WIP: Address-Based Operational Semantics and Static Factorisation Analysis For Probabilistic Programs

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

While probabilistic programming has been studied from a mathematical perspective for over 40 years, its formalistations often differ from the systems used in practice. In this work, we introduce address-based operational semantics for a formal probabilistic programming language (PPL). Our semantics more closely model the inner workings of PPLs which make use of user-defined sample addresses like Gen, Pyro, or PyMC. Furthermore, by translating a program to its control-flow-graph, we define a sound static analysis that approximates the dependency structure of random variables in the program. As a result, we obtain a factorization of the implicitly defined program density, which allows us to show the equivalence of programs to either Bayesian or Markov networks. Lastly, we demonstrate how this statically obtained dependency structure can be used to speed up posterior inference for probabilistic programs in a case study.

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. WIP: Address-Based Operational Semantics and Static Factorisation Analysis For Probabilistic Programs. In . ACM, New York, NY, USA, 29 pages. https://doi.org/XXXXXXX

1 INTRODUCTION

Probabilistic programming provides an intuitive means to specify probabilistic models as programs. Typically, probabilistic programming systems enable automatic posterior inference for these programs. Besides these practical aspects, probabilistic programming has been examined from a mathematical perspective for over 40 years [Barthe et al. 2020; Kozen 1979]. However, there often is a gap between the formalisation of probabilistic programs and their practical implementation. Stan [Carpenter et al. 2017] is one of the few probabilistic programming languages (PPLs) that is formalised and widely used in real-world applications [Gorinova et al. 2019].

In this work, we present novel 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], Pyro [Bingham et al. 2019], PyMC [Salvatier et al. 2016], or BeanMachine [Tehrani et al. 2020]. 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 = pymc.Normal(f"x_{i}", mu=0., sigma=1.) # PyMC
x = sample("x_"+str(i), Normal(0.,1.))      # our formal syntax
```


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

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

Addresses are dynamically computed identifiers for random variables, which can be different from the name of the program variable they are assigned to.

We will interpret a probabilistic program as a function p, called program density, that maps a trace $\mathtt{tr}$ to its (unnormalised) density $\in \mathbb{R}_{\geq 0}$. 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 are 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:

$$p(\mathtt{tr}) = \prod_{k=1}^{K} p_k(\mathtt{tr}).$$

For each factor p_k, we can *statically* determine the set of addresses that contribute to it. If the program contains no loops and uses unique constant addresses for each sample statement, then the factorisation theorem implies that the program is equivalent to a Bayesian network. If the program contains loops, has dynamically computed addresses, or has non-unique addresses, then the program density still factorises into K terms. However, only an equivalence to the more general Markov networks can be shown. The reason is that we have to consider a potentially unbounded number of random variables and cyclic dependencies. The connection of probabilistic programs to Bayesian networks is well-known and often stated informally [CITE]. In contrast, this work constructs a factorisation statically from source code, formally correct with respect to the operational semantics of the program, which is non-trivial even in the case when programs describe models with an unbounded number of random variables.

1.1 Overview

Section 2. Address-Based Semantics For Probabilistic Programs introduces our novel 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 a program as a function over only a fixed set of finite variables, we extended this approach to interpret programs as functions over traces. Gorinova et al. injected the values of the random variables directly into the initial program state, whereas in our semantics the elements of the trace are used depending on dynamically evaluated addresses. Furthermore, our formal PPL and semantics support while loops.

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 at which addresses a trace 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.

We formally describe the well-known correspondence between a program and its CFG. 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. The factorisation theorems follow almost directly from the soundness of the provenance analysis.

In Section 3.3, we prove the factorisation theorem for programs without loops by explicitly constructing the factors from the evaluation functions. In this section, we also discuss the equivalence of programs to either Bayesian or Markov networks.

Section 4. Static Factorisation for Programs With Loops presents a clever trick to generalise the factorisation result of Section 3 to programs that may contain while loops. The idea is to mathematically unroll while loops to obtain a directed acyclic graph with a countably infinite number of nodes that essentially represent all possible program paths. This *unrolled CFG* preserves the program semantics and allows us to effectively reuse the proofs of Section 3. Again, with the resulting factorisation, equivalence of programs to Markov networks can be shown.

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 even if a program contains while loops.

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. 2014].

Syntax of Expressions:

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

Syntax of Statements:

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

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. However, for our purposes it is not necessary to characterise this set in detail and we define $\mathcal{V}$ to be the set of all values. A program contains a finite set of program variables denoted by x or x_i. 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 $E_1, \ldots, E_n$.

2.1 Operational Semantics

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

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

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

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

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

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

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

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

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

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 interpreted inductively and the semantics of a while loop can be rewritten as

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

where $\text{repeat}_n(S) = (S; \ldots; S)$ repeated n-times. In particular, if a while loop does not terminate then $\nexists \sigma' \colon (\sigma, \text{while } E \text{ do } S) \Downarrow^{\mathtt{tr}} \sigma'$. This makes the presented operational semantics partial, where errors and non-termination are modelled implicitly.

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 $\mathtt{tr}(V_0)$. Note that the function pdf_f may be an arbitrary function to real values. However, in the context of probabilistic programming it can be interpreted as the density function of distribution f. Lastly, the value $\mathtt{tr}(V_0)$ is stored in the program state at variable x. The semantics of sample statements are summarised in the following rule:

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

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 meaning of a program S is a function $p_S \colon \mathcal{T} \to \mathbb{R}_{\geq 0}$ given by*

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

To model non-termination explicitly, we would need to introduce special looping states σ_∞ and add following rules to the operational semantics:

$$\frac{\forall n \in \mathbb{N}_0 \colon (\sigma, \text{repeat}_n(S)) \Downarrow^{\mathtt{tr}} \sigma'_n \wedge \sigma'_n(E) = \mathtt{true}}{(\sigma, \text{while } E \text{ do } S) \Downarrow^{\mathtt{tr}} \sigma_\infty(\mathbf{p} \mapsto \lim_{n \to \infty} \sigma'_n(\mathbf{p}))} \qquad \frac{}{(\sigma_\infty, S) \Downarrow^{\mathtt{tr}} \sigma_\infty} \tag{1}$$

This would lead to slight modifications of the subsequent proofs in form of more case distinctions and is omitted in the remainder of this work for the sake of brevity.

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"; p = 0.25;
else
    addr = "z"; p = 0.75;
r = sample(addr, p)
```

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 `true` and 0 otherwise.

