Static Factorisation of Probabilistic Programs With User-Labelled Sample Statements and While Loops

MARKUS BÖCK, TU Wien, Austria
JÜRGEN CITO, TU Wien, Austria

It is commonly known that any Bayesian network can be implemented as a probabilistic program, but the reverse direction is not so clear. In this work, we address the open question to what extent a probabilistic program with user-labelled sample statements and while loops can be represented as a graphical model. To this end, we introduce operational semantics for probabilistic programming languages (PPLs) with user-labelled sample statements that is modelled after real-world PPLs inhabiting this language feature like Gen, Turing, or Pyro. Furthermore, by formally translating a program to its control-flow graph, we define a sound static analysis that approximates the dependency structure of the random variables in the program. As a result, we obtain a factorisation of the implicitly defined program density, which allows us to show the equivalence of programs to either Bayesian or Markov networks. The factorisation result holds even for programs that define an unbounded number of random variables. In a case study, we use the statically obtained structure to speed up posterior inference, which demonstrates that our static analysis enables new opportunities for optimising inference algorithms.

CCS Concepts: • **Theory of computation → Operational semantics**; **Program analysis**; • **Mathematics of computing** → *Bayesian computation*; *Bayesian networks*; *Markov networks.*

Additional Key Words and Phrases: Probabilistic programming, operational semantics, static program analysis, factorisation, Bayesian networks, Markov networks

ACM Reference Format:
Markus Böck and Jürgen Cito. 2024. Static Factorisation of Probabilistic Programs With User-Labelled Sample Statements and While Loops. In *Proceedings of ACM Conference (Conference'24).* ACM, New York, NY, USA, 33 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Probabilistic programming provides an intuitive means to specify probabilistic models as programs. Typically, probabilistic programming systems also come with machinery for automatic posterior inference. This enables practitioners without substantial knowledge in Bayesian inference techniques to work with complex probabilistic models.

Besides these practical aspects, there is also a long history of studying probabilistic programming languages (PPLs) from a mathematical perspective [Barthe et al. 2020; Kozen 1979]. It is commonly known that any Bayesian network [Koller and Friedman 2009] can be implemented as a probabilistic program [Gordon et al. 2014], but the reverse direction is not so clear. The understanding is that first-order probabilistic programs without loops can be compiled to Bayesian networks, because the number of executed sample statements is fixed and finite, such that each sample statement

Authors' Contact Information: Markus Böck, markus.h.boeck@tuwien.ac.at, TU Wien, Vienna, Austria; Jürgen Cito, juergen.cito@tuwien.ac.at, TU Wien, Vienna, Austria.

Conference'24, July 2024, Washington, DC, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

corresponds to exactly one random variable [Borgström et al. 2011; Sampson et al. 2014; van de Meent et al. 2018]. Generally, a graphical representation of a probabilistic program is desirable as it lends itself for optimising inference algorithms by exploiting the dependency structure between random variables. In fact, many probabilistic programming languages require the user to explicitly construct the model in a graphical structure [Lunn et al. 2000; McCallum et al. 2009; Milch et al. 2007; Minka 2012; Salvatier et al. 2016].

However, in the universal probabilistic programming paradigm probabilistic constructs are embedded in a Turing-complete language [Goodman et al. 2012]. While loops and recursive function calls easily lead to an unbounded number of executed sample statements which makes a direct compilation to a graphical representation impossible. Even more, many PPLs allow the user to label each sample statement with a dynamic address. This further complicates matters, because in this case even a single sample statement may correspond to a potentially infinite number of random variables. As a consequence, many approaches resort to dynamic methods to leverage the dependency structure of this kind of probabilistic programs for efficient inference [Chen et al. 2014; Wu et al. 2016; Yang et al. 2014].

In this work, we are expanding the theoretical knowledge about the connection between probabilistic programs and graphical models by proving that the density of a probabilistic program with K user-labelled sample statements decomposes into K factors. This is true even if the program contains while loops. Moreover, the dependency structure implied by the factorisation can be computed *statically* which enables new opportunities for the optimisation of inference algorithms.

To this end, we present density-based operational semantics for PPLs that make use of user-defined addresses in sample statements, like Gen [Cusumano-Towner et al. 2019], Turing [Ge et al. 2018], or Pyro [Bingham et al. 2019]. Addresses are dynamically computed identifiers for random variables, which can be different from the name of the program variable they are assigned to. Below are examples of such address-based sample statements in several PPLs, where the address expressions are highlighted with red:

```
x = {:x => i} ~ Normal(0.,1.)              # Gen
x = pyro.sample(f"x{i}", Normal(0.,1.))    # Pyro
x[i] ~ Normal(0.,1.)                       # Turing
x = sample("x"+str(i), Normal(0.,1.))      # our formal syntax
```

We interpret a probabilistic program as a function p, called program density, that maps a trace $\texttt{tr}$ to its (unnormalised) density $p(\texttt{tr}) \in \mathbb{R}_{\geq 0}$. In this context, a trace is simply a map from addresses to values, where we need not further specify how these values are set. In practice, the values are set depending on the used inference algorithm. For instance, for a Metropolis Hastings algorithm, one could sample these values from a proposal distribution [Wingate et al. 2011], whereas for the Hamiltonian Monte Carlo algorithm one would set these values by simulating a Hamiltonian physics system [Hoffman et al. 2014]. In variational inference methods, these values would be perturbed by gradient ascent [Kucukelbir et al. 2017].

Having described the syntax and semantics of our formal PPL in Section 2, we will prove a factorisation of the program density. Namely, if a program contains K sample statements, then the program density can be written as a product of K terms: $p(\texttt{tr}) = \prod_{k=1}^{K} p_k(\texttt{tr})$. For each factor p_k, we can *statically* determine the set of addresses that contribute to it. To evaluate this factorisation from a practical perspective, we have implemented a static analysis that can generate sub-programs for each factor. In Section 5, we demonstrate that these sub-programs can be used to speed-up the runtime of a Metropolis-Hastings inference algorithm on a benchmark set of 15 probabilistic programs. The majority of these benchmark programs make use of while loops to define an unbounded number of random variables.

1.1 Overview

Section 2. Address-Based Semantics For Probabilistic Programs introduces our semantics for a formal probabilistic programming language. It is a generalisation of the operational semantics of Core Stan [Gorinova et al. 2019]. Instead of interpreting the program density p as a function over only a fixed set of finite variables, we extended this approach to interpret the density as function over traces. In contrast to Gorinova et al., this allows us to consider programs with dynamic user-defined addresses and while loops. These language features greatly expand the class of expressible probabilistic models, for instance, models with an infinite number of random variables.

Section 3. Static Factorisation for Programs Without Loops examines the factorisation of the program density for the simplified case where programs do not contain while loops.

In Section 3.2, we present a static provenance analysis that finds the dependencies of a program variable x at any program state σ. The dependencies are described by specifying for which addresses a trace `tr` needs to be known to also know the value of x in σ. This analysis forms the basis for finding the factorisation over addresses and operates on the control-flow graph (CFG) of a program.

Formally mapping the operational semantics onto the CFG allows us to prove soundness of the provenance analysis in Section 3.2. This is achieved by constructing evaluation functions that map traces directly to the values of variables at program states. These functions require the traces to be only evaluated at the addresses the variable statically depends on.

In Section 3.3, we prove $p(\texttt{tr}) = \prod_{k=1}^{K} p_k(\texttt{tr})$ for programs without loops by explicitly constructing one factor per sample statement p_k from the evaluation functions. If we further restrict the program to static addresses, we arrive at the language of Gorinova et al., where our factorisation theorem implies equivalence to Bayesian networks. We show that already in the case of dynamic addresses, programs are only equivalent to the more general Markov networks, because of the possibility of cyclic dependencies and an infinite number of random variables.

Section 4. Static Factorisation for Programs With Loops presents a way to generalise the factorisation result of Section 3 to the much more challenging case where programs may contain while loops. The idea is to mathematically unroll while loops to obtain a directed acyclic graph that essentially represents all possible program paths. This *unrolled CFG* preserves the program semantics and allows us to effectively reuse the proofs of Section 3. In this case the density still factorises into K terms – one factor per sample statement. Like dynamic addresses, while loops may also be used to model an infinite number of random variables and thus, the resulting factorisation in general implies the equivalence of programs to Markov networks.

Section 5. Case Study: Speeding-Up Posterior Inference demonstrates how the statically obtained factorisation of the program density is non-trivial such that it can be used to significantly decrease the runtime of a Metropolis Hastings inference algorithm. By generating sub-programs for each factor, our static analysis enables the optimisation of the inference algorithms in a way that is typically reserved for PPLs that require the explicit construction of a graphical model. Such a static approach was previously not possible for programs with dynamic addresses or while loops. Several examples of such programs are included in our benchmark set.

2 Address-Based Semantics For Probabilistic Programs

First, we introduce the syntax of our formal probabilistic programming language in terms of expressions and statements similar to Gorinova et al. [2019] and Hur et al. [2015]. We assume that the constants of this language range over a set of values including booleans, integers, real numbers, immutable real-valued vectors, finite-length strings, and a special `null` value. We define $\mathcal{V}$ to be the set of all values. A program contains a finite set of program variables denoted by x, x_i, y, or z. We further assume a set of built-in functions g. We assume that functions g are total and return `null` for erroneous inputs. The only non-standard construct in the language are sample statements. For sample statements, the expression E_0 corresponds to the user-defined sample address, which is assumed to evaluate to a string. The symbol f ranges over a set of built-in distributions which are parameterised by arguments $E_1, \dots, E_n$.

Syntax of Expressions:

$E ::=$	expression
c	constant
x	variable
$g(E_1, \dots, E_n)$	function call

Syntax of Statements:

$S ::=$	statement
$x = E$	assignment
$S_1; S_2$	sequence
$\texttt{if}\ E\ \texttt{then}\ S_1\ \texttt{else}\ S_2$	if statement
$\texttt{skip}$	skip
$\texttt{while}\ E\ \texttt{do}\ S$	while loop
$x = \texttt{sample}(E_0, f(E_1, \dots, E_n))$	sample

2.1 Operational Semantics

The meaning of a program is the density implicitly defined by its sample statements. This density is evaluated for a *program trace* and returns a real number. A program trace is a mapping from finite-length addresses to numeric values, $\texttt{tr} \colon \textbf{Strings} \to \mathcal{V}$. We denote the set of all traces with $\mathcal{T}$. How the value at an address is determined is not important for this work and in practice typically depends on the algorithm that is used to perform posterior inference for the program. For instance, if an address α corresponds to an observed variable, the value $\texttt{tr}(\alpha)$ is fixed to the data. If an address α does not appear in a program, then its value can be assumed to be $\texttt{tr}(\alpha) = \texttt{null}$.

For each trace `tr`, the operational semantics of our language are defined by the big-step relation $(\sigma, S) \Downarrow^{\texttt{tr}} \sigma'$, where S is a statement and $\sigma \in \Sigma$ is a program state – a finite map from program variables to values

$$\sigma ::= x_1 \mapsto V_1, \dots, x_n \mapsto V_n, \quad x_i \text{ distinct, } V_i \in \mathcal{V}.$$

A state can be naturally lifted to a map from expressions to values $\sigma \colon \textbf{Exprs} \to \mathcal{V}$:

$$\sigma(x_i) = V_i, \quad \sigma(c) = c, \quad \sigma(g(E_1, \dots, E_n)) = g(\sigma(E_1), \dots, \sigma(E_n)).$$

The operational semantics for the non-probabilistic part of our language are standard:

$$\frac{}{(\sigma, \texttt{skip}) \Downarrow^{\texttt{tr}} \sigma} \qquad \frac{}{(\sigma, x = E) \Downarrow^{\texttt{tr}} \sigma[x \mapsto \sigma(E)]} \qquad \frac{(\sigma, S_1) \Downarrow^{\texttt{tr}} \sigma' \quad (\sigma', S_2) \Downarrow^{\texttt{tr}} \sigma''}{(\sigma, S_1; S_2) \Downarrow^{\texttt{tr}} \sigma''}$$

$$\frac{\sigma(E) = \texttt{true} \quad (\sigma, S_1) \Downarrow^{\texttt{tr}} \sigma'}{(\sigma, \texttt{if}\ E\ \texttt{then}\ S_1\ \texttt{else}\ S_2) \Downarrow^{\texttt{tr}} \sigma'} \qquad \frac{\sigma(E) = \texttt{false} \quad (\sigma, S_2) \Downarrow^{\texttt{tr}} \sigma'}{(\sigma, \texttt{if}\ E\ \texttt{then}\ S_1\ \texttt{else}\ S_2) \Downarrow^{\texttt{tr}} \sigma'}$$

$$\frac{\sigma(E) = \texttt{false}}{(\sigma, \texttt{while}\ E\ \texttt{do}\ S) \Downarrow^{\texttt{tr}} \sigma} \qquad \frac{\sigma(E) = \texttt{true} \quad (\sigma, (S; \texttt{while}\ E\ \texttt{do}\ S)) \Downarrow^{\texttt{tr}} \sigma'}{(\sigma, \texttt{while}\ E\ \texttt{do}\ S) \Downarrow^{\texttt{tr}} \sigma'}$$

Above, $\sigma[x \mapsto V]$ denotes the updated program state σ', where $\sigma'(x) = V$ and $\sigma'(y) = \sigma(y)$ for all other variables $y \neq x$. The inference rules are to be interpreted inductively such that the semantics of a while loop can be rewritten as

$$\begin{aligned}(\sigma, \texttt{while}\ E\ \texttt{do}\ S) \Downarrow^{\texttt{tr}} \sigma' \iff \quad & \exists n \in \mathbb{N}_0 \colon \big((\sigma, \text{repeat}_n(S)) \Downarrow^{\texttt{tr}} \sigma' \wedge \sigma'(E) = \texttt{false}\big) \wedge \\ & \forall m < n \colon \big((\sigma, \text{repeat}_m(S)) \Downarrow^{\texttt{tr}} \sigma'_m \wedge \sigma'_m(E) = \texttt{true}\big),\end{aligned}$$

where $\text{repeat}_n(S) = (S; \dots; S)$ repeated n-times. In particular, if a while loop does not terminate for state σ then $\nexists \sigma' \colon (\sigma, \texttt{while}\ E\ \texttt{do}\ S) \Downarrow^{\texttt{tr}} \sigma'$. This makes the presented operational semantics partial, where errors and non-termination are modelled implicitly. Formally, $\Downarrow^{\texttt{tr}}$ is the least-relation closed under the stated rules.

At sample statements, we inject the value of the trace at address V_0. We multiply the density, represented by the reserved variable $\mathbf{p}$, with the value of function pdf_f evaluated at $\texttt{tr}(V_0)$. Note that the function pdf_f may be an arbitrary function to real values. However, in the context of probabilistic programming, we interpret pdf_f as the density function of distribution f with values in $\mathbb{R}_{\geq 0}$. Lastly, the value $\texttt{tr}(V_0)$ is stored in the program state at variable x. The semantics of sample statements are summarised in the following rule:

$$\frac{\forall i\colon \sigma(E_i) = V_i \wedge V_i \neq \texttt{null} \qquad V_0 \in \textbf{Strings} \qquad V = \texttt{tr}(V_0) \wedge V \neq \texttt{null}}{(\sigma, x = \texttt{sample}(E_0, f(E_1, \dots, E_n))) \Downarrow^{\texttt{tr}} \sigma[x \mapsto V, \mathbf{p} \mapsto \sigma(\mathbf{p}) \times \text{pdf}_f(V; V_1, \dots, V_n)]}$$

Finally, the meaning of a program S is defined as the mapping of trace to density. This definition is well-defined, because the semantics are deterministic for each trace. If a trace leads to errors or non-termination, then the density is undefined.

