine whether two variables are dependent in the posterior [31]. For instance, if we are mainly interested in the posterior distribution of s, we could automatically infer that this variable is independent from l and remove the corresponding sample statement from the model. For complicated models, this may speed-up inference algorithms significantly. In fact, Hur et al. developed a program slicing method to achieve the same goal. While in contrast to their approach, our analysis comes with no correctness guarantees (we may not find all independencies), it is useful for many programs and can be applied to a multitude of PPLs.

4.2 HMC Assumptions Checker

We turn back to our motivating example, see Figure 3. Recall that all runtime errors caused by applying the Hamiltonian Monte Carlo algorithm can be prevented statically by disallowing discrete variables in the model. We already presented the corresponding LASAPP analysis in Section 2 and include it again in Listing 2. However, the assumption of continuous variables is not the only one for HMC and we continue to implement more checks.

As a gradient-based method, HMC requires the program density to be differentiable with respect to all random variables. Stochastic control flow often leads to discontinuous densities and can be statically avoided, if we check whether a random variable is control dependent on another variable. This can be done with an algorithm similar to the more general dependency analysis of Section 4.1. We abbreviate this subroutine with `has_random_control_deps`.

Another important assumption of HMC is that the set of random variables is the same in each possible execution of the program. Two recent publications [7, 34] used a random walk program to demonstrate that Pyro's HMC implementation fails to recover the correct posterior. The program starts with a number on the real line and randomly decreases or increases the number. Once the number surpasses a threshold the program returns, see Figure 7. As program termination depends on the random values drawn, a different number of steps and thus random variable instantiations may be needed to surpass the threshold. Inference fails because Pyro's standard implementation is not designed to handle such models.

```
model = program.get_model()
variables = program.get_random_variables()
for rv in random_variables:
  props = lasapp.infer_distribution_properties(rv)
  if props.is_discrete():
    raise Warning("Discrete Variables ...")

  if has_random_control_deps(program, rv):
    raise Warning("Stochastic Control Flow ...")

  control_parents = program.get_control_parents(rv.node)
  for parent in control_parents:
    if parent.kind == "for":
      if has_random_control_deps(program, parent):
        raise Warning("Stochastic For Loop Range ...")
    if parent.kind == "while":
      raise Warning("Sample In While Loop ...")

call_graph = program.get_call_graph(model.node)
# check for sample statements in recursive calls ...
```

Listing 2: HMC assumption checker written in LASAPP.

We extend our program analysis to check for sample statements appearing in while-loop bodies to prevent an unbounded number of random variables, see Listing 2. For the same reason, we give a warning if a for-loop range depends on a random variable. Further, if a sample statement appears in an if branch where the conditional is stochastic, we check that a random variable with the same address is defined in all other branches (omitted from Listing 2 for brevity, see the full implementation in the replication package). Lastly, to detect if sample statements appear in recursive calls which may also lead to an unbounded number of random variabels, we inspect the call graph (also omitted).

Applying the analysis to the aforementioned random walk Pyro program results in a warning at the correct source code location that explains why the HMC algorithm is not a suitable choice, see Figure 7.

4.3 Parameter Constraint Verifier

One of the bigger challenges when starting with probabilistic programming is that probabilistic programs are typically not executed

```
def pedestrian():
  start = pyro.sample("start", dist.Uniform(0, 3))
  t = 0
  position = start
  distance = torch.tensor(0.0)
  while position > 0 and position < 10:
    step = pyro.sample(f"step_{t}", dist.Uniform(-1, 1))
WARNING: Random variable appears in a while loop body. This may lead to an unbounded
number of random variables which is not supported by HMC.
    distance = distance + step.abs()
    position = position + step
    t = t + 1
  pyro.sample("obs", dist.Normal(1.1, 0.1), obs=distance)
  return start
```

Figure 7: Pedestrian model for which Pyro's HMC fails [34]. The LASAPP analysis (Listing 2) provides the error message.

sequentially and values of variables are random. This may lead to runtime errors that occur with low probability and are hard to reproduce. Therefore, it is often beneficial to think about the entire *range of values* a random variable can take to verify that a program runs without errors for every possible execution trace.

This line of thought motivated the use of abstract interpretation in form of interval analysis as described in Section 3.2.2 to verify that the parameters passed to distributions satisfy constraints. For example, the standard deviation parameter of a Normal distribution has to be strictly positive. This check, among others, is also included in Stan's pedantic mode implemented directly in the compiler [4]. In contrast, the analysis presented in the following is written in the LASAPP framework and can be applied to multiple PPLs at once.