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

As the addresses in Listing 1 are static and identical to the program variables, the program is well-described by existing PPL semantics. However, in Listing 2 we rewrite the program with *dynamic addresses* which results in the same above density. Most existing PPL semantics can at best interpret the program as a three dimensional probabilistic model [CITE], whereas our semantics correctly constructs a four dimensional density. Our interpretation is closer to the implementations of sample-based PPLs that lazily evaluate if statements like Gen or Pyro.

Listing 3. Program with duplicate address.

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

Listing 4. 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))
```

Those PPL semantics that handle dynamic addresses typically assume that each address appears only once in the execution of a program [Cusumano-Towner et al. 2020; Lew et al. 2019] [TODO]. 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 probabilistic model, it can in fact be translated to density-based PPLs like Stan or Turing. Lastly, the presented semantics correctly describe programs with while loops. Such programs are significantly harder to describe, because while loops easily lead to an unbounded number of addresses as can be seen in Listing 4.

2.2 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

$\texttt{tr} \mapsto p(\texttt{tr} \oplus \texttt{obs\_tr})$ then corresponds to the *unnormalised* posterior density, where we merge the trace $\texttt{tr}$, encapsulating latent variables, with the observed trace $\texttt{obs\_tr}$ where

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

2.3 Measure-theoretic interpretation

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:

$$\tau ::= \{\texttt{null}\}, \mathbb{Z}, \mathbb{R}, \mathbb{R}^2, \dots$$

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

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 $\tau_{\texttt{tr}} : \textbf{Strings} \to \textbf{Types}$ 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

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

be the set of all type functions that correspond to traces with a finite number of non-$\texttt{null}$ values. We enumerate the addresses, $\textbf{Strings} = \{\alpha_i : i \in \mathbb{N}\}$, and define a measureable space for each type function, $s : \textbf{Strings} \to \textbf{Types}$, $\mathcal{M}_s = (M_s, \Sigma_s) = \bigotimes_{i \in \mathbb{N} :\, s(\alpha_i) \neq \{\texttt{null}\}} \mathcal{M}_{s(\alpha_i)}$. If $s \in \mathcal{S}_{<\infty}$, then $\mathcal{M}_s$ is a finite product space with product measure ν_s. For the set of traces $\mathcal{T}$, we define the σ-algebra Σ with $A \in \Sigma \Leftrightarrow \forall s : \textbf{Strings} \to \textbf{Types} : \pi_s(\{\texttt{tr} \in A : \tau_{\texttt{tr}} = s\}) \in \Sigma_s$, where $\pi_s : \mathcal{T} \to M_s$ is the canonical projection. The reference measure on $(\mathcal{T}, \Sigma)$ is $\nu(A) = \sum_{s \in \mathcal{S}_{<\infty}} \nu_s(\pi_s(\{\texttt{tr} \in A : \tau_{\texttt{tr}} = s\}))$. Loosely, traces with an infinite number of non-$\texttt{null}$ values, can only corresponds to non-terminating program runs, which we do not measure in this brief construction.

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

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

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

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

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 $\texttt{null}$. This makes p_S the Radon-Nikodym derivative of μ 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.

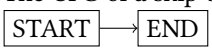
3 STATIC FACTORISATION FOR PROGRAMS WITHOUT LOOPS

3.1 Control-Flow-Graph

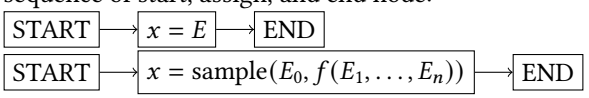
The goal of this work is to statically find a factorisation of the density implicitly defined by a probabilistic program in terms of addresses. This static analysis will operate on the control-flow-graph (CFG) of a program. We begin by considering programs without loops first and describe how we can 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.

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. 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 both mathematically and as diagrams. Sub-graphs are drawn by circular nodes, branch and join nodes are drawn as diamond nodes, and assign, start, and end nodes have rectangular shape. In text, branch nodes are written as $\text{Branch}(E)$. Assign nodes are denoted with $\text{Assign}(x = E)$ and $\text{Assign}(x = \text{sample}(E_0, \dots))$.

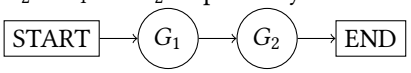
- The CFG of a skip-statement, $S = \text{skip}$, consists of only start and end node.

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

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

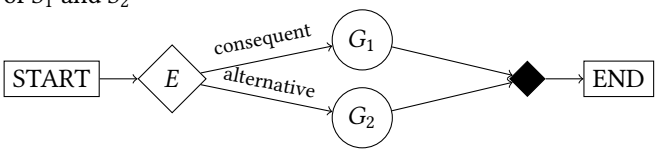
  

  The resulting CFG can be written as $G = (G_1 \cup G_2) \setminus \{N^1_{\text{end}}, N^2_{\text{start}}\}$, with

$$N_{\text{start}} = N^1_{\text{start}}, \quad N_{\text{end}} = N^2_{\text{end}}, \quad \text{predecessor}(N^1_{\text{end}}) \to \text{successor}(N^2_{\text{start}}),$$

  where N^i_{start} is the start node of G_i and N^i_{end} is the end node of G_i.

- The CFG of an if statement, $S = (\text{ if } E \text{ then } S_1 \text{ else } S_2)$, is also defined in terms of the CFGs of S_1 and S_2

  The resulting CFG can be written as

$$G = G_1 \setminus \{N^1_{\text{start}}, N^1_{\text{end}}\} \cup G_2 \setminus \{N^2_{\text{start}}, N^2_{\text{end}}\} \cup \{N_{\text{start}}, N_{\text{end}}, N_{\text{branch}}, N_{\text{join}}\}, \text{ where}$$

$$N_{\text{branch}} = \text{Branch}(E), \quad N_{\text{start}} \to N_{\text{branch}}, \quad N_{\text{join}} \to N_{\text{end}},$$
$$N_{\text{branch}} \to \text{successor}(N^1_{\text{start}}) =: \text{consequent}(N_{\text{branch}}),$$
$$N_{\text{branch}} \to \text{successor}(N^2_{\text{start}}) =: \text{alternative}(N_{\text{branch}}),$$
$$\text{predcons}(N_{\text{join}}) := \text{predecessor}(N^1_{\text{end}}) \to N_{\text{join}},$$
$$\text{predalt}(N_{\text{join}}) := \text{predecessor}(N^2_{\text{end}}) \to N_{\text{join}}.$$

Above, we have also introduced some additional labels for the successors of branch nodes and predecessors of join nodes. Furthermore, we say the tuple $(N_{\text{branch}}, N_{\text{join}})$ is a brain-join pair and define $\text{BranchJoin}(G)$ 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:

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

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

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

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 semantics of a CFG G is the function $p_G \colon \mathcal{T} \to \mathbb{R}_{\geq 0}$ given by*[1]

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

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. See Appendix A.1. □

[1] If we want to model non-termination explicitly, we need to add the case $p_G(\texttt{tr}) = \lim_{i \to \infty} \sigma_i(\mathbf{p})$ if $\forall \in \mathbb{N} : \exists \sigma_i, N_i :$ $N_i \neq \text{END} \land (\sigma_i, N_i) \xrightarrow{\texttt{tr}} (\sigma_{i+1}, N_{i+1})$, compare (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, which 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, i.e., 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* $\text{AssignNodes}(G, x)$ *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 are 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*

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

A branch node B and join node J are a branch-join pair $(B, J) \in \text{BranchJoin}(G)$ 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 of N 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 $N = \text{Assign}(x = \textit{sample}(E_0, f(E_1, \dots, E_n)))$, 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 $\text{prov}(N, x)$ in Algorithm 1. This algorithm is inspired by standard methods for dependence analysis [CITE?]. In the algorithm we use $\text{vars}(E)$, 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

$$\text{prov}(N, E) = \bigcup_{y \in \text{vars}(E)} \text{prov}(N, y). \tag{2}$$

The algorithm works as follows. If we want to find $\text{prov}(N, x)$, we first have to find all reaching definitions for x. If the reaching definition N' is a sample node, $x = \text{sample}(E_0, \dots)$, then the value of x can only dependent on the addresses generated by E_0, $\text{addresses}(N')$, 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 $\text{addresses}(N)$ 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.

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

Algorithm 1 Computing the provenance set $\text{prov}(N, x)$ statically from the CFG.

Input CFG node N, variable x
prov $\leftarrow$ EmptySet()
queue $\leftarrow [(N, x)]$
while queue.is_not_empty() **do**
  $(N, x) \leftarrow$ queue.pop()
  for $N' \in \text{RD}(N, x)$ **do**
    if $N' = \text{Assign}(x = \text{sample}(E_0, f(E_1, \ldots, E_n)))$ **then**
      prov $\leftarrow$ prov $\cup$ addresses(N')
      $E' \leftarrow E_0$ ▷ E_0 comes from pattern-matching N'
    else if $N' = \text{Assign}(x = E)$ **then**
      $E' \leftarrow E$ ▷ E comes from pattern-matching N'
    for $y \in \text{vars}(E')$ **do**
      if is_unmarked((N', y)) **then**
        mark((N', y))
        queue.push((N', y))
    for $N_{\text{bp}} \in \text{BP}(N')$ **do** ▷ $N_{\text{bp}} = \text{Branch}(E_{N_{\text{bp}}})$
      for $y \in \text{vars}(E_{N_{\text{bp}}})$ **do**
        if is_unmarked((N_{bp}, y)) **then**
          mark((N_{bp}, y))
          queue.push((N_{bp}, y))
return prov