Definition 1. *The density of a program S is the function $p_S\colon \mathcal{T} \to \mathbb{R}_{\geq 0}$ given by*

$$p_S(\texttt{tr}) \coloneqq \begin{cases} \sigma(\boldsymbol{p}) & \textit{if}\ \exists\, \sigma\colon ((x_i \mapsto \texttt{null}, \boldsymbol{p} \mapsto 1), S) \Downarrow^{\texttt{tr}} \sigma \\ \text{undefined} & \textit{otherwise} \end{cases}$$

Listing 1. Simple probabilistic program with stochastic branching.

```
p = sample("p", Uniform(0,1))
x = sample("x", Bernoulli(p))
if x == 1 then
    y = sample("y", Bernoulli(0.25))
else
    z = sample("z", Bernoulli(0.75))
```

Listing 2. Program of Listing 1 rewritten with dynamic addresses.

```
p = sample("p", Uniform(0,1))
x = sample("x", Bernoulli(p))
if x == 1 then
    addr = "y"; prob = 0.25;
else
    addr = "z"; prob = 0.75;
r = sample(addr, Bernoulli(prob))
```

Consider the simple probabilistic program in Listing 1. The density defined by our operational semantics is given below, where δ_v denotes the delta function whose value is 1 if v equals $\texttt{true}$ and 0 otherwise.

$$\begin{aligned} p(\texttt{tr}) = {} & \text{pdf}_{\text{Uniform}}(\texttt{tr}(\text{"p"}); 0, 1) \\ & \times \text{pdf}_{\text{Bernoulli}}(\texttt{tr}(\text{"x"}); \texttt{tr}(\text{"p"})) \\ & \times \left(\delta_{\texttt{tr}(\text{"x"})} \text{pdf}_{\text{Bernoulli}}(\texttt{tr}(\text{"y"}); 0.25) + (1 - \delta_{\texttt{tr}(\text{"x"})})\right) \\ & \times \left(\delta_{\texttt{tr}(\text{"x"})} + (1 - \delta_{\texttt{tr}(\text{"x"})}) \text{pdf}_{\text{Bernoulli}}(\texttt{tr}(\text{"z"}); 0.75)\right). \end{aligned}$$

As the addresses in Listing 1 are static and identical to the program variables, the program is well-described by many existing PPL semantics. However, in Listing 2, we rewrite the program with *dynamic addresses* which results in the same above density. Most formal PPLs studied in prior work do not support user-defined addresses for sample statements and cannot interpret a program with three sample statements as a four-dimensional joint density. In contrast, our presented semantics correctly interprets the program with dynamic addresses. This interpretation accurately describes the density of programs written in PPLs that lazily evaluate if statements like Gen or Pyro.

To the best of our knowledge, there are only two existing semantic models that support user-defined addresses [Lee et al. 2019; Lew et al. 2023]. These approaches are denotational and sampling-based as opposed to our density-based operational semantics. As sampling-based semantics they

Listing 3. Program with duplicate address.

```
x = sample("x", Normal(0.,1.))
x = sample("x", Normal(2*x+1,1.))
```

Listing 4. Program with random address.

```
n = sample("n", Poisson(5))
x = sample("x_"+str(n), Normal(0.,1.))
```

Listing 5. Program with while loop implementing a Geometric distribution.

```
b = true; i = 0;
while b do
  i = i + 1
  b = sample("b_"+i, Bernoulli(0.25))
```

require that an address appears only once during the execution of a program. Our semantics do not rely on this assumption and can explain programs like the one shown in Listing 3. Even though the program does not correspond to a probability distribution, it can in fact be translated to density-based PPLs like Stan or Turing. Lastly, the presented semantics correctly describe programs with while loops and random addresses. Such programs are significantly harder to describe, because both cases easily lead to an unbounded number of addresses as can be seen in Listing 4 and Listing 5.

2.2 Remark Regarding a Measure-theoretic Interpretation

Even though it suffices to consider the explicit computation of the density of a probabilistic program in this work, a measure-theoretic interpretation of the presented operational semantics is still worth exploring. The density of a measure μ with respect to some reference measure ν is formally given by the Radon-Nikodym derivative p such that $\mu(A) = \int_A p \, d\nu$. Thus, the measure-theoretic interpretation of the operational semantics is described by a construction of a measure space of traces $(\mathcal{T}, \Sigma, \nu)$ such that the density p_S is measurable. Then, $\mu_S(A) = \int_A p_S \, d\nu$ is the measure denoted by the program S. See Appendix C for a possible construction of this measure space.

3 Static Factorisation for Programs Without Loops

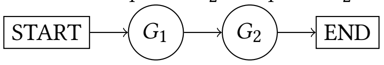
The goal of this work is to statically find a factorisation of the density implicitly defined by a probabilistic program in terms of addresses. This static analysis will operate on the control-flow graph (CFG) of a program. We begin by considering programs without loops first and describe how we can formally translate such a program to its corresponding CFG. We will equip this CFG with semantics equivalent to the operational semantics of the original program. The following construction is fairly standard but necessary for formally verifying the correctness of the factorisation.

3.1 Control-Flow Graph

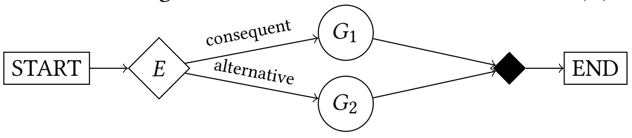
A CFG has five types of nodes: start, end, assign, branch, and join nodes. Branch nodes have two successor nodes, end nodes have no successor, all other nodes have one. Only join nodes have multiple predecessors. The CFG contains exactly one start node from which every other node is reachable. Further, it contains exactly one end node that is reachable from any other node. We describe the recursive translation rules for program S as diagrams, which serve to convey the general idea of the construction. For a formal description see Appendix A. Sub-graphs are drawn by circular nodes. Branch and join nodes are drawn as diamond nodes, the latter filled black. Assign, start, and end nodes have rectangular shape. In text, branch nodes are written as Branch(E). Assign nodes are denoted with Assign($x = E$) and Assign($x = \texttt{sample}(E_0, \dots)$).

- The CFG of a skip-statement, $S = \texttt{skip}$, consists of only start and end node:
$$\begin{tikzcd} \boxed{\text{START}} \arrow[r] & \boxed{\text{END}} \end{tikzcd}$$
- The CFG of an assignment, $x = E$, or sample statement $x = \texttt{sample}(E_0, f(E_1, \dots, E_n))$, is a sequence of start, assign, and end node:
$$\begin{tikzcd} \boxed{\text{START}} \arrow[r] & \boxed{x = E} \arrow[r] & \boxed{\text{END}} \end{tikzcd} \qquad \begin{tikzcd} \boxed{\text{START}} \arrow[r] & \boxed{x = \texttt{sample}(E_0, f(E_1, \dots, E_n))} \arrow[r] & \boxed{\text{END}} \end{tikzcd}$$

- The CFG for a sequence of statements, $S = S_1; S_2$, is recursively defined by "stitching together" the CFGs G_1 and G_2 of S_1 and S_2:



- The CFG of an if statement, $S = (\texttt{if } E \texttt{ then } S_1 \texttt{ else } S_2)$, is also defined in terms of the CFGs of S_1 and S_2 together with a branch node $B = \text{Branch}(E)$ and a join node J:



We denote the two successors of the branch node B as consequent(B) and alternative(B). Furthermore, we say the tuple (B, J) is a brain-join pair and define BranchJoin(G) to be the set of all branch-join pairs of G.

3.1.1 CFG Semantics. For a control-flow graph G we define the small-step semantics for a trace $\texttt{tr}$ as a relation that models the transition from node to node depending on the current program state:

$$(\sigma, N) \xrightarrow{\texttt{tr}} (\sigma', N')$$

The semantics for the CFG of program S are set-up precisely such that they are equivalent to the operational semantics of program S. The transition rules depending on node type and program state are given below:

$$\frac{N = \text{START} \quad N' = \text{successor}(N)}{(\sigma, N) \xrightarrow{\texttt{tr}} (\sigma, N')} \qquad \frac{N = \text{Assign}(x = E) \quad N' = \text{successor}(N)}{(\sigma, N) \xrightarrow{\texttt{tr}} (\sigma[x \mapsto \sigma(E)], N')}$$

$$\frac{N = \text{Join} \quad N' = \text{successor}(N)}{(\sigma, N) \xrightarrow{\texttt{tr}} (\sigma, N')} \quad \frac{N = \text{Branch}(E) \quad \sigma(E) = \texttt{true}}{(\sigma, N) \xrightarrow{\texttt{tr}} (\sigma, \text{consequent}(N))} \quad \frac{N = \text{Branch}(E) \quad \sigma(E) = \texttt{false}}{(\sigma, N) \xrightarrow{\texttt{tr}} (\sigma, \text{alternative}(N))}$$

$$\frac{\begin{array}{c} N = \text{Assign}(x = \texttt{sample}(E_0, f(E_1, \ldots, E_n))) \quad N' = \text{successor}(N) \\ \forall i \colon \sigma(E_i) = V_i \wedge V_i \neq \texttt{null} \quad V_0 \in \textbf{Strings} \quad V = \texttt{tr}(V_0) \wedge V \neq \texttt{null} \end{array}}{(\sigma, N) \xrightarrow{\texttt{tr}} (\sigma[x \mapsto V, \mathbf{p} \mapsto \sigma(\mathbf{p}) \times \text{pdf}_f(V; V_1, \ldots, V_n)], N')}$$

Finally, the meaning of a CFG is a mapping from trace to density. It is defined for traces $\texttt{tr}$ if there exists a path of node transitions from start to end node.

Definition 2. *Let $\sigma_0 = (x_i \mapsto \texttt{null}, \boldsymbol{p} \mapsto 1)$ be the initial program state. The density of a CFG G is the function $p_G \colon \mathcal{T} \to \mathbb{R}_{\geq 0}$ given by*

$$p_G(\texttt{tr}) := \begin{cases} \sigma(\boldsymbol{p}) & \text{if } \exists\, \sigma \in \Sigma, n \in \mathbb{N} \colon \forall i = 1, \ldots, n \colon \exists\, \sigma_i, N_i \colon \\ & (\sigma_0, START) \xrightarrow{\texttt{tr}} \ldots \xrightarrow{\texttt{tr}} (\sigma_i, N_i) \xrightarrow{\texttt{tr}} \ldots \xrightarrow{\texttt{tr}} (\sigma, END) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Proposition 1. *For all programs S without while loops and corresponding control-flow graph G, it holds that for all traces* $\texttt{tr}$

$$p_S(\texttt{tr}) = p_G(\texttt{tr}).$$

Proof. By structural induction. See Appendix B.1. □

3.2 Static Provenance Analysis

The provenance of variable x in CFG node N are all the addresses $\subseteq$ **Strings** that contribute to the computation of its value in node N. In this section, we define an algorithm that operates on the CFG and statically determines the provenance for any variable x at any node N. As this is a static algorithm, it is an over-approximation of the true provenance. In Section 3.2.1, we will prove soundness of the presented algorithm. We show that indeed the value of variable x at node N can be computed from the values in the trace `tr` at the addresses which make up the provenance. But first, we introduce some useful concepts which are similar to definitions found in data-flow analysis [Cooper and Torczon 2011; Hennessy and Patterson 2011].

Definition 3. *For a CFG G, let* AssignNodes(G, x) *be all nodes N' in G that assign variable x, $N' = \text{Assign}(x = \dots)$. The set of* reaching definitions *for variable x in node N is defined as*

$$\text{RD}(N, x) = \{N' \in \text{AssignNodes}(G, x) : \exists \textit{ path } (N', \dots, N_i, \dots, N),\ N_i \notin \text{AssignNodes}(G, x)\}.$$

Intuitively, the set of reaching definitions are all nodes that could have written the value of x in the program state before executing node N.

Definition 4. *For CFG G and node N the set of* branch parents *is defined as*

$$\text{BP}(N) = \{B : (B, J) \in \text{BranchJoin}(G) \land \exists \textit{ path } (B, \dots, N, \dots, J)\}.$$

A branch node B and join node J are a branch-join pair $(B, J) \in$ BranchJoin(G) if they belong to the same if-statement (see the translation rule for if-statements to CFGs).

The set of branch parents are all parent branch nodes of N that determine if the program branch that N belongs to is executed.

Lastly, we need the set of all strings that the address expression of a sample statement may evaluate to.

Definition 5. *For a sample CFG node $N = \text{Assign}(x = \texttt{sample}(E_0, f(E_1, \dots, E_n)))$, we define the set of all possible addresses as*

$$\text{addresses}(N) = \{\sigma(E_0) : \sigma \in \Sigma\} \subseteq \textbf{Strings}.$$

Now we are able to define the algorithm to statically approximate the provenance of any variable x at any node N denoted by prov(N, x) in Algorithm 1. This algorithm is inspired by standard methods for dependence analysis. In the algorithm we use vars(E), which is defined as the set of all program variables in expression E. It is often useful to determine the provenance of an entire expression instead of only a single variable. For expressions E, we lift the definition of prov

$$\text{prov}(N, E) = \textstyle\bigcup_{y \in \text{vars}(E)} \text{prov}(N, y). \tag{1}$$

The algorithm works as follows. If we want to find prov(N, x), we first have to find all reaching definitions for x. If the reaching definition N' is a sample node, $x = \texttt{sample}(E_0, \dots)$, then the value of x can only dependent on the addresses generated by E_0, addresses(N'), and the provenance of E_0. If N' is an assignment node, $x = E$, we have to recursively find the provenance of the expression E. Lastly, there can be multiple reaching definitions for x in different branches and the value of x also depends on the branching condition. Thus, we also find all branch parents and recursively determine their provenance and add it to the final result.

In practice the potentially infinite set addresses(N) can be represented by the CFG node N itself. An implementation of Algorithm 1 would return a set of CFG sample nodes rather than a potentially infinite set of strings.

Algorithm 1 Computing the provenance set prov(N, x) statically from the CFG.

```
1:  **Input** CFG node $N$, variable $x$
2:  prov $\leftarrow \emptyset$
3:  queue $\leftarrow [(N, x)]$
4:  **while** queue is not empty **do**
5:      $(N, x) \leftarrow$ queue.pop()
6:      **for** $N' \in \mathrm{RD}(N, x)$ **do**
7:          **if** $N' = \mathrm{Assign}(x = \texttt{sample}(E_0, f(E_1, \ldots, E_n)))$ **then**
8:              prov $\leftarrow$ prov $\cup$ addresses$(N')$
9:              $E' \leftarrow E_0$                    ▷ $E_0$ comes from pattern-matching $N'$
10:         **else if** $N' = \mathrm{Assign}(x = E)$ **then**
11:             $E' \leftarrow E$                      ▷ $E$ comes from pattern-matching $N'$
12:         **for** $y \in \mathrm{vars}(E')$ **do**
13:             **if** is_unmarked$((N', y))$ **then**
14:                 mark$((N', y))$
15:                 queue.push$((N', y))$
16:         **for** $N_{\mathrm{bp}} \in \mathrm{BP}(N')$        ▷ $N_{\mathrm{bp}} = \mathrm{Branch}(E_{N_{\mathrm{bp}}})$ **do**
17:             **for** $y \in \mathrm{vars}(E_{N_{\mathrm{bp}}})$ **do**
18:                 **if** is_unmarked$((N_{\mathrm{bp}}, y))$ **then**
19:                     mark$((N_{\mathrm{bp}}, y))$
20:                     queue.push$((N_{\mathrm{bp}}, y))$
21: **return** prov
```

We get a clearer sense of the introduced definitions and Algorithm 1 by considering a simple program in Fig. 1. On the right you can see examples for reaching definitions and branch parents for several nodes drawn as arrows. These arrows illustrate how Algorithm 1 computes the provenance of variables m and s at the sample node for x. Note that if we were to change the if statement to `(if (b == 1) then (m = 1) else (m = 1))`, then the variable "b" would still be in the provenance set of m. This illustrates that the presented algorithm can only over-approximate the provenance and not compute it exactly.

```
b = sample("b", Bernoulli(0.5))
s = sample("s", InverseGamma(1.,1.))
if b == 1 then
    m = sample("mu", Normal(0.,1.))
else
    m = 1
x = sample("x", Normal(m, s))
```

$$N = \mathrm{Assign}(\mathtt{x} = \ldots)$$

$$V_{\mathrm{s}}^{N}(\mathtt{tr}) = \mathtt{tr}(\text{"s"})$$

$$V_{\mathrm{m}}^{N}(\mathtt{tr}) = \mathrm{ife}(\mathtt{tr}(\text{"b"}), \mathtt{tr}(\text{"mu"}), 1)$$

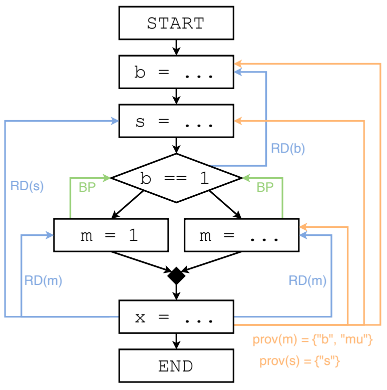


Fig. 1. Reaching definitions, branch parents, and the provenance set for a simple example. The evaluation functions introduced in Section 3.2.1 are also given for node N.