We give a brief overview of the probabilistic analysis to find constraint violations in distribution arguments:

(1) Initialise an empty "assumptions map" $\mathcal{A}$.
(2) Find all sample statements.
(3) For each random variable R, retrieve the value range V and add $R \mapsto V$ to $\mathcal{A}$.
(4) For each random variable, retrieve its parameter constraints.
(5) Check for violations by comparing against the estimated range for the parameter node in $\mathcal{A}$.

The LASAPP implementation can be seen in Listing 3.

```
assumptions = {}                                          # (1)
variables = program.get_random_variables()                # (2)
for rv in random_variables:
  props = lasapp.infer_distribution_properties(rv)
  assumptions[rv] = Interval(props.support)               # (3)

for rv in random_variables:
  props = lasapp.infer_distribution_properties(rv)
  for param in rv.distribution.params:
    constraint = props.param_constraints[param]           # (4)
    estimated_range = program.estimate_value_range(       # (5)
      expr=param.node,
      assumptions=assumptions
    )
   if not is_subset(estimated_range, constraint):         # (5)
     raise Warning("Constraint violation ...")
```

Listing 3: Parameter constraint verifier written in LASAPP.

The analysis is best explained by considering the example programs in Figure 8. The Normal distribution spans the entire real number line and thus in the left program, the variables `slope`, `intercept`, and `sigma` are masked with $[-\infty, \infty]$. However, this violates the parameter constraint of `sigma` as explained before.

In the right program, the value range of the variable `prob` is more difficult to estimate. Again, `z` is replaced with $[-\infty, \infty]$ and `u` is replaced with $[0, 1]$. Then, these intervals are transformed according to the mathematical expressions by applying interval arithmetic rules. Lastly, since either of the two branches could be taken, we compute the union of the estimated ranges for `prob` to be $[0, 1.5]$ which also violates the constraint for the "success probability" of the Geometric distribution, which has to be less than 1. Note that the right program would result in an error with a probability of less than 0.1%.

```
import pymc as pm
with pm.Model() as linear_model:
  slope = pm.Normal("a")
  intercept = pm.Normal("b")
  sigma = pm.Normal("s")

  pm.Normal("y",
    mu=slope*x+intercept,
    sigma=sigma, observed=y,
  )
WARNING: Parameter of Normal distribution
has constraint [0.0, inf], but values are
estimated to be in [-inf, inf].
```

```
@gen function model()
  b ~ bernoulli(0.999) # [0,1]
  if b
    z ~ normal(0.,1.)    # Real
    prob = 1/(1+exp(z))# [0,1]
  else
    u ~ beta(1,1)        # [0,1]
    prob = 1.5 * u    # [0,1.5]
  end
  x ~ geometric(prob)# [0,1.5]
WARNING: Parameter p of Geometric
distribution has constraint [0, 1],
but values are estimated to be in [0.0,
1.5].
end
```

Figure 8: PyMC (left) and Gen program (right) with parameter constraint violations.

4.4 Model-Guide Validation

Many posterior inference algorithms benefit from making random value draws not from the distributions defined in the model, but from so-called proposal distributions. Consider the linear model in Figure 2. In the model, we define broad *uninformative priors* for the slope and intercept parameters as to not bias the model towards specific values for these parameters. However, in inference, it makes sense to sample values, for instance, in the neighbourhood of the least squares solution to the line fitting problem, because we know that the posterior distribution will have low probability outside of this neighbourhood.

The theoretical justification of this method is the importance sampling equation (1), where we propose values from distribution q to produce unbiased samples from p [43].

$$\begin{aligned}\mathbb{E}_{X\sim p}[f(X)] &= \int f(x)p(x)dx \\ &= \int f(x)\frac{p(x)}{q(x)}q(x)dx = \mathbb{E}_{X\sim q}[f(X)w(X)]\end{aligned} \tag{1}$$

However, for the weights $w(x) = \frac{p(x)}{q(x)}$ to be well defined, we require that p is *absolute continuous* with respect to q:

$$p(x) > 0 \implies q(x) > 0. \tag{2}$$

In probabilistic programming, proposals come in form of so-called *guide programs* [12, 33], which also need to satisfy (2). We formulate a static analysis in the LASAPP framework, which automatically verifies that a model and guide program satisfy the

absolute continuity property. This analysis is described briefly in five steps, where we assume that all random variables are univariate and that all addresses are static.