```
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))
```

$$N = \text{Assign}(\texttt{x} = \ldots)$$
$$V_{\texttt{s}}^{N}(\texttt{tr}) = \texttt{tr}(\text{"s"})$$
$$V_{\texttt{m}}^{N}(\texttt{tr}) = \delta_{\texttt{tr}(\text{"b"})} \cdot \texttt{tr}(\text{"mu"}) + (1 - \delta_{\texttt{tr}(\text{"b"})}) \cdot 1$$

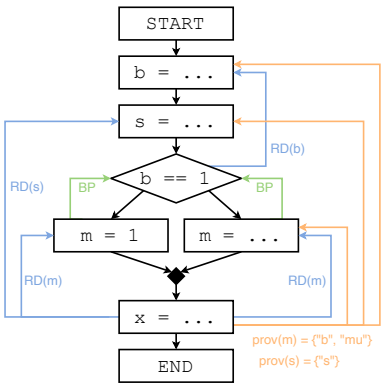


Fig. 1. Reaching definitions, branch parents, and the provenance set for a simple example. The evaluation functions introduced in Section 3.2.1 are also given for node N.

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 provenance calculation would be unchanged. This illustrates that the presented algorithm can only over-approximate the provenance and not compute it exactly.

Lastly, we list properties of the provenance set that will become useful in subsequent proofs.

Lemma 1. prov *has following properties:*

- *For all nodes N, N', and variable x it holds that*

$$\mathrm{RD}(N,x) = \mathrm{RD}(N',x) \implies \mathrm{prov}(N,x) = \mathrm{prov}(N',x). \tag{3}$$

- *If $N' \in \mathrm{RD}(N,x)$ is an assignment node, $N' = \mathrm{Assign}(x = E)$, then*

$$\mathrm{prov}(N',E) = \bigcup_{y \in \mathrm{vars}(E)} \mathrm{prov}(N',y) \subseteq \mathrm{prov}(N,x). \tag{4}$$

- *if $N' \in \mathrm{RD}(N,x)$ is a sample node, $N' = \mathrm{Assign}(x = \mathit{sample}(E_0, \dots))$, then*

$$\mathrm{prov}(N',E_0) \subseteq \mathrm{prov}(N,x) \quad \textit{and} \quad \mathrm{addresses}(N') \subseteq \mathrm{prov}(N,x). \tag{5}$$

- *For all $N' \in \mathrm{RD}(N,x)$ it holds that for all branch parents $N_{bp} = \mathrm{Branch}(E_{N_{bp}}) \in \mathrm{BP}(N')$ we have $\mathrm{prov}(N_{bp}, E_{N_{bp}}) \subseteq \mathrm{prov}(N,x)$ and*

$$\begin{aligned} &\textstyle\bigcup_{N_{bp} \in \mathrm{BP}(N')} \mathrm{prov}(N_{bp}, E_{N_{bp}}) = \\ &\textstyle\bigcup_{N_{bp} \in \mathrm{BP}(N')} \bigcup_{y \in \mathrm{vars}(E_{N_{bp}})} \mathrm{prov}(N_{bp}, y) \subseteq \mathrm{prov}(N,x). \end{aligned} \tag{6}$$

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

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 defined what we mean that a set of addresses contribute to the computation. Suppose that there is a function f that maps a trace $\mathtt{tr}$ to a value v. The set of addresses A contribute to the computation of value v, if any other trace $\mathtt{tr}'$ that has the same values as $\mathtt{tr}$ for all addresses $\alpha \in A$, gets mapped to the same value $f(\mathtt{tr}') = v$. In other words, if we change the trace $\mathtt{tr}$ at an address $\beta \in \mathbf{Strings} \setminus A$, the value of $f(\mathtt{tr})$ remains unchanged. Or equivalently, if $f(\mathtt{tr}) \neq f(\mathtt{tr}')$, then there must be an address $\alpha \in A$ such that $\mathtt{tr}(\alpha) \neq \mathtt{tr}'(\alpha)$.

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

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

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

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

Lemma 2. *If $A \subseteq A'$ and $f \in [\mathcal{T}|_A \to B]$, then $f \in [\mathcal{T}|_{A'} \to B]$. Thus*

$$A \subseteq A' \implies [\mathcal{T}|_A \to B] \subseteq [\mathcal{T}|_{A'} \to B]. \tag{7}$$

Proof. Let $f \in [\mathcal{T}|_A \to B]$. Assume $\forall \alpha \in A' : \mathtt{tr}(\alpha) = \mathtt{tr}'(\alpha)$. In particular, this holds for the smaller set A, $\forall \alpha \in A : \mathtt{tr}(\alpha) = \mathtt{tr}'(\alpha)$, which implies $f(\mathtt{tr}) = f(\mathtt{tr}')$. Thus, $f \in [\mathcal{T}|_{A'} \to B]$. □

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

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

Proof. Assume $\forall \alpha \in A_1 \cup A_2 : \mathtt{tr}(\alpha) = \mathtt{tr}'(\alpha)$. Thus, $f_i(\mathtt{tr}) = f_i(\mathtt{tr}')$ and $h(\mathtt{tr}) = h(\mathtt{tr}')$. □

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

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

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

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 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 functions $V_x^{N'} \in [\mathcal{T}|_{\text{prov}(N',x)} \to \mathcal{V}]$ exist for all variables.

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

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

  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 $y \neq x$, $\text{RD}(N, y) = \text{RD}(N', y)$ and by Eq. (3) we have $\text{prov}(N, y) = \text{prov}(N', y)$, implying that $V_y^N \in [\mathcal{T}|_{\text{prov}(N,y)} \to \mathcal{V}]$.
  For x, $\{N'\} = \text{RD}(N, x)$ since N' is the single predecessor of N and assigns x. By Eq. (4) $\text{prov}(N', E) \subseteq \text{prov}(N, x)$, and by Eq. (7)

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

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

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

  As before, for $x \neq y$, $V_y^N \in [\mathcal{T}|_{\text{prov}(N,y)} \to \mathcal{V}]$.
  For x, by definition $V_{E_0}^{N'}(\mathtt{tr}) \in \text{addresses}(N')$. Since $N' \in \text{RD}(N, x)$ it follows from Eq. (5) that $\text{addresses}(N') \subseteq \text{prov}(N, x)$ and by Eq. (5) $\text{prov}(N', E_0) \subseteq \text{prov}(N, x)$. Lastly, by Eq. (7)

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

- **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 each variable x, there are two cases:

(1) There exists a reaching definition $N' \in \text{RD}(J, x)$ on a path $(B, \ldots, N', \ldots, J)$. 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 $V_x^i \in [\mathcal{T}|_{\text{prov}(J,x)} \to \mathcal{V}]$. Let