3.2.1 Soundness. Recall that the purpose of finding the provenance of variable x at node N is to determine which addresses contribute to the computation of its value. To prove that Algorithm 1 indeed finds the correct set of addresses, first, we have to precisely define what we mean that a set of addresses contribute to the computation. Suppose that there is a function f that maps a trace $\texttt{tr}$ to a value v. The set of addresses A contribute to the computation of value v, if any other trace $\texttt{tr}'$ that has the same values as $\texttt{tr}$ for all addresses $\alpha \in A$, gets mapped to the same value $f(\texttt{tr}') = v$. In other words, if we change the trace $\texttt{tr}$ at an address $\beta \in \textbf{Strings} \setminus A$, the value of $f(\texttt{tr})$ remains unchanged. Or equivalently, if $f(\texttt{tr}) \neq f(\texttt{tr}')$, then there must be an address $\alpha \in A$ such that $\texttt{tr}(\alpha) \neq \texttt{tr}'(\alpha)$.

Definition 6. *For a function from traces to an arbitrary set B, $f : \mathcal{T} \to B$, we write $f \in [\mathcal{T}|_A \to B]$ for a subset of addresses $A \subseteq$ **Strings**, if and only if*

$$\forall \alpha \in A\colon \texttt{tr}(\alpha) = \texttt{tr}'(\alpha) \implies f(\texttt{tr}) = f(\texttt{tr}').$$

*That is, changing the values of $\texttt{tr}$ at addresses $\beta \in$ **Strings** $\setminus A$ does not change the value $f(\texttt{tr})$.*

The next lemma states that we can always over-approximate the set of addresses that contribute to the computation of the values $f(\texttt{tr})$.

Lemma 1. *If $A \subseteq A'$ and $f \in [\mathcal{T}|_A \to B]$, then $f \in [\mathcal{T}|_{A'} \to B]$. In other words,*

$$A \subseteq A' \implies [\mathcal{T}|_A \to B] \subseteq [\mathcal{T}|_{A'} \to B]. \tag{2}$$

If we combine two functions with provenance A_1 and A_2 respectively, then the resulting function has provenance $A_1 \cup A_2$. This is shown in the following lemma.

Lemma 2. *Let $f_1 \in [\mathcal{T}|_{A_1} \to B_1]$, $f_2 \in [\mathcal{T}|_{A_2} \to B_2]$, $h\colon B_1 \times B_2 \to C$. Then*

$$\texttt{tr} \mapsto h(f_1(\texttt{tr}), f_2(\texttt{tr})) \in [\mathcal{T}|_{A_1 \cup A_2} \to C].$$

This brings us to the main soundness proof of Algorithm 1. Proposition 2 states that we can equip each CFG node N with evaluation functions that depend on the addresses determined by prov in the sense of Definition 6. The evaluation functions directly map the trace $\texttt{tr}$ to the values of variables at node N such that the values agree with the small-step CFG semantics relation $\xrightarrow{\texttt{tr}}$. Importantly, while the operational semantics are defined for each individual trace, the evaluation functions work for all traces. In Fig. 1, you can see concrete examples of these evaluation functions for a simple program.

Proposition 2. *For each node N in the CFG G and variable x, there exists an evaluation function*

$$V_x^N \in [\mathcal{T}|_{\text{prov}(N,x)} \to \mathcal{V}],$$

such that for all traces $\texttt{tr}$ with $\sigma_0 = (x_i \mapsto \texttt{null}, \boldsymbol{p} \mapsto 1)$ and execution sequence

$$(\sigma_0, START) \xrightarrow{\texttt{tr}} \cdots \xrightarrow{\texttt{tr}} (\sigma_i, N_i) \xrightarrow{\texttt{tr}} \cdots \xrightarrow{\texttt{tr}} (\sigma_l, END)$$

we have

$$\sigma_i(x) = V_x^{N_i}(\texttt{tr}).$$

Proof. The CFG G is a directed acyclic graph and the statement is proven by induction on G. See Appendix B.2. □

Intuitively, V_x^N computes the value of x before executing N. These evaluation functions V_x^N can be lifted to evaluation functions for expressions V_E^N:

$$V_c^N(\texttt{tr}) = c, \quad V_{g(E_1,\ldots,E_n)}^N(\texttt{tr}) = g(V_{E_1}^N(\texttt{tr}), \ldots, V_{E_n}^N(\texttt{tr})).$$

By Lemma 2, if $V_y^N \in [\mathcal{T}|_{\text{prov}(N,y)} \to \mathcal{V}]$ for all $y \in \text{vars}(E)$, then $V_E^N \in [\mathcal{T}|_{\text{prov}(N,E)} \to \mathcal{V}]$.

3.3 Static Factorisation Theorem for Programs Without Loops

With Proposition 2, we can prove the first factorisation theorem by combining the evaluation functions V_x^N to express the program density p in a factorised form.

Theorem 1. *Let G be the CFG for a program S without while loop statements. Let $N_1, \dots, N_K$ be all sample nodes in G. For each sample node $N_k = \text{Assign}(x_k = \texttt{sample}(E_0^k, f^k(E_1^k, \dots, E_{n_k}^k)))$, let*

$$A_k = \text{addresses}(N_k) \cup \textstyle\bigcup_{i=0}^{n_k} \text{prov}(N_k, E_i^k) \cup \bigcup_{N' \in \text{BP}(N_k)} \text{prov}(N', E_{N'}).$$

Then, there exist functions $p_k \in [\mathcal{T}|_{A_k} \to \mathbb{R}_{\geq 0}]$ such that if $p_S(\texttt{tr}) \neq \text{undefined}$, then

$$p_S(\texttt{tr}) = p_G(\texttt{tr}) = \prod_{k=1}^{K} p_k(\texttt{tr}).$$

Proof. For each sample node N_k, define $b_k(\texttt{tr}) \coloneqq \bigwedge_{N' \in \text{BP}(N_k)} t_{N'}^k(V_{E_{N'}}^{N'}(\texttt{tr}))$ where $t_{N'}^k(v) = v$ if N_k is in the consequent branch of N' and $t_{N'}^k(v) = \neg v$ if N_k is in the alternate branch. By Lemma 2, we have

$$b_k \in [\mathcal{T}|_{\bigcup_{N' \in \text{BP}(N_k)} \text{prov}(N', E_{N'})} \to \{\texttt{true}, \texttt{false}\}].$$

Above, $V_{E_{N'}}^{N'}(\texttt{tr})$ are precisely the evaluations of the branch conditions for node N_k. Thus, $b_k(\texttt{tr}) = \texttt{true}$ if N_k is in the execution sequence for $\texttt{tr}$ else $\texttt{false}$. Define the factor p_k as

$$p_k(\texttt{tr}) \coloneqq \delta_{b_k(\texttt{tr})} \text{pdf}_{f^k} \left(\texttt{tr}(V_{E_0^k}^{N_k}(\texttt{tr})); V_{E_1^k}^{N_k}(\texttt{tr}), \dots, V_{E_{n_k}^k}^{N_k}(\texttt{tr}) \right) + (1 - \delta_{b_k(\texttt{tr})}).$$

By Proposition 2, we have $V_{E_i^k}^{N_k} \in [\mathcal{T}|_{\text{prov}(N_k, E_i^k)} \to \mathcal{V}]$ and $V_{E_0^k}^{N_k} \in [\mathcal{T}|_{\text{prov}(N_k, E_0^k)} \to \text{addresses}(N_k)]$. Thus, it follows from Lemma 2 that $p_k \in [\mathcal{T}|_{A_k} \to \mathbb{R}_{\geq 0}]$ by construction.

□

Again, consider the example program in Fig. 1. The density of this model factorises like below:

$$\begin{aligned}
p(\texttt{tr}) &= \text{pdf}_{\text{Bernoulli}}(\texttt{tr}(\text{"b"}); 0.5) && \in [\mathcal{T}|_{\{\text{"b"}\}} \to \mathbb{R}_{\geq 0}] \\
&\times \text{pdf}_{\text{InverseGamma}}(\texttt{tr}(\text{"s"}); 1, 1) && \in [\mathcal{T}|_{\{\text{"s"}\}} \to \mathbb{R}_{\geq 0}] \\
&\times \left(\delta_{\texttt{tr}(\text{"b"})} \text{pdf}_{\text{Normal}}(\texttt{tr}(\text{"mu"}); 0, 1) + (1 - \delta_{\texttt{tr}(\text{"b"})})\right) && \in [\mathcal{T}|_{\{\text{"b"},\text{"mu"}\}} \to \mathbb{R}_{\geq 0}] \\
&\times \text{pdf}_{\text{Normal}}(\texttt{tr}(\text{"x"}); V_{\texttt{m}}^N(\texttt{tr}), V_{\texttt{s}}^N(\texttt{tr})) && \in [\mathcal{T}|_{\{\text{"x"},\text{"b"},\text{"mu"},\text{"s"}\}} \to \mathbb{R}_{\geq 0}]
\end{aligned}$$

This is exactly the factorisation obtained from Theorem 1. However, if we would replace the if-statement with a nonsensical one, `(if (b == 1) then (m = 1) else (m = 1))`, then the fourth factor is in $[\mathcal{T}|_{\{\text{"x"},\text{"b"},\text{"s"}\}} \to \mathbb{R}_{\geq 0}]$ with $V_{\texttt{m}}^N(\texttt{tr}) = \text{ife}(\texttt{tr}(\text{"b"}), 1, 1) = 1$. In this case, the static analysis over-approximates the true provenance with the spurious dependency on "b".

3.3.1 Equivalence to Bayesian Networks. If we restrict the way sample statements are used in our PPL, we can establish an equivalence to Bayesian networks. A Bayesian network [Koller and Friedman 2009] is a directed acyclic graph $\mathcal{G}$ over a finite set of nodes $X_1, \dots, X_n$ which represent random variables, such that the joint distribution factorises according to the graph $\mathcal{G}$,

$$P(X_1, \dots, X_n) = \prod_{i=1}^{n} P(X_i \mid \text{parents}_{\mathcal{G}}(X_i)).$$

If we assume that all sample statements of a program have a constant unique address α_k, $N_k = \text{Assign}(x_k = \texttt{sample}(\alpha_k, f^k(E_1^k, \dots, E_{n_k}^k)))$, then we can identify each factor $p_k \in [\mathcal{T}|_{A_k} \to \mathbb{R}_{\geq 0}]$ of Theorem 1 with the unique address $\alpha_k \in$ **Strings**. We construct the Bayesian network $\mathcal{G}$ by mapping each address to a random variable X_{α_k}. Since α_k is unique, p_k is the only relevant factor for X_{α_k} and can be interpreted as conditional density function of X_{α_k}. To see this, rewrite

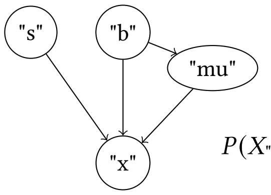
$$p_k(\mathtt{tr}) = \delta_{b_k(\mathtt{tr})}\mathrm{pdf}_{f^k}\left(\mathtt{tr}(\alpha_k); V^{N_k}_{E^k_1}(\mathtt{tr}), \dots, V^{N_k}_{E^k_{n_k}}(\mathtt{tr})\right) + (1 - \delta_{b_k(\mathtt{tr})})\mathbf{1}_{\mathtt{null}}(\mathtt{tr}(\alpha_k)),$$

where the function $\mathbf{1}_{\mathtt{null}}(v)$ equals 1 if $v = \mathtt{null}$ else 0. If the node N_k is not executed, then the value of X_{α_k} can be assumed to be null. As pdf_{f_k} is a probability density function and $\mathbf{1}_{\mathtt{null}}$ is the density of a Dirac distribution centered at null, the factor p_k is also a density function for each choice of values in tr at addresses $\alpha \in A_k \setminus \{\alpha_k\}$. Thus, we introduce the edge $\alpha_j \longrightarrow \alpha_k$ if $\alpha_j \in A_k \setminus \{\alpha_k\}$, such that $\mathrm{parents}_{\mathcal{G}}(X_{\alpha_k}) = A_k \setminus \{\alpha_k\}$ and

$$P(X_{\alpha_k} \,|\, \mathrm{parents}_{\mathcal{G}}(X_{\alpha_k})) = p_k(\{\alpha_j \mapsto X_j \colon \alpha_j \in A_k\}).$$

In the above construction, it is important that each sample statement has an *unique* address. This guarantees no cyclic dependencies and a well-defined conditional probability distributions.

Below you can see the Bayesian network equivalent to the example program of Fig. 1. Since the value of the program variable m depends on which program branch is taken during execution, the random variable $X_{\text{"x"}}$ not only depends on $X_{\text{"mu"}}$, but also on $X_{\text{"b"}}$.



$$\begin{aligned}
P(X_{\text{"b"}}) &= \mathrm{pdf}_{\text{Bernoulli}}(X_{\text{"b"}}; 0.5)\\
P(X_{\text{"s"}}) &= \mathrm{pdf}_{\text{InverseGamma}}(X_{\text{"s"}}; 1, 1)\\
P(X_{\text{"mu"}} | X_{\text{"b"}}) &= \delta_{X_{\text{"b"}}}\mathrm{pdf}_{\text{Normal}}(X_{\text{"mu"}}; 0, 1) + (1 - \delta_{X_{\text{"b"}}})\mathbf{1}_{\mathtt{null}}(X_{\text{"mu"}})\\
P(X_{\text{"x"}} | X_{\text{"b"}}, X_{\text{"mu"}}, X_{\text{"s"}}) &= \mathrm{pdf}_{\text{Normal}}(X_{\text{"x"}}; \mathrm{ife}(\delta_{X_{\text{"b"}}}, X_{\text{"mu"}}, 1), X_{\text{"s"}})
\end{aligned}$$

3.3.2 Equivalence to Markov Networks. If the sample addresses are not unique or not constant, then the factorisation is in general not a Bayesian network. However, we can show equivalence to the more general Markov networks. A Markov network [Koller and Friedman 2009] is an *undirected* graph $\mathcal{H}$ over random variables X_i that represents their dependencies. We consider Markov networks where the joint distribution of X_i factorises over a set of cliques $\mathcal{D} \subseteq \mathrm{cliques}(\mathcal{H})$:

$$P(\vec{X}) = \prod_{D \in \mathcal{D}} \phi_D(D).$$

We again identify a random variable X_α for each address $\alpha \in \bigcup_{k=1}^{K} A_k$ and construct a Markov network $\mathcal{H}$ by connecting node X_α to X_β if $\alpha \in A_k \wedge \beta \in A_k$ for any k. Thus, $D_k = \{X_\alpha \colon \alpha \in A_k\}$ forms a clique and p_S factorises over $\mathcal{H}$ with $\phi_{D_k}(D_k) = p_k(\{\alpha_j \mapsto X_j \colon \alpha_j \in A_k\})$.

Consider the program with constant but non-unique sample addresses in Listing 6. With nine sample statements we get a Markov network over $\{X_{\text{"F"}}, X_{\text{"P0"}}, X_{\text{"P1"}}, X_{\text{"D0"}}, X_{\text{"D1"}}\}$. The nine factors correspond to the address sets {"F"}, {"F", "P0"}, {"F", "P1"}, {"F", "D0", "P1"}, {"F", "D1", "P0"}, $2 \times$ {"F", "P0", "D0"}, and $2 \times$ {"F", "P1", "D1"}. In the consequent branch of the program the distribution in the sample statement with address "P1" depends on "D0", while in the alternative branch, the distribution of "P0" depends on "D1". Thus, there is the dependency cycle "P0" $\longrightarrow$ "D0" $\longrightarrow$ "P1" $\longrightarrow$ "D1" $\longrightarrow$ "P0", which means the obtained factorisation is not a Bayesian network representation for the program. Note that the program can also be rewritten with dynamic addresses to eliminate the if statement. In this case, the factorisation still admits cyclic dependencies.