(1) Find all sample statements for the model and guide program and group them by address.
(2) Get the path condition pc(stmt) for each sample statement in terms of function parameters and random variables.
(3) Express the support interval $[a, b]$ of sample statements with address X symbolically in form of distribution constraints $\texttt{dc(stmt)} = a \leq X \wedge X \leq b$.
(4) For each address X of program p, combine all sample statement constraints:

$$\texttt{constraints}(p, X) = \bigvee_{\substack{\texttt{stmt} \in \text{sample-} \\ \text{stmts of } X \text{ in } p}} \texttt{pc(stmt)} \wedge \texttt{dc(stmt)}$$

(5) For each address X, use an SMT-solver to prove that the following predicate is unsatisfiable:

$$\neg[\texttt{constraints(model}, X) \implies \texttt{constraints(guide}, X)]$$

Again, we show a simplified version of the LASAPP implementation in Listing 4.

```
def get_dist_constraint(rv):
  props = lasapp.infer_distribution_properties(rv)
  a, b = Interval(props.support)
  X = SExpr(rv)
  return And(a < X, X < b)

rvs = program.get_random_variables()                    # (1)
model = program.get_model(); guide = program.get_guide()
model_rvs = ...;             guide_rvs = ...;
model_rvs_by_name = ...;     guide_rvs_by_name = ...;

A = {rv.node: SExpr(rv) for rv in rvs} # assumptions
pc = {                                                  # (2)
  **{rv: program.get_path_condition(
rv.node, model.node, A) for rv in model_rvs},
  **{rv: program.get_path_condition(
rv.node, guide.node, A) for rv in guide_rvs}
}

dc = {rv: get_dist_constraint(rv) for rv in rvs}        # (3)

for name, model_stmts in model_rvs_by_name.items():
  guide_stmts = guide_rvs_by_name[name]
  impl = Implies(
    Or([And(pc[stmt],dc[stmt]) for stmt in model_stmts]),
    Or([And(pc[stmt],dc[stmt]) for stmt in guide_stmts])
  )                                                     # (4)
  if check(Not(impl)) == satisfiable:                   # (5)
    raise Warning("Absolute continuity violation ...")
```

Listing 4: Model-guide validation written in LASAPP.

To illustrate the analysis, consider the model and guide program in Figure 9, where the number of samples per execution is variable. In the model program, the sample statement for B in line 6 has path condition $(A = 1)$ with no further constraints, and the sample statement for B on line 8 has path condition $(A \neq 1)$ with values coming from a Gamma distribution with positive support $(B \geq 0)$. Similarly, in the guide program there is only one sample statement for B on line 8 with a Gamma distribution. With the equations in steps (4) and (5) we arrive at the following formula:

$$\neg\left[(A = 1) \vee ((A \neq 1) \wedge (B \geq 0)) \implies \texttt{true} \wedge (B \geq 0)\right].$$

On the left hand side are exactly the conditions for a execution trace to have a positive probability of being generated by the model program, $p(x) > 0$. On the right hand side are the analogous conditions for the guide program, $q(x) > 0$. Therefore, the implications corresponds to the absolute continuity property (2). For the given example, a SMT-solver (e.g. Z3 [15]) can readily find the counterexample $(A = 1) \wedge (B = -1)$ proving the guide program to be incompatible with the model. For all other addresses except E, formula (5) is unsatisfiable, thus proving the implication.

```
from pyro import sample                  from pyro import sample
def model(I: bool):                      def guide(I: bool):
  A = sample('A',Bernoulli(0.5))           if I:
  if A == 1:                                 A = sample('A',Bernoulli(0.9))
    B = sample('B',Normal(0.,1.))          else:
  else:                                      A = sample('A',Bernoulli(0.1))
    B = sample('B',Gamma(1,1))             B = sample('B',Gamma(1,1))

  if B > 1 and I:                          if B > 1 and I:
    sample('C',Beta(1,1))                    sample('C',Uniform(0,1))
  if B < 1 and I:                          else:
    sample('D',Normal(0.,1.))                sample('D',Normal(0.,1.))
  if B < 2:                                  sample('E',Normal(0.,1.))
    sample('E',Normal(0.,1.))
```

Figure 9: Model and guide program in Pyro which are not compatible in terms of the absolute continuity property.