$$V_x^J(\mathtt{tr}) := \delta_{V_E^B(\mathtt{tr})} V_x^1(\mathtt{tr}) + (1 - \delta_{V_E^B(\mathtt{tr})}) V_x^2(\mathtt{tr}).$$

  Since $B \in \text{BP}(N')$, by Eq. (6) $\text{prov}(B, E) \subseteq \text{prov}(J, x)$ and thus $V_x^J \in [\mathcal{T}|_{\text{prov}(J,x)} \to \mathcal{V}]$.

(2) All reaching definitions of x (if there are any) are predecessors of B. Then, $\text{RD}(J, x) = \text{RD}(B, x)$ and by Eq. (3) $\text{prov}(J, x) = \text{prov}(B, x)$. Define $V_x^J := V_x^B$.

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

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

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

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

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

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

If N_{i+1} is not a join node, then $\sigma_{i+1}(x) = V_x^{N_{i+1}}(\mathtt{tr})$ 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 $N_j = \text{Branch}(E)$, $(N_j, N_{i+1}) \in \text{BranchJoin}(G)$, for some $j \leq i$. Let $N_1' = \text{predcons}(N_{i+1})$, $N_2' = \text{predalt}(N_{i+1})$.

By the semantics of if statements, $V_E^B(\mathtt{tr}) = \mathtt{true}$ iff $N_i = N_1'$ and $V_E^B(\mathtt{tr}) = \mathtt{false}$ 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, $N' \in \text{RD}(N_{i+1}, x)$, then by definitions of V_x^1, V_x^2, and $V_x^{N_{i+1}}$ we have

$$\sigma_{i+1}(x) = \delta_{V_E^B(\mathtt{tr})} V_x^1(\mathtt{tr}) + (1 - \delta_{V_E^B(\mathtt{tr})}) V_x^2(\mathtt{tr}) = V_x^{N_{i+1}}(\mathtt{tr}).$$

If there is no such N', then

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

□

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 $N_k = \text{Assign}(x_k = sample(E_0^k, f^k(E_1^k, \dots, E_n^k)))$, let*

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

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 $b_k(\texttt{tr}) \coloneqq \prod_{N' \in \text{BP}(N_k)} V_{E_{N'}}^{N'}(\texttt{tr})$ where

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

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

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

By construction, $p_k \in [\mathcal{T}|_{A_k} \to \mathbb{R}_{\geq 0}]$. □

For the example program in Fig. 1, the density 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 $[\mathcal{T}|_{\{\text{"b"},\text{"s"}\}} \to \mathbb{R}_{\geq 0}]$ with $V_{\texttt{m}}^N(\texttt{tr}) = \delta_{\texttt{tr}(\text{"b"})} \cdot 1 + (1 - \delta_{\texttt{tr}(\text{"b"})}) \cdot 1 = 1$. Thus, 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, $N_k = \text{Assign}(x_k = \text{sample}(\alpha_k, f^k(E_1^k, \dots, E_n^k)))$, 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 $\alpha_k \in \textbf{Strings}$. 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

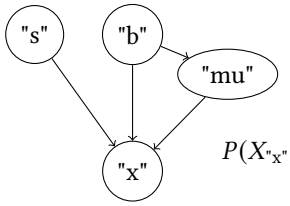
$$p_k(\texttt{tr}) = \delta_{b_k(\texttt{tr})} \text{pdf}_{f^k}\left(\texttt{tr}(\alpha_k); V_{E_1^k}^{N_k}(\texttt{tr}), \dots, V_{E_n^k}^{N_k}(\texttt{tr})\right) + (1 - \delta_{b_k(\texttt{tr})}) \mathbf{1}_{\texttt{null}}(\texttt{tr}(\alpha_k)),$$

where the function $\mathbf{1}_{\texttt{null}}(v)$ equals 1 if $v = \texttt{null}$ 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 $\mathbf{1}_{\texttt{null}}$ 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 $\alpha_j \to \alpha_k$ if $\alpha_j \in A_k \setminus \{\alpha_k\}$, such that $\text{parents}_{\mathcal{G}}(X_{\alpha_k})) = A_k \setminus \{\alpha_k\}$ and

$$P(X_{\alpha_k} \mid \text{parents}_{\mathcal{G}}(X_{\alpha_k})) = p_k(\{\alpha_j \mapsto X_j : \alpha_j \in A_k\}).$$

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 $X_{\texttt{"x"}}$ not only depends on $X_{\texttt{"mu"}}$, but also on $X_{\texttt{"b"}}$.



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

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 $\mathcal{H}$:

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

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, $D_k = \{X_\alpha : \alpha \in A_k\}$ forms a clique and p_S factorises over $\mathcal{H}$ with $\phi_{D_k}(D_k) = p_k(\{\alpha_j \mapsto X_j : \alpha_j \in A_k\})$.

Consider the program with constant but non-unique sample addresses in Listing 5. With nine sample statements we get a Markov network over $\{X_{\texttt{"F"}}, X_{\texttt{"P0"}}, X_{\texttt{"P1"}}, X_{\texttt{"D0"}}, X_{\texttt{"D1"}}\}$ shown in Fig. 2. 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 cyclic "P0" $\to$ "D0" $\to$ "P1" $\to$ "D1" $\to$ "P0", which means there is no Bayesian network representation for the program. If we rewrite the model with dynamic addresses shown in Listing 6, the red dashed edges are introduced in the Markov network in Fig. 2 and we have one factor for address set {"F"}, one for set {"F", "P0", "P1"}, and three factors for address set {"F", "P0", "P1", "D0", "D1"}.

Lastly, while the programs considered up until this point were equivalent to finite Bayesian networks or finite Markov networks, the small program in Listing 7 is equivalent to a Markov network with an infinite number of nodes.

Listing 7. Probabilistic program equivalent to a Markov network with an infinite number of variables.

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

Listing 5. 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(if prep_0 == 1 then 0.2 else 0.8))
    prep_1 = sample("P1", Bernoulli(if damage_0 == 1 then 0.75 else 0.5))
    damage_1 = sample("D1", Bernoulli(if prep_1 == 1 then 0.2 else 0.8))
else
    prep_1 = sample("P1", Bernoulli(0.5))
    damage_1 = sample("D1", Bernoulli(if prep_1 == 1 then 0.2 else 0.8))
    prep_0 = sample("P0", Bernoulli(if damage_1 == 1 then 0.75 else 0.5))
    damage_0 = sample("D0", Bernoulli(if prep_0 == 1 then 0.2 else 0.8))
```