Lastly, while the programs considered up until this point were equivalent to finite Bayesian networks or finite Markov networks, note that the small program in Listing 4 is equivalent to a Markov network with an infinite number of nodes $\{X_{\text{"n"}}\} \cup \{X_{\text{"x\_}i\text{"}} \colon i \in \mathbb{N}\}$.

Listing 6. Probabilisitic program for the hurricane example of [Milch et al. 2005].

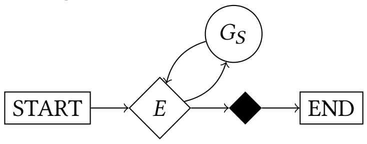
```
first_city_ixs = sample("F", Bernoulli(0.5))
if first_city_ixs == 0 then
    prep_0 = sample("P0", Bernoulli(0.5))
    damage_0 = sample("D0", Bernoulli(prep_0 == 1 ? 0.20 : 0.80))
    prep_1 = sample("P1", Bernoulli(damage_0 == 1 ? 0.75 : 0.50))
    damage_1 = sample("D1", Bernoulli(prep_1 == 1 ? 0.20 : 0.80))
else
    prep_1 = sample("P1", Bernoulli(0.5))
    damage_1 = sample("D1", Bernoulli(prep_1 == 1 ? 0.20 : 0.80))
    prep_0 = sample("P0", Bernoulli(damage_1 == 1 ? 0.75 : 0.50))
    damage_0 = sample("D0", Bernoulli(prep_0 == 1 ? 0.20 : 0.80))
```

4 Static Factorisation for Programs With Loops

In this section, we describe how the argument for proving the factorisation theorem for programs without loops, can be extended to generalise the result to programs with loops.

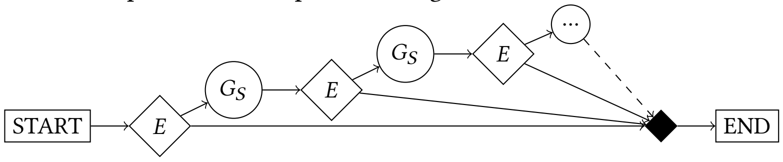
4.1 Unrolled Control-Flow Graph

The translation rules for programs to their CFG can be extended to support while loops as follows. Let G_S be the sub-CFG of statement S. The CFG of (`while` E do S) is given by



For this CFG, prov(N, x) is still well-defined by Algorithm 1, but the proof of Proposition 2 does not work anymore since it requires a directed acyclic graph.

To establish Proposition 2 for programs with while loops, we introduce a second type of control-flow graph, the *unrolled CFG*, which again is a directed acyclic graph. All translation rules are as before except for while loops, which is given below.



The resulting graph G contains for every $i \in \mathbb{N}$ branch nodes B_i = Branch(E) and *copies* of the sub-CFG G_i of statement S. It contains a single join node J for which we have $(B_i, J) \in$ BranchJoin(G) for all i. Let N^i_{first} be successor of the start node in G_i and N^i_{last} the predecessor of the end node in G_i. In the new graph G, the first branch node B_1 is successor of the start node and the end node is the successor of the join node J. The branch nodes B_i have two successors N^i_{first} and J. N^i_{last}, the last node of sub-graph G_i, resumes in the next branch node B_{i+1}.

This definition implies that the number of nodes of G is countable. Furthermore, since the unrolled CFG consists only of start, end, branch, join, and assign nodes, we can equip it with the same semantics as the standard CFG. As before, the semantics are equivalent.

Proposition 3. *For all program S with CFG G and unrolled CFG H, it holds that for all traces* tr

$$p_S(\texttt{tr}) = p_G(\texttt{tr}) = p_H(\texttt{tr}).$$

Proof. By structural induction. See Appendix B.3. □

As mentioned, Algorithm 1 still works for the unrolled CFG. However, since the unrolled CFG contains an infinite number of nodes, in practice, we run this algorithm on the standard CFG with a finite number of nodes. We will show that the computations on the standard CFG produce an over-approximation of the provenance set in the unrolled CFG. But first, we define the connection between the two graph types.

Definition 7. *Let G be the CFG and H the unrolled CFG of program S. For each node $M \in H$, we define $\iota_{\text{cfg}}(M) \in G$ as the node from which M was copied in the construction of H.*

The definition of ι_{cfg} is best understood when considering the example in Fig. 2, where you can see the CFG and unrolled CFG of a program with the mapping ι_{cfg}.

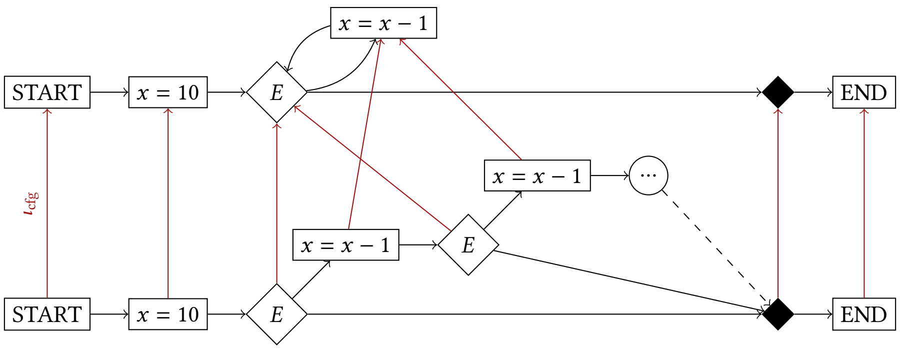


Fig. 2. CFG and unrolled CFG for program `(x = 10; while (x < 10) do (x = x - 1))` and their connection via the map ι_{cfg} shown with red edges.

The following lemma states that we can over-approximate the provenance set for variable x at node M in the unrolled CFG by applying Algorithm 1 to the corresponding node $\iota_{\text{cfg}}(M)$ in the standard CFG.

Lemma 3. *Let G be the CFG and H the unrolled CFG of program S. For each node $M \in H$ following statements hold:*

$$\{\iota_{\text{cfg}}(M') \colon M' \in \text{RD}(M, x)\} \subseteq \text{RD}(\iota_{\text{cfg}}(M), x) \tag{3}$$

$$\{\iota_{\text{cfg}}(M') \colon M' \in \text{BP}(M)\} = \text{BP}(\iota_{\text{cfg}}(M)) \tag{4}$$

$$\text{prov}(M, x) \subseteq \text{prov}(\iota_{\text{cfg}}(M), x) \tag{5}$$

Proof. This follows from the fact that the standard CFG and unrolled CFG of while loops produce paths that are compatible regarding reaching definitions and branch parents. The former produces paths of form START, $B, \overrightarrow{S_1}, B, \overrightarrow{S_2}, \dots, ..., \overrightarrow{S_n}, B, J,$ END, while the latter produces paths of form START, $B_1, \overrightarrow{S_1}, B_2, \overrightarrow{S_2}, \dots, ..., \overrightarrow{S_n}, B_{n+1}, J,$ END, where $\overrightarrow{S_i}$ are paths in the sub-CFGs. The full proof is given in Appendix B.4. □

4.2 Static Factorisation Theorem for Programs With Loops

The unrolled CFG has the important property that it is a directed acyclic graph with a potentially infinite number of nodes, but with a single root node. Further, each node $N \neq$ START still has exactly one predecessor except for join nodes. Therefore, Proposition 2 can be generalised to unrolled CFGs with only a slight modification to the proof.

Proposition 4. *For each node M in the* unrolled *CFG H and variable x, there exists an evaluation function*

$$V_x^M \in [\mathcal{T}|_{\text{prov}(M,x)} \to \mathcal{V}],$$

such that for all traces $\texttt{tr}$ *with* $\sigma_0 = (x_i \mapsto \texttt{null}, \boldsymbol{p} \mapsto 1)$ *and execution sequence*

$$(\sigma_0, START) \xrightarrow{\texttt{tr}} \cdots \xrightarrow{\texttt{tr}} (\sigma_i, M_i) \xrightarrow{\texttt{tr}} \cdots \xrightarrow{\texttt{tr}} (\sigma_l, END)$$

we have

$$\sigma_i(x) = V_x^{M_i}(\texttt{tr}).$$

Proof. The unrolled CFG H is constructed in a way such that we can prove the statement by well-founded induction on the relation $M_1 \prec M_2 \Leftrightarrow M_1$ *is parent of* M_2 (i.e. there is an edge from M_1 to M_2 in H) similar to the proof of Proposition 2. The full proof is given in Appendix B.5. □

As before, Proposition 4 allows us to explicitly construct the factorisation of the program density.

Theorem 2. *Let G be the CFG for a program S. Let $N_1, \dots, N_K$ be all sample nodes in G. For each sample node* $N_k = \text{Assign}(x_k = \texttt{sample}(E_0^k, f^k(E_1^k, \dots, E_{n_k}^k)))$, *let*

$$A_k = \text{addresses}(N_k) \cup \textstyle\bigcup_{i=0}^{n_k} \text{prov}(N_k, E_i^k) \cup \bigcup_{N' \in \text{BP}(N_k)} \text{prov}(N', E_{N'}).$$

Then, there exist functions $p_k \in [\mathcal{T}|_{A_k} \to \mathbb{R}_{\geq 0}]$ *such that if* $p_S(\texttt{tr}) \neq$ undefined, *then*

$$p_S(\texttt{tr}) = p_G(\texttt{tr}) = \prod_{k=1}^{K} p_k(\texttt{tr}).$$

Proof. We give a proof sketch and refer to Appendix B.6 for the full proof. For each sample node M_j in the *unrolled* CFG H, we construct $\tilde{p}_j \in [\mathcal{T}|_{\tilde{A}_j} \to \mathbb{R}_{\geq 0}]$ as in the proof of Theorem 1. Next, we group the nodes M_j by their corresponding CFG node:

$$p_k = \prod_{j\colon \iota_{\text{cfg}}(M_j) = N_k} \tilde{p}_j$$

From Lemma 3, we have that if $\iota_{\text{cfg}}(M_j) = N_k$, then $\tilde{A}_j = A_k$. Thus, $p_k \in [\mathcal{T}|_{A_k} \to \mathbb{R}_{\geq 0}]$ □

4.2.1 Equivalence to Markov Networks. As in Section 3.3.2, Theorem 2 implies that a program S with K sample statements is equivalent to a Markov network $\mathcal{H}$ with nodes $\{X_\alpha \colon \alpha \in \bigcup_{k=1}^K A_k\}$ even if it contains loops. Again, the sets $D_k = \{X_\alpha \colon \alpha \in A_k\}$ form cliques in $\mathcal{H}$ and p_S factorises over $\mathcal{H}$ with $\phi_{D_k}(D_k) = p_k(\{\alpha_j \mapsto X_j \colon \alpha_j \in A_k\})$:

$$P(\vec{X}) = \prod_{k=1}^{K} \phi_{D_k}(D_k).$$

5 Case Study: Speeding-up Posterior Inference

While the statically obtained factorisation is provably correct, it is natural to ask whether the quality of the factorisation is good enough to be of practical use. To answer this question, we have implemented a program transformation based on the static factorisation that generates sub-programs for each factor (described in Section 5.1). This program transformation operates on probabilistic programs that are implemented in our research PPL which models the formal language introduced in Section 2. In Section 5.2, we demonstrate that if the change to the current program trace is small, then we can leverage the sub-programs to compute the density of the updated trace faster. The employed optimisation trick is well-known but typically restricted to programs with a finite number of random variables and PPLs that require the user to explicitly construct the graphical structure. In contrast, our transformation operates statically on the source code and results in improved computation time even if a program contains while loops. We evaluate our approach on 15 probabilistic programs, the majority of which make use of while loops to declare an unbounded number of random variables. The implementation and the benchmark programs can be found in the openly-available replication package [Anonymous 2024].

5.1 Generating Sub-programs from the Static Factorisation

We generate a sub-program for each sample statement (and thus for each factor) by deriving a program slicing method from the static provenance analysis. For sample node N_k, we want to generate a sub-program that

- continues execution directly from N_k,
- stops execution once all sample nodes that depend on N_k have been reached,
- accumulates the density for all dependencies and for N_k itself.

The first requirement is achieved by storing not only the value of a random variable in the trace when executing a sample statement, but also storing the state of the program σ — the values of all program variables. For the other two requirements, we need to collect all sample nodes that depend on N_k. Recall that a sample node N_j depends on N_k if the factor $p_j \in [\mathcal{T}|_{A_j} \to \mathbb{R}_{\geq 0}]$ depends on addresses $\alpha \in \text{addresses}(N_k) \subseteq A_j$. As explained in Section 3.2, these dependencies can be found by modifying Algorithm 1 to collect the sample nodes instead of their addresses.

```
          Original program                        Sub-program for factor
----------------------------------------------------------------------------------
# Program 1                              # factor for "B"
A = sample("A", Normal(0.0, 1.0))        A = σ("A")
B = sample("B", Normal(A, 1.0))          B = score("B", Normal(A, 1.0))
C = sample("C", Normal(A, 1.0))          C = read_trace("C")
D = sample("D", Normal(B+C, 1.0))        D = score("D", Normal(B+C, 1.0))
E = sample("E", Normal(A, 1.0))
----------------------------------------------------------------------------------
# Program 2                              # factor for "b"
i = 0; b = 1;                            i = σ("i"); b = σ("b");
while b do                               b = score("b"+str(i),Bernoulli(0.5))
  b = sample("b"+str(i),Bernoulli(0.5))  i = i + 1
  i = i + 1                              while b do
                                           b = score("b"+str(i),Bernoulli(0.5))
                                           i = i + 1
----------------------------------------------------------------------------------
# Program 3                              # factor for "z"
i = 0;                                   i = σ("i");
while i < N do                           z = score("z"+str(i),Bernoulli(0.5))
  z = sample("z"+str(i),Bernoulli(0.5))  m = (z == 1) ? -2.0 : 2.0
  m = (z == 1) ? -2.0 : 2.0              x = score("x"+str(i),Normal(m,1.0))
  x = sample("x"+str(i),Normal(m,1.0))
  i = i + 1
----------------------------------------------------------------------------------
```

Fig. 3. Example sub-programs generated from the static factorisation alongside the original program.

To generate the sub-programs, we perform a type of reachability analysis to find all CFG nodes that lie on a path from N_k to N_j and slice the program according to this set of nodes. We will illustrate this process with three examples and refer to our implementation for further details [Anonymous 2024]. Consider program 1 in Figure 3. To generate a sub-program for the factor of sample statement "B", we omit the sample statements "A" and "E" as they do not lie on a path from "B" to "D" — the only sample statement that depends on "B". However, we still need to read the value of program variable `A` from the cached program state σ and we read the value for random variable "C" from the trace. Continuing executing directly from a sample statement by caching the program state means that we can continue a loop from a specific iteration as exemplified by program 2 in Figure 3. This is particularly powerful, if the only dependency of a sample statement belongs to the same loop iteration. In this case, an entire while loop can be replaced by a single iteration which is shown for program 3 in Figure 3.

5.2 Speeding-up Density Computation for Single-Site Trace Updates

Many posterior inference algorithms can be improved by exploiting the factorisation of the model density. For instance, in sequential Monte Carlo methods probabilistic programs are executed one factor at a time, after which resampling is performed to discard low quality samples [Meent et al. 2015]. Furthermore, variational inference methods may use the factorisation to reduce the variance in gradient estimation [Ranganath et al. 2014]. However, in this section we focus on the single-site Metropolis Hastings inference (MH) algorithm [Wingate et al. 2011]. The single-site MH algorithm works by selecting a single address α in each iteration at random and proposing a new value v. Even though we select a single address for the update, the proposal may have to produce additional values for sample statements that were not executed for trace $\texttt{tr}$, but due to stochastic branching or dynamic addresses are now executed for the new trace $\texttt{tr}'$. For instance in Figure 3 program 2, if we change the value of the last random variable "b_i" from 0 to 1, then we may have to sample values for multiple "b_j" where $j > i$ until we get $\texttt{tr}'(\text{"b\_}j\text{"}) = 0$.