Note that there are also other desirable properties of guides. Li et al. [33] considers *faithfulness* and *parsimony*. The former states that a guide can encode all conditional dependencies of the model and the latter states that it does not encode more dependencies than necessary. We could also implement an analysis in LASAPP to verify these properties by comparing the graphical representations of the guide and model program, see Section 4.1.

5 RELATED WORK

5.1 Language-Agnostic Program Analysis

Research on static program analysis is plentiful [16, 23]. Therefore, in this section, we focus on program analysis methods that aim to support language-agnostic analyses to some extent.

In 2015, Google presented the Tricorder framework [38]. Similar to the LASAPP language servers, the Tricorder ecosystem consists of analyzer worker services written in different languages that all implement a common language-agnostic API. Analyzers may be written in any language and may analyze any language. The analysis driver calls out to the language- or compiler-specific workers to run the program analyses.

The Cobra tool by Hoffmann [27] achieves language-agnosticism by interpreting programs simply as a list of tokens. These tokens are annotated with categories (identifier, keyword, type, etc.) and with additional context information. A special query language was developed to formulate program analyses on these lists of tokens.

The domain-specific language and infrastructure Boa [17, 18] was developed for easier and reproducible analyses of software repositories. Boa implements a representation of source code as abstract syntax trees with generic nodes like declaration, type, method,

etc. It provides language features inspired by visitor patterns to allow users to query and analyse the ASTs of large-scale projects.

The methods listed above focus on program analyses for general program properties that are applicable to many languages and are not powerful enough to implement the analyses presented in this work. In contrast, our framework is designed specifically for the analysis of probabilistic programs written in different host-languages.

Tangentially related, there are efforts to create language-agnostic representations of source code for machine learning models. For instance, Zügner et al. [46] presented a multi-lingual code summarization model.

Lastly, we like to mention the compilation framework LLVM [32]. LLVM is designed around a language-independent intermediate representation to facilitate program analyses. However, this representation is too low-level for the analyses considered in this work.

5.2 Static Analysis For Probabilistic Programs

The field of static analysis for probabilistic programming is young and largely under-explored. In this section, we will highlight prior work related to the probabilistic program analyses presented in Section 4 and refer to Bernstein's survey [3] for more methods and details.

In 2014, Hur et. al. [28] presented a way to reduce probabilistic programs to only those statements that are relevant to estimate the distribution of the return expression. This program slicing approach is a semantics-preserving transformation which can speed up posterior inference.

In 2014, Sampson et al. [40] introduced probabilistic assertion statements and an evaluation approach to either statically or dynamically verify them. In 2018, Hoffman et al. [26] presented a method to automatically analyze the source code of density functions for conjugacy structure and leveraged the structure for improved inference.

Stan has received the most attention with regards to static analysis. In 2018, SlicStan [22], a compositional version of the probabilistic programming language, was published. By relaxing the strict block syntax of Stan, it makes the code more flexible, reusable, and beginner friendly. Also, a semantics-preserving translation to Stan was developed. In the following years, Stan was also extended with a pedantic compilation mode [4]. Similar to the analyses in Section 4.2 and 4.3, warnings will be raised if there are any distribution usage issues or stochastic control flow. Lastly, a static transformation method relying on the extraction of the factor graph was developed to make draws from the prior and posterior predictive distribution of Stan programs [5].

Most recently Wang et al. [45] developed a type system to enforce absolute continuity of Pyro guide programs and Li et al. [33] presented an automatic approach to generate sound guide programs for a given Pyro program.

While the analyses of Section 4 have some overlap with the methods above, the key difference is that through the LASAPP framework our analyses are not restricted to one particular probabilistic programming language, but are written in a language-agnostic way and can be applied to any PPL which implements the API bindings.

6 THREATS TO VALIDITY

External validity. One potential limiting factor of our approach is that the framework and its abstractions may be only suitable for the selected set of probabilistic programming languages and the implemented program analyses. However, we argue that LASAPP generalises to new PPLs and analyses. The fact that the framework supports many different kinds of sample statements (function calls, function definitions, special syntax) shows its flexibility and adaptability to new languages. Also, the classic static analyses that are the backbone of the API are versatile. In particular, data and control flow was important for almost every program analysis presented in Section 4. Furthermore, the modularity of the LASAPP framework allows for the extension with additional classic static analysis methods if needed.