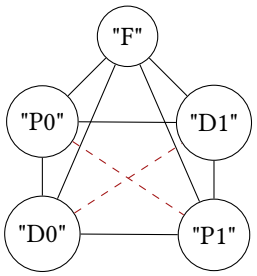


Fig. 2. Markov network of the hurricane model.

Listing 6. Second version of program in Listing 5 with dynamic addresses.

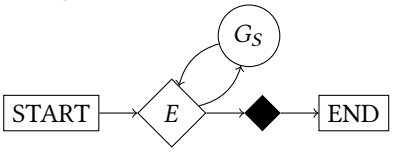
```
first_city_ixs = sample("F", Bernoulli(0.5))
first = str(first_city_ixs)
second = str(1 - first_city_ixs)

prep_first = sample("P"+first, Bernoulli(0.5))
damage_first = sample("D"+first,
    Bernoulli(if prep_first == 1 then 0.2 else 0.8))
prep_second = sample("P"+second,
    Bernoulli(if damage_first == 1 then 0.75 else 0.5))
damage_second = sample("D"+second,
    Bernoulli(if prep_second == 1 then 0.2 else 0.8))
```

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.

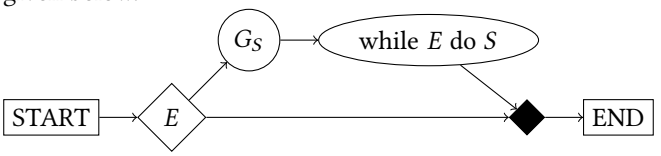
4.1 Unrolled Control-Flow-Graph

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, $\text{prov}(N, x)$ 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*. All translation rules are as before except for while loops, which is given below.



This is a recursive definition, which we will describe in more detail. The resulting graph G contains for every $i \in \mathbb{N}$ branch nodes B_i, join nodes J_i, and *copies* of the sub-CFG G_i of statement S. The edges are given by

- $\text{START} \to B_1$, $J_1 \to \text{END}$
- $B_i \to \text{successor}(\text{START}_{G_i})$, $\text{predecessor}(\text{END}_{G_i}) \to B_{i+1}$, $B_i \to J_i$, $J_{i+1} \to J_i$

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. Again, the semantics are equivalent.