The proposed trace $\texttt{tr}'$ is then accepted with probability $A = \min\left(1, \frac{p(\texttt{tr}')\times Q_\alpha(\texttt{tr})}{p(\texttt{tr})\times Q_\alpha(\texttt{tr}')}\right)$, where Q_α captures the probability of the proposal process (sampling a value for α and all additional values as explained above). If the address comes from sample node N_k, then we can optimise the runtime of this algorithm by noting that $\log p(\texttt{tr}') - \log p(\texttt{tr}) = \log p_k(\texttt{tr}') - \log p_k(\texttt{tr})$. Thus, by being able to compute the factors $p_k(\texttt{tr}')$ and $p_k(\texttt{tr})$ separately, we can more efficiently compute the ratio of densities $p(\texttt{tr}')/p(\texttt{tr})$.

As mentioned earlier, this optimisation trick is well-known and easily implemented for PPLs that require the user to explicitly construct the probabilistic model graphically with one node per random variable. By generating a sub-program to compute p_k we enable this optimisation for programs with an unbounded number of random variables and dynamic dependencies.

5.3 Experiment Results

To evaluate the speed-up made possible by our static factorisation, we gathered a benchmark set of 15 probabilistic programs. We specifically sampled probabilistic programs that make use of while loops to define an unbounded number of random variables. For comparison, we also included models that define only a finite number of random variables with fixed dependency structure For these programs, we implemented a factorisation that is representative of PPLs that explicitly construct a probabilistic graphical model. For each model, we generated N traces from the prior and randomly selected an address α for each trace $\texttt{tr}$. Then, we perform a resample update for this address to get a new trace $\texttt{tr}'$. In this update, we sample a new value for α from the prior and also sample values for any new sample statements that are encountered due to stochastic branching, as

Table 1. Average runtime of computing $p(\texttt{tr}')$ for traces $\texttt{tr}'$ that are generated by resampling traces $\texttt{tr}$ at a single site in microseconds (see Section 5.2). The column "None" reports the runtime of computing $p(\texttt{tr}')$ from scratch. The column "Static Analysis" reports the runtime of updating the old density by computing $\log p_k(\texttt{tr}') - \log p_k(\texttt{tr})$ with the generated sub-programs (see Section 5.1). The column "Fixed-Finite" reports the runtime of updating the density with a hand-written factorisation that is representative of PPLs that explicitly construct a probabilistic graphical model. The time in relation to computing from scratch is given in parenthesis. Lastly, the number of random variables defined in each model can be found in column "# RVs".

		Factorisation		
Model	**# RVs**	**None**	**Static Analysis**	**Fixed-Finite**
Aircraft	∞	8.303	4.217 (0.51)	-
Captcha	∞	948.582	335.915 (0.35)	-
Dirichlet process	∞	70.388	35.223 (0.50)	-
Geometric	∞	0.665	0.531 (0.80)	-
GMM (fixed number of clusters)	209	45.703	3.589 (0.08)	3.061 (0.07)
GMM (variable number of clusters)	∞	40.798	5.008 (0.12)	-
HMM (fixed sequence length)	21	4.777	4.714 (0.99)	1.837 (0.38)
HMM (variable sequence length)	∞	6.600	6.110 (0.93)	-
Hurricane	5	0.893	0.677 (0.76)	-
LDA (fixed number of topics)	551	165.376	25.385 (0.15)	12.503 (0.08)
LDA (variable number of topics)	∞	155.461	15.390 (0.10)	-
Linear regression	102	3.634	6.949 (1.91)	6.811 (1.87)
Marsaglia	∞	0.495	1.199 (2.42)	-
Pedestrian	∞	4.532	3.986 (0.88)	-
Urn	∞	8.945	5.961 (0.67)	-

explained in Section 5.2. We measure the runtime of computing $\log p_k(\texttt{tr}') - \log p_k(\texttt{tr})$ with the generated sub-programs and compare it against computing $p(\texttt{tr}')$ from scratch. In Table 1, you can see the results of the experiment run on a M2 Pro CPU. We averaged the measured times over all runs N. Note that we selected N for each model such that the benchmark time is approximately 1-2 seconds. N is greater than 10000 except for the Captcha model where $N = 1000$.

The first observation that we want to highlight is that computing $\log p_k(\texttt{tr}') - \log p_k(\texttt{tr})$ can lead to longer runtime if there is no structure to exploit in the model, $p_k \approx p$. Consider the linear regression model in which the 100 observations depend on the slope and intercept variables. When changing the value of one of these variables, we have to subtract the log-probability of observing the 100 data points according to the old value, $\log p_k(\texttt{tr})$, and add the log-probability of observing the data points according to the new value $\log p_k(\texttt{tr}')$. This effectively doubles the runtime, because when computing $p(\texttt{tr}')$ from scratch, we only have to add up the log-probability according to the new value. The same is true for the Marsaglia model which implements a rejection sampling method for the Normal distribution such that all variables depend on each other.

The factorisation approach shines when there is a lot of structure in the models like for the Aircraft, Captcha, and Dirichlet Process models, where the runtime is more than halved. It is particularly powerful, when there are latent variables that only affect independent data variables. This is the case for the Gaussian mixture model (GMM) where there is one cluster allocation variable per data point and for the Latent Dirichlet Allocation model (LDA) where there is similar dependency structure. For these models we observe 6-12x speed-ups. These kind of speed-ups are also achievable with existing methods provided that we only have a finite number of random variables. With our proposed method, we can transfer these speed-ups to the case where the GMM has an unknown number of clusters and to the case where the LDA has an unknown number of topics. In both of these cases the models instantiate an unbounded number of random variables. Note that the handwritten "fixed-finite" factorisation of the LDA model is twice as fast as the "static analysis" factorisation, because it was optimised to do less read-from-trace operations rather than leveraging the dependency structure more effectively. Also note that there are model specific improvements for the factorisation of GMM and LDA to only update the log-probability of data points that belong to a certain cluster / allocation which further reduces the runtime to 2.41 and 2.46 microseconds, respectively. However, these improvements are beyond the scope of automated static methods.

The only model where our approach fails to effectively leverage the full dependency structure is the Hidden Markov Model (HMM). If the sequence length is fixed then the current state only affects the next state. Our static analysis over-approximates this structure and outputs that the current state affects all future states. This is the reason why the "fixed-finite" factorisation outperforms the static one. Now, if we modify the program by unrolling the while loop, i.e. repeating its body ten times, our factorisation approach results in an improved runtime of 2.261 μs, because in this case the dependency structure is not over-approximated anymore. This suggests that our analysis may be improved by detecting loops for which the number of iterations is fixed and statically identifiable. Note that if the sequence length is variable and a transition to a terminal state is possible at any point, indeed the current state affects all future states. In this case, our static analysis is exact, but no improvements in runtime are measurable.

Lastly, we want to note that the Hurricane model is the only model with a finite number of random variables but a dynamic dependency structure (see Section 3.3.2). Thus, for this model existing approaches are also not applicable, but our method can reduce the runtime by over 20%.

6 Related Work

6.1 Semantics of Probabilistic Programming Languages

One of the first works to formally define the semantics of a probabilistic programming language was Kozen [1979]. Derivatives of this semantics can be found throughout literature, for example. see Hur et al. [2015], Dahlqvist and Kozen [2019], or Barthe et al. [2020]. We give a brief (and incomplete) overview of existing semantic models for PPLs.

In recent years, a lambda calculus for probabilistic programming was presented by Borgström et al. [2016]. This work and a domain theoretic approach for probabilistic programming [Vákár et al. 2019] had an influence on the development of the statistical PCF [Mak et al. 2021] (programming computable functions), which was used to prove differentiability of terminating probabilistic programs. Probabilistic programming was also examined from a categorical perspective [Heunen et al. 2017] and the semantics of higher order probabilistic programs is also researched [Dahlqvist and Kozen 2019; Staton et al. 2016].

As already mentioned, the semantics presented in this work generalise those of Gorinova et al. [2019] which formalises the core language constructs of Stan by interpreting a program as a function over a fixed set of finite variables. In our semantics, we interpret programs as functions over traces which are mappings from addresses to values similar to dictionaries. Another dictionary based approach to probabilistic programming can be found in Cusumano-Towner et al. [2020] and denotational semantics based on trace types are introduced in Lew et al. [2019].

To the best of our knowledge there are only two existing semantic models that support user-defined addresses [Lee et al. 2019; Lew et al. 2023]. These approaches are denotational and sampling-based in contrast to our density-based operational semantics.

6.2 Graphical Representations and Dependency Analysis

The advantages of representing a probabilistic model graphically are well-understood [Koller and Friedman 2009]. For this reason, some PPLs like PyMC [Salvatier et al. 2016], Factorie [McCallum et al. 2009], or Infer.NET [Minka 2012] require the user to explicitly define their model as a graph. This is not possible if we want to support while loops in our programs. Modifying the Metropolis Hastings algorithm to exploit the structure of the model is also common practice, e.g. see Nori et al. [2014]; van de Meent et al. [2018]; Wu et al. [2016].

The connection of simple probabilistic programs to Bayesian networks is known [van de Meent et al. 2018]. Borgström et al. [2011] consider a small imperative language and give semantics in terms of factor graphs – a concept related to Bayesian networks. Sampson et al. [2014] compile probabilistic programs to Bayesian networks by treating loops as black-box functions in order to verify probabilistic assertion statements. Previous work extended Bayesian networks to so-called contingent Bayesian networks in order to support models with an infinite number of random variables or cyclic dependencies [Milch et al. 2005], but they cannot be straightforwardly applied to PPLs with user-labelled sample statements and while loops.

The probabilistic dependency structure of programs has also been exploited in previous work. Baah et al. [2008] use it for fault diagnosis, Hur et al. [2014] designed an algorithm for slicing probabilistic programs, and Bernstein et al. [2020] extract the factor graph of programs to automate model checking for models implemented in Stan. Castellan and Paquet [2019] enable incremental computation in the Metropolis Hastings algorithm by graphically modelling the data-dependencies between sample statements of first-order probabilistic programs.

In contrast to the methods above, in this work we statically determine a density factorisation for programs even if they contain while loops. This factorisation implies an equivalence to Markov networks and enables new opportunities for optimising inference. These opportunities are new, because in universal probabilistic programming the common approach is to track dependencies dynamically for optimised inference [Chen et al. 2014; Wu et al. 2016; Yang et al. 2014].

Lastly, it is worth mentioning that static analysis for probabilistic programming has also been explored for applications different from optimising inference. For instance, Lee et al. [2019] developed a static analysis to verify so-called guide programs for stochastic variational inference in Pyro. Another example is the extension of the Stan compiler with a pedantic mode that statically catches common pitfalls in probabilistic programming [Bernstein 2023]. For a a more complete survey on static analysis for probabilistic programming see Bernstein [2019].

7 Conclusion

In this work, we introduced address-based operational semantics for a formal probabilistic programming language to address the open question to what extent a probabilistic program with user-labelled sample statements can be represented graphically. We presented a sound static analysis that approximates the dependency structure of random variables in the program and used this structure to factorise the implicitly defined program density. The factorisation allowed us to show equivalence of probabilistic programs to either Bayesian or Markov networks. Lastly, we demonstrated that we can speed-up a single-site Metropolis Hastings algorithm by generating a sub-program for each factor based on the dependency structure and leveraging them to optimise program density computation. This illustrates that our static analysis has the potential to be used to optimise inference algorithms in a way that is typically reserved for PPLs which require the explicit construction of a graphical model.

References

Anonymous. 2024. Static Factorisation of Probabilistic Programs With User-Labelled Sample Statements and While Loops. https://anonymous.4open.science/r/PPLStaticFactor/.

George K Baah, Andy Podgurski, and Mary Jean Harrold. 2008. The probabilistic program dependence graph and its application to fault diagnosis. In *Proceedings of the 2008 international symposium on Software testing and analysis*. 189–200.

Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva. 2020. *Foundations of probabilistic programming*. Cambridge University Press.

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

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

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.

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.

Johannes Borgström, Ugo Dal Lago, Andrew D Gordon, and Marcin Szymczak. 2016. A lambda-calculus foundation for universal probabilistic programming. *ACM SIGPLAN Notices* 51, 9 (2016), 33–46.

Johannes Borgström, Andrew D Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. 2011. Measure transformer semantics for Bayesian machine learning. In *Programming Languages and Systems: 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 20*. Springer, 77–96.

Simon Castellan and Hugo Paquet. 2019. Probabilistic programming inference via intensional semantics. In *Programming Languages and Systems: 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings 28*. Springer, 322–349.

Yutian Chen, Vikash Mansinghka, and Zoubin Ghahramani. 2014. Sublinear approximate inference for probabilistic programs. *stat* 1050 (2014), 6.

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

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).

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.

Fredrik Dahlqvist and Dexter Kozen. 2019. Semantics of higher-order probabilistic programs with conditioning. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–29.

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.

Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. 2012. Church: a language for generative models. *arXiv preprint arXiv:1206.3255* (2012).

Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. 2014. Probabilistic programming. In *Future of software engineering proceedings*. 167–181.

Maria I Gorinova, Andrew D Gordon, and Charles Sutton. 2019. Probabilistic programming with densities in SlicStan: efficient, flexible, and deterministic. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

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

Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability theory. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 1–12.

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.

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

Chung-Kil Hur, Aditya V Nori, Sriram K Rajamani, and Selva Samuel. 2015. A provably correct sampler for probabilistic programs. In *35th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

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

Dexter Kozen. 1979. Semantics of probabilistic programs. In *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*. IEEE, 101–114.

Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M Blei. 2017. Automatic differentiation variational inference. *Journal of machine learning research* (2017).

Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. 2019. Towards verified stochastic variational inference for probabilistic programs. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–33.

Alexander K Lew, Marco F Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K Mansinghka. 2019. Trace types and denotational semantics for sound programmable inference in probabilistic languages. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32.

Alexander K Lew, Matin Ghavamizadeh, Martin C Rinard, and Vikash K Mansinghka. 2023. Probabilistic Programming with Stochastic Probabilities. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1708–1732.

David J Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. 2000. WinBUGS-a Bayesian modelling framework: concepts, structure, and extensibility. *Statistics and computing* 10 (2000), 325–337.

Carol Mak, C-H Luke Ong, Hugo Paquet, and Dominik Wagner. 2021. Densities of almost surely terminating probabilistic programs are differentiable almost everywhere. In *Programming Languages and Systems: 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings 30.* Springer International Publishing, 432–461.

Andrew McCallum, Karl Schultz, and Sameer Singh. 2009. Factorie: Probabilistic programming via imperatively defined factor graphs. *Advances in Neural Information Processing Systems* 22 (2009).

Jan-Willem Meent, Hongseok Yang, Vikash Mansinghka, and Frank Wood. 2015. Particle Gibbs with ancestor sampling for probabilistic programs. In *Artificial Intelligence and Statistics.* PMLR, 986–994.

Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L Ong, and Andrey Kolobov. 2007. BLOG: Probabilistic models with unknown objects. (2007).

Brian Milch, Bhaskara Marthi, David Sontag, Stuart Russell, Daniel L Ong, and Andrey Kolobov. 2005. Approximate inference for infinite contingent Bayesian networks. In *International Workshop on Artificial Intelligence and Statistics.* PMLR, 238–245.

Tom Minka. 2012. Infer. NET 2.5. *http://research. microsoft. com/infernet* (2012).

Aditya Nori, Chung-Kil Hur, Sriram Rajamani, and Selva Samuel. 2014. R2: An efficient MCMC sampler for probabilistic programs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 28.

Rajesh Ranganath, Sean Gerrish, and David Blei. 2014. Black box variational inference. In *Artificial intelligence and statistics.* PMLR, 814–822.

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

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.

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.

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.

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An introduction to probabilistic programming. *arXiv preprint arXiv:1809.10756* (2018).