Internal validity. We have demonstrated the effectiveness of the LASAPP framework by presenting four language-agnostic program analyses. At this point, we were only able to evaluate the analyses on selected programs and focused on showing the versatility of the framework. Due to the complexity of the general purpose PPLs considered in this work, it was infeasible to reason about the correctness of the algorithms. Future work will include the formalisation of parts of the analyses and theoretical guarantees will be established for simpler and more restricted PPLs. Furthermore, it is difficult to evaluate the four presented analyses by means of quantitatively studying them on open-source real-world programs. This is due to the fact that the program analyses target bugs that occur during development and usually are fixed before the source code is manifested as part of public software repositories.

7 CONCLUSION

In this work, we presented the first language-agnostic static analysis framework for probabilistic programs LASAPP. We have demonstrate the effectiveness and expressiveness of the LASAPP framework by presenting four program analyses that address problems discussed in literature and applied them to five state-of-the-art probabilistic program languages.

To conclude, we describe the key characteristics of our language-agnostic static analysis framework for probabilistic programming by reflecting on the insights gained from designing LASAPP. First, probabilistic programming languages are special in the sense that their programs can be universally interpreted as probabilistic models. They only differ in the underlying posterior inference machinery and expressiveness. This justifies the goal of a language-agnostic framework as the same analysis for some probabilistic program properties is meaningful for many PPLs.

It is clear that probabilistic program analyses rely on classical static analysis methods. By designing LASAPP in a modular fashion, we made an effort to clearly separate the two types of analysis. This allows programming languages experts to build the language servers and implement the needed classical static analysis techniques like data and control flow analysis. The probabilistic programming systems experts can then provide the bindings for their PPL, where we have identified sample statements as the core language construct. The complexity of these implementations is then abstracted away by LASAPP's API. We argue that the selection of LASAPP's API methods forms a set of essential abstractions.

Finding dependencies of random variables via data and control flow is important for almost all analyses. Further, interval arithmetic and symbolic execution lends itself to handle the stochastic nature of probabilistic programs.

We have demonstrated the versatility of these abstraction by implementing four analyses tackling common problems in probabilistic programming including checking the assumptions of the HMC inference algorithm and verifying the compatibility of guide programs. In future work, we aim to formalise LASAPP and prove correctness of these analyses for a more restricted PPL as opposed to the general-purpose languages considered in this paper.

REFERENCES

[1] Götz Alefeld and Günter Mayer. 2000. Interval analysis: theory and applications. *Journal of computational and applied mathematics* 121, 1-2 (2000), 421–464.

[2] Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva. 2020. *Foundations of Probabilistic Programming*. Cambridge University Press.

[3] Ryan Bernstein. 2019. Static analysis for probabilistic programs. *arXiv preprint arXiv:1909.05076* (2019).

[4] Ryan Bernstein. 2023. *Abstractions for Probabilistic Programming to Support Model Development*. Ph. D. Dissertation. Columbia University.

[5] Ryan Bernstein, Matthijs Vákár, and Jeannette Wing. 2020. Transforming probabilistic programs for model checking. In *Proceedings of the 2020 ACM-IMS on Foundations of Data Science Conference*. 149–159.

[6] Mathieu Besançon, Theodore Papamarkou, David Anthoff, Alex Arslan, Simon Byrne, Dahua Lin, and John Pearson. 2021. Distributions.jl: Definition and Modeling of Probability Distributions in the JuliaStats Ecosystem. *Journal of Statistical Software* 98, 16 (2021), 1–30. https://doi.org/10.18637/jss.v098.i16

[7] Raven Beutner, C-H Luke Ong, and Fabian Zaiser. 2022. Guaranteed bounds for posterior inference in universal probabilistic programming. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 536–551.

[8] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. 2019. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research* 20, 1 (2019), 973–978.

[9] Keith D Cooper and Linda Torczon. 2011. *Engineering a compiler*. Elsevier.

[10] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 238–252.

[11] Marco Cusumano-Towner, Alexander K Lew, and Vikash K Mansinghka. 2020. Automating involutive mcmc using probabilistic and differentiable programming. *arXiv preprint arXiv:2007.09871* (2020).

[12] Marco F Cusumano-Towner and Vikash K Mansinghka. 2018. Using probabilistic programs as proposals. *arXiv preprint arXiv:1801.03612* (2018).

[13] Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. 2019. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th acm sigplan conference on programming language design and implementation*. 221–236.

[14] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.

[15] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer, 337–340.

[16] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. 2008. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 7 (2008), 1165–1178.

[17] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 422–431.

[18] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. 2015. Boa: Ultra-large-scale software repository and source-code mining. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 1 (2015), 1–34.