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

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

Proof. See Appendix A.2. □

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 a 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. 3, where you can see the CFG and unrolled CFG of a program with the mapping ι_{cfg}.

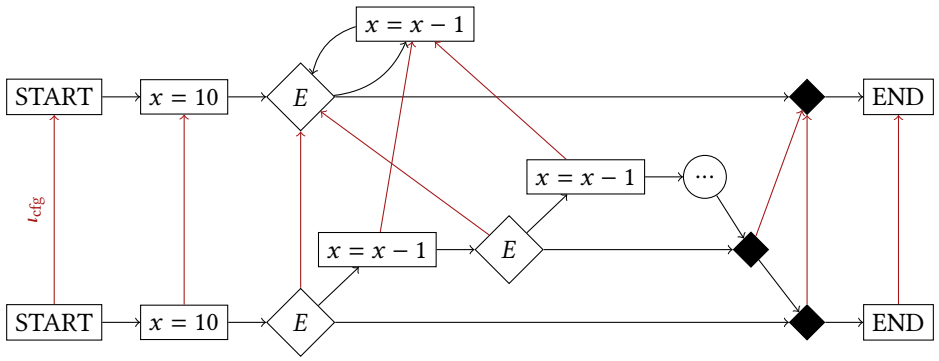


Fig. 3. CFG and unrolled CFG for program `(x = 10; while (x < 10) do (x = x - 1))` and their connection via the map ι_{cfg} shown with red edges.

The next 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 4. *Let G be the CFG and H the unrolled CFG of program S. For each node $M \in H$ following statements hold.*

$$\{\iota_{\text{cfg}}(M') \colon M' \in \text{RD}(M, x)\} \subseteq \text{RD}(\iota_{\text{cfg}}(M), x) \tag{8}$$

$$\{\iota_{\text{cfg}}(M') \colon M' \in \text{BP}(M)\} = \text{BP}(\iota_{\text{cfg}}(M)) \tag{9}$$

$$\text{prov}(M, x) \subseteq \text{prov}(\iota_{\text{cfg}}(M), x) \tag{10}$$

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, $B, \overrightarrow{S_1}, B, \overrightarrow{S_2}, \ldots, ..., \overrightarrow{S_n}, B, J$, END, while the latter produces paths of form START, $B_1, \overrightarrow{S_1}, B_2, \overrightarrow{S_2}, \ldots, ..., \overrightarrow{S_n}, B_{n+1}, J_{n+1}, J_n, \ldots, J_1$, END, where $\overrightarrow{S_i}$ are paths in the sub-CFGs. The full proof is given in Appendix A.3. □

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 which have two predecessors. Therefore, Proposition 2 can be generalised to unrolled CFGs without having to modify 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 $\mathtt{tr}$ *with* $\sigma_0 = (x_i \mapsto \mathtt{null}, \boldsymbol{p} \mapsto 1)$ *and execution sequence*

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

we have

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

As before, Proposition 4 allows us to explicitly construct the factorisation of the program density. However, since there is a potentially infinite number of sample nodes in the unrolled CFG, in the proof we will group their contribution to the density to get a factorisation over a finite number of terms. The factorisation theorem for programs with loops reads exactly the same as the theorem for programs without loops.

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

$$A_k = \text{addresses}(N_k) \cup \text{prov}(N_k, E_0^k) \cup \bigcup_{i=1}^n \text{prov}(N_k, E_i^k) \cup \bigcup_{N' \in \text{BP}(N_k)} \text{prov}(N', E_{N'}).$$

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

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

Proof. Let H be the *unrolled* CFG of program S and let $M_j = \text{Assign}(x^j = \text{sample}(E_0^j, f^j(E_1^j, \ldots, E_n^j)))$ be all sample nodes in H (countably many). Like in the proof of Theorem 1, define for each M_j, $b_j(\mathtt{tr}) := \prod_{M' \in \text{BP}(M_j)} V_{E_{M'}}^{M'}(\mathtt{tr})$ where

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

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

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

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

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

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 $i = 0, \ldots, n$ we have $E_i^k = E_i^j$ and by Eq. (10)

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

and by Eq. (9)

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

such that

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

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

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 $\phi_{D_k}(D_k) = p_k(\{\alpha_j \mapsto X_j\colon \alpha_j \in A_k\})$,

$$P(\vec{X}) = \prod_{D \in \text{cliques}(\mathcal{H})} \phi_D(D) = \prod_{k=1}^{K} \phi_{D_k}(D_k).$$

Consider the program in Listing 8. After starting at a random position in $[0, 3]$, the iteration takes random steps in either direction until travelling a distance of 10 units or until reaching the origin 0. In this process, an unbounded number of random variables may be instantiated. While the sample statement with address "step_t" has seemingly no dependencies, the provenance analysis tells us that it depends on all variables with a "step" address and the "start" variable through the while loop condition. This is an over-approximation since, in fact, the variable with address "step_t" depends only on the variables from previous iterations, i.e. variables with address "step_k" for $k < t$, and the "start" variable. Note that due to the fact that this program only contains three sample statements, the factorisation obtained from Theorem 2 is close to being trivial, i.e. one factor equal to the whole program density depending on all sample addresses. In the next section, we will see an example where the factorisation reveals non-trivial structure.

5 CASE STUDY: SPEEDING-UP POSTERIOR INFERENCE

In this section, we demonstrate that the density factorisation of Theorem 1 and Theorem 2 is non-trivial by showing how the statically inferred dependency structure can be used to speed-up posterior inference. 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 construct the graphical structure explicitly [CITE]. In contrast, our analysis operates statically on the source code and results in improved inference time even if a program contains while loops.

Consider the Gaussian mixture model in Listing 9, which we have implemented with *immutable* lists and while loops. The primitive function `append(a,v)` returns a *copy* of list `a` with value `v` added to the end. The function `get(a,i)` returns the `i`-th element of list `a`.