David Wingate, Andreas Stuhlmüller, and Noah Goodman. 2011. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics.* JMLR Workshop and Conference Proceedings, 770–778.

Yi Wu, Lei Li, Stuart Russell, and Rastislav Bodik. 2016. Swift: Compiled inference for probabilistic programming languages. *arXiv preprint arXiv:1606.09242* (2016).

Lingfeng Yang, Patrick Hanrahan, and Noah Goodman. 2014. Generating efficient MCMC kernels from probabilistic programs. In *Artificial Intelligence and Statistics.* PMLR, 1068–1076.

A Control-Flow Graph Construction

Definition 8. *A CFG is a tuple $(N_{\text{start}}, N_{\text{end}}, \mathcal{N}, \mathcal{E})$ comprised of start and end node, a set of branch, join, and assign nodes, $\mathcal{N}$, and a set of edges, $\mathcal{E} \subseteq \mathcal{N}^* \times \mathcal{N}^*$, where $\mathcal{N}^* = \mathcal{N} \cup \{N_{\text{start}}, N_{\text{end}}\}$.*

- The CFG of a skip-statement, $S = \texttt{skip}$, consists of only start and end node:

$$\begin{tikzcd} \boxed{\text{START}} \arrow[r] & \boxed{\text{END}} \end{tikzcd}$$

$G = \big(N_{\text{start}}, N_{\text{end}}, \emptyset, \{(N_{\text{start}}, N_{\text{end}})\}\big)$

- The CFG of an assignment, $x = E$, or sample statement $x = \texttt{sample}(E_0, f(E_1, \dots, E_n))$, is a sequence of start, assign, and end node:

$$\begin{tikzcd} \boxed{\text{START}} \arrow[r] & \boxed{x = E} \arrow[r] & \boxed{\text{END}} \end{tikzcd}$$

$$\begin{tikzcd} \boxed{\text{START}} \arrow[r] & \boxed{x = \texttt{sample}(E_0, f(E_1, \dots, E_n))} \arrow[r] & \boxed{\text{END}} \end{tikzcd}$$

$$G = \big(N_{\text{start}}, N_{\text{end}}, \{A\}, \{(N_{\text{start}}, A), (A, N_{\text{end}})\}\big),$$

where $A = \text{Assign}(x = \dots)$ is the corresponding assign node.

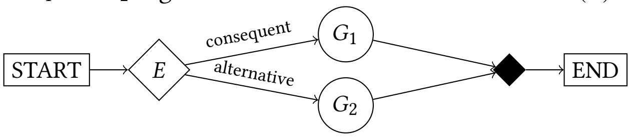
- The CFG for a sequence of statements, $S = S_1; S_2$, is recursively defined by "stitching together" the CFGs $G_1 = (N^1_{\text{start}}, N^1_{\text{end}}, \mathcal{N}^1, \mathcal{E}^1)$ and $G_2 = (N^2_{\text{start}}, N^2_{\text{end}}, \mathcal{N}^2, \mathcal{E}^2)$ of S_1 and S_2:

$$\begin{tikzcd} \boxed{\text{START}} \arrow[r] & G_1 \arrow[r] & G_2 \arrow[r] & \boxed{\text{END}} \end{tikzcd}$$

$$G = \big(N^1_{\text{start}},\ N^2_{\text{end}},\ \mathcal{N}^1 \cup \mathcal{N}^2,\ \mathcal{E}^1 \setminus \{(N^1_e, N^1_{\text{end}})\} \cup \mathcal{E}^2 \setminus \{(N^2_{\text{start}}, N^2_s)\} \cup \{(N^1_e, N^2_s)\}\big),$$

where N^1_e is the predecessor node of N^1_{end} in G_1, $(N^1_e, N^1_{\text{end}}) \in \mathcal{E}^1$, and N^2_s is the successor node of N^2_{start} in G_2, $(N^2_{\text{start}}, N^2_s) \in \mathcal{E}_2$.

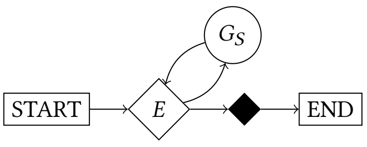
- The CFG of an if statement, $S = (\texttt{if } E \texttt{ then } S_1 \texttt{ else } S_2)$, is also defined in terms of the CFGs of S_1 and S_2 together with a branch node $B = \text{Branch}(E)$ and a join node J:



$$\begin{aligned} G = \big(&N_{\text{start}},\ N_{\text{end}},\ \mathcal{N}^1 \cup \mathcal{N}^2 \cup \{B, J\},\\ &\mathcal{E}^1 \setminus \{(N^1_{\text{start}}, N^1_s), (N^1_e, N^1_{\text{end}})\} \cup \mathcal{E}^2 \setminus \{(N^2_{\text{start}}, N^2_s), (N^2_e, N^2_{\text{end}})\} \cup\\ &\{(N_{\text{start}}, B), (B, N^1_s), (B, N^2_s), (N^1_e, J), (N^2_e, J), (J, N_{\text{end}})\}\big), \end{aligned}$$

The nodes N^1_s, N^1_e, N^2_s, N^2_e are defined as in the last case. This construction assumes that $\mathcal{E}^1 \neq \emptyset$ and $\mathcal{E}^2 \neq \emptyset$. Otherwise, the branch would be a `skip` statement and we would connect B with J directly. The two predecessors of J are denoted with $\text{predcon}(J)$ and $\text{predalt}(J)$ depending on the path.

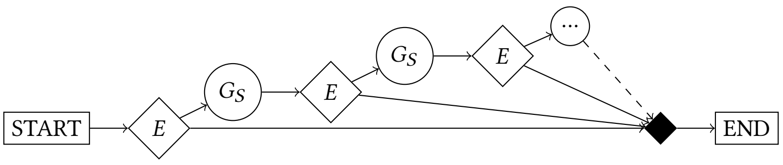
- The CFG of ($\texttt{while}$ E $\texttt{do}$ S) is given in terms of the sub-cfg G_S of statement S.



$$G = \big(N_{\text{start}},\ N_{\text{end}},\ \mathcal{N}^S \cup \{B, J\},$$
$$\mathcal{E}^S \setminus \{(N^S_{\text{start}}, N^S_s), (N^S_e, N^S_{\text{end}})\} \cup \{(N_{\text{start}}, B), (B, N^S_s), (B, J), (N^S_e, B), (J, N_{\text{end}})\}\big),$$

where $G_S = (N^S_{\text{start}}, N^S_{\text{end}}, \mathcal{N}^S, \mathcal{E}^S)$.
Again, if $S =$ `skip`, then we would have the edge (B, B) instead of (B, N^S_s) and (N^S_e, B).

- The unrolled CFG of (`while` E `do` S) is defined as follows:



$$G = \big(N_{\text{start}},\ N_{\text{end}},$$
$$\textstyle\bigcup_{i\in\mathbb{N}} \mathcal{N}^{S_i} \cup \{B_i : i \in \mathbb{N}\} \cup \{J\},$$
$$\textstyle\bigcup_{i\in\mathbb{N}} \mathcal{E}^{S_i} \setminus \{(N^{S_i}_{\text{start}}, N^{S_i}_s), (N^{S_i}_e, N^{S_i}_{\text{end}})\} \cup \{(N_{\text{start}}, B_1), (B, J), (J, N_{\text{end}})\} \cup$$
$$\{(B_i, N^{S_i}_s) : i \in \mathbb{N}\} \cup \{(N^{S_i}_e, B_{i+1}) : i \in \mathbb{N}\}\big),$$

where $G_{S_i} = (N^{S_i}_{\text{start}}, N^{S_i}_{\text{end}}, \mathcal{N}^{S_i}, \mathcal{E}^{S_i})$ is the i-th copy of sub-graph G_S. $N^{S_i}_s$ and $N^{S_i}_e$ are defined as in the cases before.
If $S =$ `skip`, then we would have the edges (B_i, B_{i+1}) instead of the edges $(B_i, N^{S_i}_s)$ and $(N^{S_i}_e, B_{i+1})$.

B Proofs

B.1 Proof of Proposition 1

Statement. *For all programs S with control-flow-graph G, it holds that for all traces* `tr`

$$p_S(\mathtt{tr}) = p_G(\mathtt{tr}).$$

Proof. With structural induction we prove the more general equivalence:

$$(\sigma, S) \Downarrow^{\mathtt{tr}} \sigma' \iff (\sigma, \text{START}) \xrightarrow[S]{\mathtt{tr}} \dots \xrightarrow[S]{\mathtt{tr}} (\sigma', \text{END}).$$

We write $\xrightarrow[S]{\mathtt{tr}}$ to indicate that the predecessor-successor relationship follow the CFG of program S.

For programs consisting of a single skip, assignment, or sample statement, the equality can be seen by directly comparing the operational semantic rules to the graph semantics.

For sequence $S = (S_1; S_2)$, by induction assumption we have

$$(\sigma, S_1) \Downarrow^{\mathtt{tr}} \sigma' \iff (\sigma, N^1_{\text{start}}) \xrightarrow[S_1]{\mathtt{tr}} \dots \xrightarrow[S_1]{\mathtt{tr}} (\sigma'_1, N^1) \xrightarrow[S_1]{\mathtt{tr}} (\sigma', N^1_{\text{end}})$$

$$(\sigma', S_2) \Downarrow^{\mathtt{tr}} \sigma'' \iff (\sigma', N^2_{\text{start}}) \xrightarrow[S_2]{\mathtt{tr}} (\sigma', N^2) \xrightarrow[S_2]{\mathtt{tr}} \dots \xrightarrow[S_2]{\mathtt{tr}} (\sigma'', N^2_{\text{end}})$$

By definition of the CFG of S, we see that

$$(\sigma, S) \Downarrow^{\mathtt{tr}} \sigma'' \iff (\sigma, N^1_{\text{start}}) \xrightarrow[S]{\mathtt{tr}} \dots \xrightarrow[S]{\mathtt{tr}} (\sigma'_1, N^1) \xrightarrow[S]{\mathtt{tr}} (\sigma', N^2) \xrightarrow[S]{\mathtt{tr}} \dots \xrightarrow[S]{\mathtt{tr}} (\sigma'', N^2_{\text{end}})$$

In a similar way, we proof the result for if statements $S = (\mathtt{if}\ E\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2)$ by considering the two cases $\sigma(E) = \mathtt{true}$ and $\sigma(E) = \mathtt{false}$.

$$(\sigma, S) \Downarrow^{\mathtt{tr}} \sigma' \iff (\sigma, N_{\text{start}}) \xrightarrow[S]{\mathtt{tr}} (\sigma, \text{Branch}(E)) \xrightarrow[S]{\mathtt{tr}} \dots \xrightarrow[S]{\mathtt{tr}} (\sigma', N_{\text{join}}) \xrightarrow[S]{\mathtt{tr}} (\sigma', N_{\text{end}})$$

Depending on the case, we replace the dots with the path of S_i without the start and end nodes. Note that in our notation in both cases $(\sigma, S_i) \Downarrow^{\mathtt{tr}} \sigma'$. □

B.2 Proof of Proposition 2

To prove soundness of Algorithm 1, we list properties of the provenance set that will become useful in subsequent proofs.

Lemma 4. *Let N be a node and x a variable.* prov *has following properties:*

- *For all nodes N' it holds that*
$$\text{RD}(N, x) = \text{RD}(N', x) \implies \text{prov}(N, x) = \text{prov}(N', x). \tag{6}$$
- *If $N' \in \text{RD}(N, x)$ is an assignment node, $N' = \text{Assign}(x = E)$, then*
$$\text{prov}(N', E) = \textstyle\bigcup_{y \in \text{vars}(E)} \text{prov}(N', y) \subseteq \text{prov}(N, x). \tag{7}$$
- *if $N' \in \text{RD}(N, x)$ is a sample node, $N' = \text{Assign}(x = \mathtt{sample}(E_0, \dots))$, then*
$$\text{prov}(N', E_0) \subseteq \text{prov}(N, x) \quad \textit{and} \quad \text{addresses}(N') \subseteq \text{prov}(N, x). \tag{8}$$
- *For all $N' \in \text{RD}(N, x)$ it holds that for all branch parents $N_{bp} = \text{Branch}(E_{N_{bp}}) \in \text{BP}(N')$ we have* $\text{prov}(N_{bp}, E_{N_{bp}}) \subseteq \text{prov}(N, x)$ *and*
$$\textstyle\bigcup_{N_{bp} \in \text{BP}(N')} \text{prov}(N_{bp}, E_{N_{bp}}) = \bigcup_{N_{bp} \in \text{BP}(N')} \bigcup_{y \in \text{vars}(E_{N_{bp}})} \text{prov}(N_{bp}, y) \subseteq \text{prov}(N, x). \tag{9}$$

Proof. If (N', y) is pushed into the queue, then $\text{prov}(N', y) \subseteq \text{prov}(N, x)$ as the algorithm for computing $\text{prov}(N', y)$ starts with only (N', y) in the queue. With this fact the properties follow directly from the definition of the algorithm. □

Statement. *For each node N in the CFG G and variable x, there exists an evaluation function*

$$V_x^N \in [\mathcal{T}|_{\text{prov}(N,x)} \to \mathcal{V}],$$

such that for all traces $\texttt{tr}$ *with* $\sigma_0 = (x_i \mapsto \texttt{null}, \boldsymbol{p} \mapsto 1)$ *and execution sequence*

$$(\sigma_0, START) \xrightarrow{\texttt{tr}} \cdots \xrightarrow{\texttt{tr}} (\sigma_i, N_i) \xrightarrow{\texttt{tr}} \cdots \xrightarrow{\texttt{tr}} (\sigma_l, END)$$

we have

$$\sigma_i(x) = V_x^{N_i}(\texttt{tr}).$$

Proof.

Step 1. Defining V_x^N and proving $V_x^N \in [\mathcal{T}|_{\text{prov}(N,x)} \to \mathcal{V}]$.

We begin by defining V_x^N inductively by noting that every node $N \neq$ START has exactly one predecessor except for join nodes which have two predecessors. Further, the CFG G is a finite directed acyclic graph with a single root node which makes mathematical induction possible.

Base case. For $((x_i \mapsto \texttt{null}, \mathbf{p} \mapsto 1), \text{START})$, let $V_{\mathbf{p}}^{\text{START}}(\texttt{tr}) = 1$ and $V_x^{\text{START}}(\texttt{tr}) = \texttt{null}$ for all variables x. We have $V_x^{\text{START}} \in [\mathcal{T}|_\emptyset \to \mathcal{V}]$.

Induction step.

Case 1. Let N be a node with single predecessor N' (N is not a join node). We define V_x^N based on the node type of N', for which we make the induction assumption that the evaluation function $V_x^{N'} \in [\mathcal{T}|_{\text{prov}(N',x)} \to \mathcal{V}]$ exists for all variables.

- **Case 1.1.** N' is an assignment node, $N' = \text{Assign}(z = E)$. Define

$$V_y^N(\texttt{tr}) = \begin{cases} V_E^{N'}(\texttt{tr}) & \text{if } y = z \\ V_y^{N'}(\texttt{tr}) & \text{otherwise.} \end{cases}$$

  By assumption $V_y^{N'} \in [\mathcal{T}|_{\text{prov}(N',y)} \to \mathcal{V}]$ for all y and thus, $V_E^{N'} \in [\mathcal{T}|_{\text{prov}(N',E)} \to \mathcal{V}]$.