[19] EvoArt. [n. d.]. HMC issue in the Julia Turing Forum. https://discourse.julialang.org/t/turing-inexacterror-for-discreteuniform-distribution-with-nuts-sampler/52820. Accessed: 2023-07-24.

[20] Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: a language for flexible probabilistic inference. In *International conference on artificial intelligence and statistics*. PMLR, 1682–1690.

[21] Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. 2014. Probabilistic programming. In *Future of Software Engineering Proceedings*. 167–181.

[22] Maria I Gorinova, Andrew D Gordon, and Charles Sutton. 2018. SlicStan: Improving Probabilistic Programming using Information Flow Analysis. In *Workshop on Probabilistic Programming Languages, Semantics, and Systems (PPS). https://pps2018. soic. indiana. edu/files/2017/12/SlicStanPPS. pdf*.

[23] Anjana Gosain and Ganga Sharma. 2015. Static analysis: A survey of techniques and tools. In *Intelligent Computing and Applications: Proceedings of the International Conference on ICA, 22-24 December 2014*. Springer, 581–591.

[24] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.

[25] Matthew D Hoffman, Andrew Gelman, et al. 2014. The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *J. Mach. Learn. Res.* 15, 1 (2014), 1593–1623.

[26] Matthew D Hoffman, Matthew J Johnson, and Dustin Tran. 2018. Autoconj: recognizing and exploiting conjugacy without a domain-specific language. *Advances in Neural Information Processing Systems* 31 (2018).

[27] Gerard J Holzmann. 2017. Cobra: a light-weight tool for static and dynamic program analysis. *Innovations in Systems and Software Engineering* 13, 1 (2017), 35–49.

[28] Chung-Kil Hur, Aditya V Nori, Sriram K Rajamani, and Selva Samuel. 2014. Slicing probabilistic programs. *ACM SIGPLAN Notices* 49, 6 (2014), 133–144.

[29] jianlin. [n. d.]. HMC issue in the Pyro Forum. https://forum.pyro.ai/t/mcmc-discrete-rv-parallelization-is-there-anyway-to-stop-pyro-automatically-vectorizing-tensors/4160. Accessed: 2023-07-24.

[30] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[31] Daphne Koller and Nir Friedman. 2009. *Probabilistic graphical models: principles and techniques*. MIT press.

[32] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

[33] Jianlin Li, Leni Ven, Pengyuan Shi, and Yizhou Zhang. 2023. Type-preserving, dependence-aware guide generation for sound, effective amortized probabilistic inference. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 1454–1482.

[34] Carol Mak, Fabian Zaiser, and Luke Ong. 2021. Nonparametric Hamiltonian Monte Carlo. In *International Conference on Machine Learning*. PMLR, 7336–7347.

[35] Ramon E Moore. 1966. *Interval analysis*. Vol. 4. Prentice-Hall Englewood Cliffs.

[36] Chandrakana Nandi, Dan Grossman, Adrian Sampson, Todd Mytkowicz, and Kathryn S McKinley. 2017. Debugging probabilistic programs. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 18–26.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:1912.01703 [cs.LG]

[38] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. 2015. Tricorder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 598–608.

[39] John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science* 2 (2016), e55.

[40] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S McKinley, Dan Grossman, and Luis Ceze. 2014. Expressing and verifying probabilistic assertions. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 112–122.

[41] Sam Staton, Hongseok Yang, Frank Wood, Chris Heunen, and Ohad Kammar. 2016. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. 525–534.

[42] Nazanin Tehrani, Nimar S Arora, Yucen Lily Li, Kinjal Divesh Shah, David Noursi, Michael Tingley, Narjes Torabi, Eric Lippert, Erik Meijer, et al. 2020. Bean machine: A declarative probabilistic programming language for efficient programmable inference. In *International Conference on Probabilistic Graphical Models*. PMLR, 485–496.

[43] Surya T Tokdar and Robert E Kass. 2010. Importance sampling: a review. *Wiley Interdisciplinary Reviews: Computational Statistics* 2, 1 (2010), 54–60.

[44] Matthijs Vákár, Ohad Kammar, and Sam Staton. 2019. A domain theory for statistical probabilistic programming. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[45] Di Wang, Jan Hoffmann, and Thomas Reps. 2021. Sound probabilistic inference via guide types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 788–803.
[46] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-agnostic representation learning of source code from structure and context. *arXiv preprint arXiv:2103.11318* (2021).

Received 31 July 2023; revised ??; accepted ??