Listing 8. Pedestrian example adapted from [Mak et al. 2021].

```
start = sample("start", Uniform(0,3))
t = 0
position = start
distance = 0.0
while position > 0 and distance < 10:
    step = sample("step_" + str(t), Uniform(-1,1))
    distance = distance + abs(step)
    position = position + step
    t = t + 1
sample("final_distance", Normal(1.1, 0.1), obs=distance)
```

Listing 9. GMM adapted from [Matheos et al. 2020].

```
w = sample("w", Dirichlet(fill(δ,K)))
means = list(); vars = list();
k = 1
while k <= K
  mu = sample("mu_"+str(k),
    Normal(ξ,1/sqrt(κ))
  var = sample("var_"+str(k),
    InverseGamma(α,β))
  means = append(means,mu)
  vars = append(vars,var)
  k = k + 1
i = 1
while i <= N
  z = sample("z_"+str(i), Categorical(w))
  y = sample("y_"+str(i),
   Normal(get(means,z),sqrt(get(vars,z))))
  i = i + 1
```

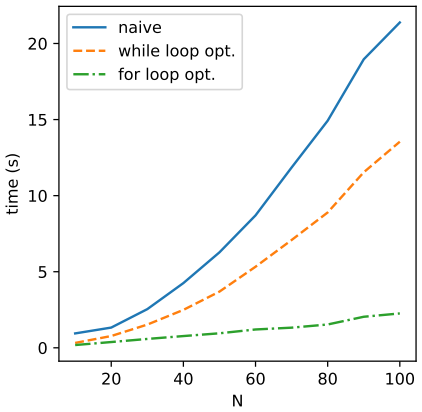


Fig. 4. Posterior inference time for the GMM by the size of observations and optimisation strategy.

For posterior inference, we consider the single-site Metropolis Hastings (MH) algorithm [Wingate et al. 2011] to infer the most likely clusters for data set y[2], as well as the most probable allocation of each data point y_i to one cluster $z_i \in [1, K]$. We fix the number of clusters to $K = 4$. The single-site MH algorithm works by selecting a single address α in each iteration at random and proposing a new value v according to the kernel $v \sim Q_\alpha(v)$. The proposed trace $\mathtt{tr}' = \mathtt{tr}[\alpha \mapsto v]$ is then accepted with probability

$$A = \min\left(1, \frac{p(\mathtt{tr}') \times Q_\alpha(\mathtt{tr}(\alpha))}{p(\mathtt{tr}) \times Q_\alpha(\mathtt{tr}'(\alpha))}\right).$$

Since we only change tr at one address, we can make use of the factorisation theorems to more efficiently compute $p(\mathtt{tr}')/p(\mathtt{tr})$. That is, if we update at address α, we only need to recompute the factors corresponding to the sample statements where α is in the provenance set. The provenance set of the arguments of distributions at sample nodes obtained from the static analysis in the sense

[2]Note that for the model at hand there exist much more efficient inference algorithms and the considered MH algorithm servers only for demonstrative purposes.

of Eq. (2) are given by

$$\begin{aligned}
\text{prov}(N_{\text{"w"}}, \texttt{fill}(\delta, K)) &= \emptyset,\\
\text{prov}(N_{\text{"mu"}}, \xi) = \text{prov}(N_{\text{"mu"}}, 1/\texttt{sqrt}(\kappa)) &= \emptyset,\\
\text{prov}(N_{\text{"var"}}, \alpha)) = \text{prov}(N_{\text{"var"}}, \beta)) &= \emptyset,\\
\text{prov}(N_{\text{"z"}}, \texttt{w}) &= \{\text{"w"}\},\\
\text{prov}(N_{\text{"y"}}, \texttt{get}(\texttt{means}, \texttt{z})) &= \{\text{"mu_}k\text{"}\colon k = 1, \ldots, K\} \cup \{\text{"z_}j\text{"}\colon j = 1 \ldots, N\},\\
\text{prov}(N_{\text{"y"}}, \texttt{get}(\texttt{vars}, \texttt{z})) &= \{\text{"var_}k\text{"}\colon k = 1, \ldots, K\} \cup \{\text{"z_}j\text{"}\colon j = 1 \ldots, N\}.
\end{aligned}$$

For instance, if we change the value for the center of cluster 1, "mu_1", we only need to update the contributions of all "y_i" to p. We do not have to recompute the contributions of the variables with address "w", "var_k", or "z_i". If we update one of the allocations "z_i", we have to be careful. From the static perspective of Algorithm 1, we do not know which value `tr`("z_j") is used in the sample statement for "y_i". Therefore, we have to recompute the contributions to the density of variables "y_j" for all $1 \leq j \leq N$.

Furthermore, for this particular program, we can *syntactically* unroll the while loops.

```
w = sample("w", Dirichlet(fill(δ,K)))
means = list(); vars = list();
mu = sample("mu_"+str(1), Normal(ξ,1/sqrt(κ))
var = sample("var_"+str(1), InverseGamma(α,β))
means = append(means,mu); vars = append(vars,var)
...
z = sample("z_"+str(1),Categorical(w))
y = sample("y_"+str(1), Normal(get(means,z),sqrt(get(vars,z))))
z = sample("z_"+str(2),Categorical(w))
y = sample("y_"+str(2), Normal(get(means,z),sqrt(get(var,z)))))
...
```

Note that this is different to the *mathematical* unrolling of while loops in Section 4.1. For the unrolled program, all sample addresses are constant and we statically know that if we update "z_i" we only need to recompute the contribution of the single data point "y_i".

In Fig. 4, you can see how the optimisations described above influence the runtime of the single-site Metropolis Hastings inference algorithm. Using the provenance information of the unmodified program with a while loop results in a 2x speed-up. This is explained by the fact that as we increase the data size N, in most iterations we update at an address "z_i". Instead of reusing the contributions of variables "z_j", $j \neq i$, and only recomputing at addresses "y_j" (N computations), the naive implementation computes $p(\texttt{tr}')$ from scratch ($2N + 2K + 1$ computations), thus doubling the runtime. Modifying the program by unrolling the while loop and using its provenance information decreases the runtime by a factor of 10 for $N = 100$. The reason is that now in most iterations only the contributions of variables "z_i" and "y_i" for a single $1 \leq i \leq N$ have to be computed.