  For $x \neq z$, $\text{RD}(N,x) = \text{RD}(N',x)$ and by Eq. (6) we have $\text{prov}(N,x) = \text{prov}(N',x)$, implying that $V_x^N \in [\mathcal{T}|_{\text{prov}(N,x)} \to \mathcal{V}]$.
  For $x = z$, $\text{RD}(N,x) = \{N'\}$ since N' is the single predecessor of N and assigns x. By Eq. (7) $\text{prov}(N',E) \subseteq \text{prov}(N,x)$, and by Eq. (2)

$$V_x^N = V_E^{N'} \in [\mathcal{T}|_{\text{prov}(N',E)} \to \mathcal{V}] \subseteq [\mathcal{T}|_{\text{prov}(N,x)} \to \mathcal{V}].$$

- **Case 1.2.** N' is a sample node, $N' = \text{Assign}(z = \texttt{sample}(E_0, f(E_1, \dots, E_n)))$. Define

$$V_y^N(\texttt{tr}) = \begin{cases} \texttt{tr}(V_{E_0}^{N'}(\texttt{tr})) & \text{if } y = z \\ V_y^{N'}(\texttt{tr}) & \text{otherwise.} \end{cases}$$

  As before, for $x \neq z$, $V_x^N \in [\mathcal{T}|_{\text{prov}(N,x)} \to \mathcal{V}]$.
  For $x = z$, by definition $V_{E_0}^{N'}(\texttt{tr}) \in \text{addresses}(N')$. Since $N' \in \text{RD}(N,x) = \{N'\}$ it follows from Eq. (8) that $\text{addresses}(N') \subseteq \text{prov}(N,x)$ and $\text{prov}(N',E_0) \subseteq \text{prov}(N,x)$. By Eq. (2)

$$V_x^N \in [\mathcal{T}|_{\text{addresses}(N') \cup \text{prov}(N',E_0)} \to \mathcal{V}] \subseteq [\mathcal{T}|_{\text{prov}(N,x)} \to \mathcal{V}].$$

- **Case 1.3.** N' is a branch or join node. Define $V_y^N = V_y^{N'}$ for all variables y.

Case 2. Let J be a join node with two predecessor nodes, $N_1' = \text{predcons}(J)$, $N_2' = \text{predalt}(J)$. Let $B = \text{Branch}(E)$ be the corresponding branching node, $(B,J) \in \text{BranchJoin}(G)$.

For variable x, there are two cases:

(1) There exists a reaching definition $N' \in \mathrm{RD}(J, x)$ on a path $(B, \dots, N', \dots, J)$. Define V_x^1 and V_x^2 depending on the node type of N_1' and N_2' as in step 1. The functions V_x^i compute the value of x after executing N_i'. As before one can see that $V_x^i \in [\mathcal{T}|_{\mathrm{prov}(J,x)} \to \mathcal{V}]$. Let

$$V_x^J(\texttt{tr}) := \mathrm{ife}(V_E^B(\texttt{tr}), V_x^1(\texttt{tr}), V_x^2(\texttt{tr})) = \begin{cases} V_x^1(\texttt{tr}) & \text{if } V_E^B(\texttt{tr}) = \texttt{true}, \\ V_x^2(\texttt{tr}) & \text{otherwise.} \end{cases}$$

Since $B \in \mathrm{BP}(N')$, by Eq. (9) $\mathrm{prov}(B, E) \subseteq \mathrm{prov}(J, x)$ and thus $V_x^J \in [\mathcal{T}|_{\mathrm{prov}(J,x)} \to \mathcal{V}]$.

(2) All reaching definitions of x (if there are any) are predecessors of B. Then, $\mathrm{RD}(J, x) = \mathrm{RD}(B, x)$ and by Eq. (6) $\mathrm{prov}(J, x) = \mathrm{prov}(B, x)$. Define $V_x^J := V_x^B$.

Step 2: Proving that $\sigma_i(x) = V_x^{N_i}(\texttt{tr})$.

Having defined the evaluation functions, we now have to prove that they indeed compute the correct values. For trace tr let the corresponding execution sequence be

$$(\sigma_0, \mathrm{START}) \xrightarrow{\texttt{tr}} \cdots \xrightarrow{\texttt{tr}} (\sigma_i, N_i) \xrightarrow{\texttt{tr}} \cdots \xrightarrow{\texttt{tr}} (\sigma_l, \mathrm{END}).$$

We will prove that $\sigma_i(x) = V_x^{N_i}(\texttt{tr})$ by induction.

Base case. The base case immediately follows from the definition $\sigma_0(x) = V_x^{\mathrm{START}}(\texttt{tr})$.

Induction step. For transition $(\sigma_i, N_i) \xrightarrow{\texttt{tr}} (\sigma_{i+1}, N_{i+1})$ the assumption is that $\sigma_j(x) = V_x^{N_j}(\texttt{tr})$ holds for all variables x and $j \leq i$.

If N_{i+1} is not a join node, then $\sigma_{i+1}(x) = V_x^{N_{i+1}}(\texttt{tr})$ follows directly from the CFG semantics and definition of $V_x^{N_{i+1}}$ in case 1 of step 1.

Lastly, we consider the case when N_{i+1} is a join node with branching node $N_j = \mathrm{Branch}(E)$, $(N_j, N_{i+1}) \in \mathrm{BranchJoin}(G)$, for some $j \leq i$. Let $N_1' = \mathrm{predcons}(N_{i+1})$, $N_2' = \mathrm{predalt}(N_{i+1})$.

By the semantics of if statements, $V_E^B(\texttt{tr}) = \texttt{true}$ iff $N_i = N_1'$ and $V_E^B(\texttt{tr}) = \texttt{false}$ iff $N_i = N_2'$. If there is a node N' between N_j and N_{i+1} in the execution sequence that assigns x, $N' \in \mathrm{RD}(N_{i+1}, x)$, then by definitions of V_x^1, V_x^2, and $V_x^{N_{i+1}}$ we have

$$\sigma_{i+1}(x) = \mathrm{ife}(V_E^B(\texttt{tr}), V_x^1(\texttt{tr}), V_x^2(\texttt{tr})) = V_x^{N_{i+1}}(\texttt{tr}).$$

If there is no such N', then

$$\sigma_{i+1}(x) = \sigma_j(x) = V_x^{N_j}(\texttt{tr}) = V_x^{N_{i+1}}(\texttt{tr}).$$

□

B.3 Proof of Proposition 3

Statement. *For all program S with CFG G and unrolled CFG H, it holds that for all traces* tr

$$p_S(\texttt{tr}) = p_G(\texttt{tr}) = p_H(\texttt{tr}).$$

Proof. We continue the proof of Proposition 1 for a while statement. Note the well-known equivalence for while statements in terms of the operational semantics:

$$(\sigma, \texttt{while}\ E\ \texttt{do}\ S) \Downarrow^{\texttt{tr}} \sigma' \iff (\sigma, \texttt{if}\ E\ \texttt{then}\ (S;\ \texttt{while}\ E\ \texttt{do}\ S)\ \texttt{else skip}) \Downarrow^{\texttt{tr}} \sigma'$$

Thus, if the program terminates for tr, it is equivalent to $S; \dots; S$ (S repeated n times), where $\sigma_0 = \sigma$, $(\sigma_i, S) \Downarrow^{\texttt{tr}} \sigma_{i+1}$, $\sigma_n = \sigma'$, such that $\sigma_i(E) = \texttt{true}$ for $i < n$, $\sigma_n(E) = \texttt{false}$.

In this case, it can be shown that the (unrolled) CFG of $S; \dots; S$ is also equivalent to the (unrolled) CFG of the while statement.

If the program does not terminate, then $p_S(\texttt{tr}) = p_G(\texttt{tr}) = p_H(\texttt{tr}) = \text{undefined}$. □

B.4 Proof of Lemma 3

Statement. *Let G be the CFG and H the unrolled CFG of program S. For each node $M \in H$ following equation holds.*

$$\{\iota_{\text{cfg}}(M') \colon M' \in \text{RD}(M, x)\} \subseteq \text{RD}(\iota_{\text{cfg}}(M), x) \tag{10}$$

Proof. We will prove following Lemma by structural induction:

Lemma. *For every path $\overrightarrow{M} = (M', \ldots, M)$ in the unrolled CFG H, there is a path in the CFG G, $\overrightarrow{N} = (N', \ldots, N)$, such that*

$$\iota_{\text{cfg}}(M') = N', \quad \iota_{\text{cfg}}(M) = N,$$

$$\text{AssignNodes}(\overrightarrow{N}, x) = \left\{\iota_{\text{cfg}}(M_i) \colon M_i \in \text{AssignNodes}(\overrightarrow{M}, x)\right\} = \text{AssignNodes}_{\text{cfg}}(\overrightarrow{M}, x),$$

for all variables x, where $\text{AssignNodes}(\overrightarrow{N}, x)$ *is the set of all CFG nodes that assign x in path $\overrightarrow{N}$ and* $\text{AssignNodes}_{\text{cfg}}(\overrightarrow{M}, x)$ *is the set of all assign nodes in path $\overrightarrow{M}$ mapped to G with ι_{cfg}.*

With this lemma we can prove the statement:

Let $M' \in \text{RD}(M, x)$. Thus, there exists a path $\overrightarrow{M} = (M', \ldots, M)$, such that M' is the only assign node for x before M. By the lemma, there is a path in the CFG $\overrightarrow{N} = (N', \ldots, N)$, such that $\iota_{\text{cfg}}(M') = N'$ and $\iota_{\text{cfg}}(M) = N$.

Further, since $\text{AssignNodes}(\overrightarrow{N}, x) = \text{AssignNodes}_{\text{cfg}}(\overrightarrow{M}, x) \subseteq \{\iota_{\text{cfg}}(M'), \iota_{\text{cfg}}(M)\} = \{N', N\}$, the node N' is also the only node in the path $\overrightarrow{N}$ before N that assigns x.

Thus, $\iota_{\text{cfg}}(M') = N' \in \text{RD}(N, x) = \text{RD}(\iota_{\text{cfg}}(M), x)$.

Proof of Lemma. By structural induction.

For CFGs consisting only of a single assign node, the unrolled CFGs are identical and the statement follows.

For a sequence of two statements $(S_1; S_2)$, let G be the CFG with sub-graphs G_1 and G_2 for statements S_1 and S_2 respectively. Let H the unrolled CFG with sub-graphs H_1 and H_2. We consider the only interesting case where $M' \in H_1$ and $M \in H_2$:

$$\overrightarrow{M} = (\underbrace{M, \ldots, M_i}_{\in H_1}, \underbrace{M_{i+1}, \ldots, M}_{\in H_2}).$$

For sub-paths $\overrightarrow{M_1} \coloneqq (M, \ldots, M_i, \text{END})$ and $\overrightarrow{M_2} \coloneqq (\text{START}, M_{i+1}, \ldots, M)$ we apply the induction assumption to get two paths $\overrightarrow{N_1} = (N', \ldots, N_1, \text{END})$, $\overrightarrow{N_2} = (\text{START}, N_2, \ldots, N)$, with $\iota_{\text{cfg}}(M') = N'$ and $\iota_{\text{cfg}}(M) = N$. The corresponding CFG path in G is $\overrightarrow{N} \coloneqq (N', \ldots, N_1, N_2, \ldots, N)$ as

$$\begin{aligned}\text{AssignNodes}_{\text{cfg}}(\overrightarrow{M}, x) &= \text{AssignNodes}_{\text{cfg}}(\overrightarrow{M_1}, x) \cup \text{AssignNodes}_{\text{cfg}}(\overrightarrow{M_2}, x) \\ &= \text{AssignNodes}(\overrightarrow{N_1}, x) \cup \text{AssignNodes}(\overrightarrow{N_2}, x) = \text{AssignNodes}(\overrightarrow{N}, x).\end{aligned}$$

For an if statement, let G be the CFG with branch node $\tilde{B}$, join node $\tilde{J}$, and sub-graphs G_1, G_2. Let H be the unrolled CFG with corresponding B, J, H_1, and H_2. We only show the statement for the case

$$\overrightarrow{M} = (B, \underbrace{M', \ldots, M}_{\in H_i}, J).$$

Again, we apply the induction assumption to $\overrightarrow{M_i} \coloneqq (\text{START}, M', \ldots, M, \text{END})$ and get a path in G_i: $\overrightarrow{N_i} = (\text{START}, N', \ldots, N, \text{END})$. Since $\iota_{\text{cfg}}(B) = \tilde{B}$, $\iota_{\text{cfg}}(J) = \tilde{J}$, which are not assign nodes, and $\text{AssignNodes}(\overrightarrow{N_i}) = \text{AssignNodes}_{\text{cfg}}(\overrightarrow{M_i})$, the statement follows for the path $(\tilde{B}, N', \ldots, N, \tilde{J})$.

Lastly, for a while loop, let G be the CFG with branch node $\tilde{B}$, join node $\tilde{J}$, and sub-graph G_S. Let H be the unrolled CFG with B_i branch nodes, J join node, and H_i sub-graphs. For the sake of brevity we consider three cases:

- For $M' \in H_i$ and $M \in H_j$

$$\overrightarrow{M} = (\underbrace{M', \dots, M_i}_{\in H_i}, B_{i+1}, \dots, B_j, \underbrace{M_j, \dots, M}_{\in H_j})$$

  we apply the induction assumption to the sub-paths and construct the path

$$\overrightarrow{N} = (\underbrace{N', \dots, N_i}_{\in G_S}, \tilde{B}, \dots, \tilde{B}, \underbrace{N_j, \dots, N}_{\in G_S}))$$

  where $\tilde{B}$ and B_i are do not contribute to AssignNodes($\overrightarrow{N}$) or AssignNodes($\overrightarrow{M}$).
- For $M' \in H_i$ and J

$$\overrightarrow{M} = (\underbrace{M', \dots, M_i}_{\in H_i}, B_{i+1}, \dots, B_l, J)$$

  we construct the path

$$\overrightarrow{N} = (\underbrace{N', \dots, N_i}_{\in G_S}, \tilde{B}, \dots, \tilde{B}, \tilde{J})$$

- For the path $\overrightarrow{M} = (B_l, J)$ the path in the CFG is $\overrightarrow{N} = (\tilde{B}, \tilde{J})$.

□

Statement. *Let G be the CFG and H the unrolled CFG of program S. For each node $M \in H$ the following equation holds.*

$$\mathrm{BP}_{\mathrm{cfg}}(M) := \{\iota_{\mathrm{cfg}}(M') \colon M' \in \mathrm{BP}(M)\} = \mathrm{BP}(\iota_{\mathrm{cfg}}(M)) \tag{11}$$

Proof. We prove this statement by structural induction.

For CFGs only consisting of a single assign node, the unrolled CFGs are identical, there are no branch parents, and the statement follows.

The sequencing of two statements $(S_1; S_2)$ does not change the branch parents of any node.

For an if statement, let G be the CFG with branch node $\tilde{B}$, join node $\tilde{J}$, and sub-graphs G_1, G_2. Let H be the unrolled CFG with corresponding B, J, H_1, and H_2. Every node M in H_i has branch parents $(\mathrm{BP}(M) \cap H_i) \cup \{B\}$.

By induction assumption $\mathrm{BP}_{\mathrm{cfg}}(M) \cap G_i = \mathrm{BP}(\iota_{\mathrm{cfg}}(M)) \cap G_i$ and thus

$$\begin{aligned}\mathrm{BP}_{\mathrm{cfg}}(M) &= \left\{\iota_{\mathrm{cfg}}(M') \colon M' \in \mathrm{BP}(M) \cap H_i\right\} \cup \{\iota_{\mathrm{cfg}}(B)\} \\ &= \left(\mathrm{BP}(\iota_{\mathrm{cfg}}(M)) \cap G_i\right) \cup \{\tilde{B}\} = \mathrm{BP}(\iota_{\mathrm{cfg}}(M)).\end{aligned}$$

Nodes B, $\tilde{B}$, J, and $\tilde{J}$ do not have branch parents.

Lastly, for a while loop, let G be the CFG with branch node $\tilde{B}$, join node $\tilde{J}$, and sub-graph G_S. Let H be the unrolled CFG with B_i branch nodes, J join node, and H_i sub-graphs.

Every node M in H_i has branch parents $(\mathrm{BP}(M) \cap H_i) \cup \{B_1, \dots, B_i\}$.

The CFG node $\iota_{\mathrm{cfg}}(M) \in G_S$ has branch parents $\left(\mathrm{BP}(\iota_{\mathrm{cfg}}(M)) \cap G_S\right) \cup \{\tilde{B}\}$.

Since $\iota_{\mathrm{cfg}}(B_j) = \tilde{B}$ and by induction assumption $\mathrm{BP}_{\mathrm{cfg}}(M) \cap G_S = \mathrm{BP}(\iota_{\mathrm{cfg}}(M)) \cap G_S$, we get

$$\begin{aligned}\mathrm{BP}_{\mathrm{cfg}}(M) &= \left\{\iota_{\mathrm{cfg}}(M') \colon M' \in \mathrm{BP}(M) \cap H_i\right\} \cup \{\iota_{\mathrm{cfg}}(B_1), \ldots, \iota_{\mathrm{cfg}}(B_i)\}\\ &= \left(\mathrm{BP}(\iota_{\mathrm{cfg}}(M)) \cap G_S\right) \cup \{\tilde{B}\} = \mathrm{BP}(\iota_{\mathrm{cfg}}(M)).\end{aligned}$$

□

Statement. *Let G be the CFG and H the unrolled CFG of program S. For each node $M \in H$ the following equation holds.*

$$\mathrm{prov}(M, x) \subseteq \mathrm{prov}(\iota_{\mathrm{cfg}}(M), x) \tag{12}$$

Proof. Follows directly from the two equations $\{\iota_{\mathrm{cfg}}(M') \colon M' \in \mathrm{RD}(M, x)\} \subseteq \mathrm{RD}(\iota_{\mathrm{cfg}}(M), x)$ and $\{\iota_{\mathrm{cfg}}(M') \colon M' \in \mathrm{BP}(M)\} = \mathrm{BP}(\iota_{\mathrm{cfg}}(M))$, since $\mathrm{prov}(M, x)$ is defined in terms of RD and BP. □

B.5 Proof of Proposition 4

Statement. *For each node M in the* unrolled *CFG H and variable x, there exists an evaluation function*

$$V_x^M \in [\mathcal{T}|_{\mathrm{prov}(M,x)} \to \mathcal{V}],$$

such that for all traces $\texttt{tr}$ *with $\sigma_0 = (x_i \mapsto \texttt{null}, \boldsymbol{p} \mapsto 1)$ and execution sequence*

$$(\sigma_0, \mathit{START}) \xrightarrow{\texttt{tr}} \cdots \xrightarrow{\texttt{tr}} (\sigma_i, M_i) \xrightarrow{\texttt{tr}} \cdots \xrightarrow{\texttt{tr}} (\sigma_l, \mathit{END})$$

we have

$$\sigma_i(x) = V_x^{M_i}(\texttt{tr}).$$

Proof. First, we have to justify the use of mathematical induction to prove properties on a potentially infinite graph. For $M_1, M_2 \in H$ define the relation $M_1 \prec M_2 \Leftrightarrow M_1$ is parent of M_2 (i.e. there is an edge from M_1 to M_2 in H). By construction of H, there are no infinite descending chains, $M_i \in H$ for $i \in \mathbb{N}$ such that $M_{i+1} \prec M_i$. This makes $\prec$ a *well-founded* relation, i.e. every non-empty subset has a minimal element. As a result, we may apply induction with respect to this relation (as has been done in the proof of Proposition 2) for graphs with infinite nodes. This proof principle is called well-founded or Noetherian induction.

To complete the proof, we only need to consider one additional case:

Case 3. Let J be a join node of a while loop with the parents $B_i = \mathrm{Branch}(E)$, $i \in \mathbb{N}$. The function $V_x^{B_i} \in [\mathcal{T}|_{\mathrm{prov}(B_i,x)} \to \mathcal{V}]$ exists by induction assumption and computes the value of x *before* executing the while loop body for the i-th time. If there is a reaching definition $N' \in \mathrm{RD}(J, x)$ on a path $(B_i, \ldots, N', \ldots, J)$, define

$$V_x^J(\texttt{tr}) \coloneqq \begin{cases} V_x^{B_i}(\texttt{tr}) & \text{if } \exists i \in \mathbb{N} \colon \forall j < i \colon V_E^{B_j}(\texttt{tr}) = \texttt{true} \land V_E^{B_i}(\texttt{tr}) = \texttt{false}, \\ \texttt{null} & \text{otherwise.} \end{cases}$$

Further, as N' belongs to the sub-CFG of the while loop body, we have that for every B_i there is a $N'_i \in \mathrm{RD}(J, x)$ on the path from B_i to J. Since $B_i \in \mathrm{BP}(N')$, by Equation (9) $\mathrm{prov}(B_i, E) \subseteq \mathrm{prov}(J, x)$ and thus $V_x^J \in [\mathcal{T}|_{\mathrm{prov}(J,x)} \to \mathcal{V}]$. If there is no reaching definition in the while loop body sub-CFG, then $\mathrm{prov}(J, x) = \mathrm{prov}(B_1, x)$ and define $V_x^J = V_x^{B_1}$. □

B.6 Proof of Theorem 2

Statement. *Let G be the CFG for a program S. Let $N_1, \dots, N_K$ be all sample nodes in G. For each sample node N_k = Assign(x_k = `sample`$(E_0^k, f^k(E_1^k, \dots, E_{n_k}^k))$), let*

$$A_k = \text{addresses}(N_k) \cup \textstyle\bigcup_{i=0}^{n_k} \text{prov}(N_k, E_i^k) \cup \bigcup_{N' \in \text{BP}(N_k)} \text{prov}(N', E_{N'}).$$

Then, there exist functions $p_k \in [\mathcal{T}|_{A_k} \to \mathbb{R}_{\geq 0}]$ such that if $p_S(\texttt{tr}) \neq$ undefined, then

$$p_S(\texttt{tr}) = p_G(\texttt{tr}) = \prod_{k=1}^{K} p_k(\texttt{tr}).$$

Proof. Let H be the *unrolled* CFG of program S. Denote the (countably many) sample nodes in H with M_j = Assign(x^j = `sample`$(E_0^j, f^j(E_1^j, \dots, E_{n_j}^j))$). Like in the proof of Theorem 1, define for each M_j, $b_j(\texttt{tr}) := \bigwedge_{M' \in \text{BP}(M_j)} t_{M'}^j(V_{E_{M'}}^{M'}(\texttt{tr}))$ where

$$b_j \in [\mathcal{T}|_{\bigcup_{M' \in \text{BP}(M_j)} \text{prov}(M', E_{M'})} \to \{\texttt{true}, \texttt{false}\}].$$

Again, $b_j(\texttt{tr}) = \texttt{true}$ if M_j is in the execution sequence for `tr` else `false`. Define

$$\tilde{p}_j(\texttt{tr}) := \delta_{b_j(\texttt{tr})} \text{pdf}_{f^j}\left(\texttt{tr}(V_{E_0^j}^{M_j}(\texttt{tr})); V_{E_1^j}^{M_j}(\texttt{tr}), \dots, V_{E_{n_j}^j}^{M_j}(\texttt{tr})\right) + (1 - \delta_{b_j(\texttt{tr})}).$$

We have $\tilde{p}_j \in [\mathcal{T}|_{\tilde{A}_j} \to \mathbb{R}_{\geq 0}]$, for

$$\tilde{A}_j = \text{addresses}(M_j) \cup \textstyle\bigcup_{i=0}^{n_j} \text{prov}(M_j, E_i^j) \cup \bigcup_{M' \in \text{BP}(M_j)} \text{prov}(M', E_{M'}).$$

Finally, we group the nodes M_j by their corresponding CFG node.

$$p_k = \prod_{j \colon \iota_{\text{cfg}}(M_j) = N_k} \tilde{p}_j$$

If $\iota_{\text{cfg}}(M_j) = N_k$, then $\tilde{A}_j = A_k$, since for $i = 0, \dots, n$ we have $E_i^k = E_i^j$ and by Eq. (5)

$$\text{prov}(M_j, E_i^j) \subseteq \text{prov}(\iota_{\text{cfg}}(M_j), E_i^j) = \text{prov}(N_k, E_i^k)$$

and by Eq. (4) it holds that

$$\{\iota_{\text{cfg}}(M') \colon M' \in \text{BP}(M_j)\} = \text{BP}(\iota_{\text{cfg}}(M_j)) = \text{BP}(N_k)$$

such that

$$\begin{aligned} \textstyle\bigcup_{M' \in \text{BP}(M_j)} \text{prov}(M', E_{M'}) &\subseteq \textstyle\bigcup_{M' \in \text{BP}(M_j)} \text{prov}(\iota_{\text{cfg}}(M'), E_{\iota_{\text{cfg}}(M')}) \\ &= \textstyle\bigcup_{N' \in \text{BP}(N_k)} \text{prov}(N', E_{N'}). \end{aligned}$$

Thus, for all j, $\iota_{\text{cfg}}(M_j) = N_k$, we have $\tilde{p}_j \in [\mathcal{T}|_{A_k} \to \mathbb{R}_{\geq 0}]$ which implies $p_k \in [\mathcal{T}|_{A_k} \to \mathbb{R}_{\geq 0}]$. □

C Towards a Measure-theoretic Interpretation of the Operational Semantics

In this section, we will briefly discuss how one may construct a measure space on traces such that the function $\texttt{tr} \mapsto p_S(\texttt{tr})$ is indeed a Radon-Nikodym derivative of a measure on this space. This construction is related to denotational semantics based on trace types [Lew et al. 2019, 2023]. We first define a measure space for each value type:

$$\tau ::= \{\texttt{null}\}, \mathbb{Z}, \mathbb{R}, \mathbb{R}^2, \dots$$
$$\mathcal{M}_\tau = (M_\tau, \Sigma_\tau, \nu_\tau) ::= (\{\texttt{null}\}, \mathscr{P}(\{\texttt{null}\}), \delta_{\texttt{null}}),\ (\mathbb{Z}, \mathscr{P}(\mathbb{Z}), \#),\ (\mathbb{R}, \mathcal{B}, \lambda),\ (\mathbb{R}^2, \mathcal{B}^2, \lambda^2),\ \dots$$

These measure spaces model the support of common distributions, but could also be extended by building sum and product spaces. They are comprised of the typical Dirac measure, power sets, counting measure, Borel sets, and Lebesgue measures.

Each trace $\texttt{tr}$ can be assigned an unique type function $\tau_{\texttt{tr}} \colon \textbf{Strings} \to \textbf{Types}$ such that for all $\alpha \in \textbf{Strings}$ we have $\texttt{tr}(\alpha) \in \tau_{\texttt{tr}}(\alpha)$. The types are assumed to be disjoint (e.g. $\mathbb{Z} \cap \mathbb{R} = \emptyset$). Let

$$\mathcal{S}_{<\infty} = \{s \colon \textbf{Strings} \to \textbf{Types} \colon |\{\alpha \in \textbf{Strings} \colon s(\alpha) \neq \{\texttt{null}\}\}| < \infty\}$$

be the set of all type functions that correspond to traces with a finite number of non-$\texttt{null}$ values. We enumerate the addresses, $\textbf{Strings} = \{\alpha_i \colon i \in \mathbb{N}\}$, and define a measureable space for each type function, $s \colon \textbf{Strings} \to \textbf{Types}$, $\mathscr{M}_s = (M_s, \Sigma_s) = \bigotimes_{i \in \mathbb{N} \colon s(\alpha_i) \neq \{\texttt{null}\}} \mathcal{M}_{s(\alpha_i)}$. If $s \in \mathcal{S}_{<\infty}$, then $\mathscr{M}_s$ is a finite dimensional product space with product measure ν_s. For the set of traces $\mathcal{T}$, we define the σ-algebra Σ with $A \in \Sigma \Leftrightarrow \forall s \colon \textbf{Strings} \to \textbf{Types} \colon \pi_s(\{\texttt{tr} \in A \colon \tau_{\texttt{tr}} = s\}) \in \Sigma_s$, where $\pi_s \colon \mathcal{T} \to M_s$ is the canonical projection. The reference measure on $(\mathcal{T}, \Sigma)$ is

$$\nu(A) = \sum_{s \in \mathcal{S}_{<\infty}} \nu_s(\pi_s(\{\texttt{tr} \in A \colon \tau_{\texttt{tr}} = s\}))$$

which effectively is a measure of traces with a finite number of non-$\texttt{null}$ values.

If a primitive distribution f with $\text{support}(f) \subseteq \tau_f$ and arguments $v_1 \dots, v_n$ corresponds to the measure $\mu_{f, v_1, \dots, v_n}$ with Radon-Nikodym derivative $\mathsf{d}\mu_{f, v_1, \dots, v_n} / \mathsf{d}\nu_{\tau_f}$, we define

$$\text{pdf}_f(v; v_1, \dots, v_n) = \begin{cases} (\mathsf{d}\mu_{f, v_1, \dots, v_n} / \mathsf{d}\nu_{\tau_f})(v) & \text{if } v \in \tau_f \\ 0 & \text{otherwise.} \end{cases}$$

The measure-theoretic interpretation of a program is the measure $\mu_S(A) = \int_{\mathcal{T}_{\text{valid}} \cap A} p_S \, \mathsf{d}\nu$, where

$$\mathcal{T}_{\text{valid}} := \{\texttt{tr} \colon p_S(\texttt{tr}) \neq \text{undefined} \wedge \forall \alpha \colon \texttt{tr}(\alpha) \neq \texttt{null} \Rightarrow p_S(\texttt{tr}[\alpha \mapsto \texttt{null}]) = \text{undefined}\}$$

is the set of all traces with well-defined density such that changing a value at any address to $\texttt{null}$ leads to an undefined density. That is, for $\texttt{tr} \in \mathcal{T}_{\text{valid}}$ all addresses that are not used during the execution of the program are set to $\texttt{null}$ and all address that are used are set to non-$\texttt{null}$ values. Such a trace necessarily has a finite number of non-$\texttt{null}$ values as the density for non-termination is undefined which is the only possibility for executing an infinite number of sample statements. This makes p_S the Radon-Nikodym derivative of μ_S with respect to ν on $\mathcal{T}_{\text{valid}}$.

We emphasize that this interpretation demands a careful analysis of the measurability of $\mathcal{T}_{\text{valid}}$ and p_S which is beyond the scope of this work. This requires additional assumptions about the program, at the very least that the primitives g are measurable. Such measurability analysis may be done along the lines of Borgström et al. [2016] and Lee et al. [2019].

Note that this construction can handle programs where random variables are of mixed-type like the variable y in the example below:

```
x = sample("x", Bernoulli(p))
if x == 1 then
    y = sample("y", Bernoulli(0.5))
else
    y = sample("y", Normal(0.,1.))
```

We list the density according to the semantics for different example trace of different type:

$$\begin{aligned}
p(\{x \mapsto 1\colon \mathbb{Z}, y \mapsto 1\colon \mathbb{Z}\}) &= \mathrm{pdf}_{\mathrm{Bernoulli}}(1; 0.5) \cdot \mathrm{pdf}_{\mathrm{Bernoulli}}(1; 0.5) = 0.5 \cdot 0.5 \\
p(\{x \mapsto 1\colon \mathbb{Z}, y \mapsto 1.0\colon \mathbb{R}\}) &= \mathrm{pdf}_{\mathrm{Bernoulli}}(1; 0.5) \cdot 0 = 0 \\
p(\{x \mapsto 0\colon \mathbb{Z}, y \mapsto 1.0\colon \mathbb{R}\}) &= \mathrm{pdf}_{\mathrm{Bernoulli}}(0; 0.5) \cdot \mathrm{pdf}_{\mathrm{Normal}}(1.0; 0.0, 1.0) \approx 0.5 \cdot 0.24197 \\
p(\{x \mapsto 0\colon \mathbb{Z}, y \mapsto 1\colon \mathbb{Z}\}) &= \mathrm{pdf}_{\mathrm{Bernoulli}}(0; 0.5) \cdot 0 = 0
\end{aligned}$$

C.1 Conditioning

Observed data is also modelled as a trace `obs_tr`, where addresses are mapped to the observed data points. All addresses that do not correspond to observed data map to `null`. The function $\texttt{tr} \mapsto p_S(\texttt{tr} \oplus \texttt{obs\_tr})$ then corresponds to the *unnormalised* posterior density, where we merge the trace `tr`, encapsulating latent variables, with the observed trace `obs_tr` like below

$$(A \oplus B)(\alpha) = \begin{cases} B(\alpha) & \text{if } B(\alpha) \neq \texttt{null}, \\ A(\alpha) & \text{otherwise.} \end{cases}$$