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]. Operational semantics were given in form of machine-state transitions, where the state contains two infinite stacks of continuous and discrete random choices. When executing a sample statement an element will be popped from these stacks. In the same work, denotational semantics were also presented, where the program state is modelled as a probability distribution

and a program is interpreted as an operator on distributions. Derivatives of this semantics can be found throughout literature, e.g. see [Hur et al. 2015] and [Dwyer et al. 2017].

In recent years, a lambda calculus for probabilistic programming was presented [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 actively researched [Dahlqvist and Kozen 2019; Staton et al. 2016].

Semantics for probabilistic programming languages are also developed for specific applications like approximate inference [Huang et al. 2021] or handling exceptions [Bichsel et al. 2018].

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 base on trace types are introduced in [Lew et al. 2019].

6.2 Graphical Representations and Static 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] or Infer.NET [Minka 2012] require the user to explicitly define their model as a graph. 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].

While the connection of simple probabilistic programs to Bayesian networks is known [van de Meent et al. 2018], our approach shows how to *statically* determine a non-trivial factorisation and show equivalence to Markov networks even for programs containing while loops. 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].

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.

However, in general, the field of static analysis for probabilistic programming is young and under-explored [Bernstein 2019]. We list a few recent advances. In 2014, [Sampson et al. 2014] introduced probabilistic assertion statements and an evaluation approach to either statically or dynamically verify them. In 2018, [Hoffman et al. 2018] presented a method to automatically analyse the source code of density functions for conjugacy structure and leveraged the structure for improved inference.

Stan has received the most attention with regards to static analysis. In 2018, SlicStan [Gorinova et al. 2019], a compositional version of the probabilistic programming language, was published. In this work, a semantics-preserving translation to Stan was developed. In the following years, Stan was also extended with a pedantic compilation mode [Bernstein 2023] statically checking for frequent programming mistakes. Lastly, a compilation scheme was designed to translate a Stan program to a generative PPL [Baudart et al. 2021].

Most recently [Wang et al. 2021] developed a type system to enforce absolute continuity of Pyro guide programs and [Li et al. 2023] presented an automatic approach to generate sound guide programs for a given Pyro program.

7 CONCLUSION

In this work, we introduced address-based operational semantics for a formal probabilistic programming language. Our semantics are better suited to reason about PPLs which make use of user-defined sample addresses like Gen, Pyro, or PyMC. 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 programs to either Bayesian or Markov networks. Lastly, we demonstrated that we can speed-up a single-site Metropolis Hastings algorithm by a factor of 10 by deriving an optimised computation of the program density from the dependency structure.

REFERENCES

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.

Guillaume Baudart, Javier Burroni, Martin Hirzel, Louis Mandel, and Avraham Shinnar. 2021. Compiling Stan to generative probabilistic languages and extension to deep probabilistic programming. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 497–510.

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.

Benjamin Bichsel, Timon Gehr, and Martin Vechev. 2018. Fine-grained semantics for probabilistic programs. In *Programming Languages and Systems: 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings 27*. Springer, 145–185.

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.

Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of statistical software* 76, 1 (2017).

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.

Matthew B Dwyer, Antonio Filieri, Jaco Geldenhuys, Mitchell Gerrard, Corina S Păsăreanu, and Willem Visser. 2017. Probabilistic program analysis. In *Grand Timely Topics in Software Engineering: International Summer School GTTSE 2015, Braga, Portugal, August 23-29, 2015, Tutorial Lectures 5*. Springer, 1–25.

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.

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.

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

Zixin Huang, Saikat Dutta, and Sasa Misailovic. 2021. Aqua: Automated quantized inference for probabilistic programs. In *Automated Technology for Verification and Analysis: 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18–22, 2021, Proceedings 19*. Springer, 229–246.

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).

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.

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

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.

George Matheos, Alexander K Lew, Matin Ghavamizadeh, Stuart Russell, Marco Cusumano-Towner, and Vikash Mansinghka. 2020. Transforming worlds: Automated involutive MCMC for open-universe probabilistic models. In *Third Symposium on Advances in Approximate Bayesian Inference*.

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.

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.

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

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).

Di Wang, Jan Hoffmann, and Thomas Reps. 2021. Sound probabilistic inference via guide types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 788–803.

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.

A PROOFS

A.1 Proof of Proposition 1

Statement. *For all programs S with control-flow-graph G, it holds that for all traces* $\mathtt{tr}$

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

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 comparing the recursive 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 $S = (\text{ if } E \text{ then } S_1 \text{ else } S_2)$ 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 $(\sigma, S_i) \Downarrow^{\mathtt{tr}} \sigma'$. □

A.2 Proof of Proposition 3

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

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

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:

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

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

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

A.3 Proof of Lemma 4

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

$$\{\iota_{\text{cfg}}(M') \colon M' \in \text{RD}(M, x)\} \subseteq \text{RD}(\iota_{\text{cfg}}(M), x) \tag{11}$$

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 only consisting 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 $\overrightarrow{M_1} :== (M, \ldots, M_i, \text{END})$ and $\overrightarrow{M_2} := (\text{START}, M_{i+1}, \ldots, M)$ 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 is $\overrightarrow{N} := (N', \ldots, N_1, N_2, \ldots, N)$ 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 $\overrightarrow{M_i} := (\text{START}, M', \ldots, M, \text{END})$ and get a CFG path $\overrightarrow{N_i} = (\text{START}, N', \ldots, N, \text{END})$. 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_i join nodes, and H_i sub-graphs. For the sake of brevity we consider three cases:

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

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

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

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

  where B, B_i, and J_i are not assign nodes.
- For $M' \in H_i$ and J_k

$$\overrightarrow{M} = (\underbrace{M', \ldots, M_i}_{\in H_i}, B_{i+1}, \ldots, B_l, J_l, J_{l-1}, \ldots, J_k)$$

  we construct the path

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

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

□

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

$$\mathrm{BP}_{\mathrm{cfg}}(M) := \{\iota_{\mathrm{cfg}}(M') \colon M' \in \mathrm{BP}(M)\} = \mathrm{BP}(\iota_{\mathrm{cfg}}(M)) \tag{12}$$

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_i join nodes, and H_i sub-graphs.

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

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

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

□

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

$$\text{prov}(M, x) \subseteq \text{prov}(\iota_{\text{cfg}}(M), x) \tag{13}$$

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

Received 16 November 2023; revised ??; accepted ??