Non-termination (in)sensitive slicing for probabilistic programs

Abu Naser Masud[1] and Federico Olmedo[2]

[1] Mälardalen University, Sweden,
abu.naser.masud@mdu.se
[2] University of Chile, Chile
federico.olmedo@dcc.uchile.cl

Abstract. The probabilistic programming language offers a high degree of flexibility through its expressive syntax and semantics. It includes specialized programming primitives for random assignments and "observe" statements, crucial for conditioning the model on observed data. This study delves into several aspects of slicing probabilistic programs (PP), spanning slice semantics, different static slicing types, slicing algorithms, and proof of correctness. Previous research on slicing PP adopt a program semantics that conflates observation failure with nontermination, yielding nontermination insensitive slices. However, observation failure and nontermination are distinct phenomena. By disentangling them in the semantics, we have identified several variants of static slicing, namely nontermination sensitive and nontermination insensitive slicing. The latter is further categorized into distribution sensitive and distribution insensitive, based on whether the slice strictly preserves the original program's outcome distribution, even in nonterminating scenarios, or weakly considers only terminating executions. We have provided semantic characterization of all the variants and devised novel algorithms to compute them by introducing a new concept called observe-nontermination dependence. Additionally, we have developed (bi)simulation based proof techniques to verify the correctness of computing all slice variants. Our contributions deepen the understanding of static slicing in probabilistic programming, potentially impacting various application domains.

1 Introduction

Program slicing plays a significant role in the process of software development. Put simply, the goal of slicing is, given a program and a set of variables of interest at a given program point, to identify the program fragments that can be removed without afecting the variable values at said point [23]. To identify such fragments, slicing techniques primarily rely on a static analysis of data and control dependencies among variables. As so defined, program slicing have proved particularly useful in tasks such as testing, program understanding, program debugging and extraction of reusable components, to name a few [25].

In this work, we focus, in particular, on slicing over *probabilistic* programs. Loosely speaking, probabilistic programs are programs written in ordinary programming languages that, on top of their usual constructs, offer the possibility of *i*) sampling values from probability distributions (*aka* random sampling) and *ii*) conditioning the value of variables through so-called observe statements (*aka* conditioning).

Probabilistic programs have found application in numerous domains. They are central in the field of machine learning due to their compelling properties for representing probabilistic models [8]. They are the cornerstone of modern cryptography —all encryption schemes are by nature probabilistic [9]—, and also of traditional randomized algorithms [14]. Finally, they are critical for privacy pourposes, as demonstrated by the notion of differential privacy [6].

Nevertheless, the research on methods for slicing probabilistic programs has been rather scarce. Hur *et al.* [10] developed the first slicing technique for (imperative) probabilistic programs. They showed that to achieve correct slices, traditional data and control dependence must be complemented with a new form of dependence accounting for the (more intricate) effects of conditioning. A few years later, Amtoft and Banerjee [2] introduced the notion of probabilistic control-flow graphs, which allows a direct adaptation of conventional slicing machinery to the case of probabilistic programs.

Both approaches [10] and [2] suffer, however, from several limitations. First, they adopt a program semantics that conflates observation violation with non-termination —two phenomena that, in our view, should be distinguished. Second, they support only a particular form of slicing known as *non-termination insensitive*. In order to yield potentially smaller sliced programs, this form of slicing allows non-terminating executions in the original program to "become" terminating in the sliced program. While this may be sensible for some applications, in other applications it is of utter relevance that the original and the sliced program share the same termination behaviour. Finally, the form of (non-termination insensitive) slicing they support is overly restrictive, leaving out program slices that one would arguably deem valid. Section 2 further expands on these shortcoming, while also providing illustrative examples.

Motivated by these limitations, in this paper, we develop the first slicing technique for probabilistic programs supporting *both* non-termination insensitive and sensitive slices. Like [2], our development builds on classical notions of slicing, which are adapted and generalized from the deterministic case to the probabilistic case.

Our main contributions include:

A novel taxonomy of slicing for probabilistic programs. We formally define the notions of non-termination sensitive and insensitive slicing for probabilistic programs (Section 5); this characterization naturally generalizes the counterpart notion for deterministic programs. We further divide non-termination insensitive slicing into two sub-categories: distribution sensitive and distribution insensitive, which differ on whether the sliced program is required to preserve the *relative* outcome probabilities of the original program

or not. The cornerstone of this characterization (and all our development) is a refined operational semantics of programs that distinguishes between observation violation and non-termination (Section 4).

Syntactic conditions for generating different slice classes. We identify a new form of dependence, dubbed *observe(-nontermination)* dependency, that *uniformly* captures the indirect —more subtle— dependencies induced by observe statements and/or non-terminating loops in probabilistic programs (Section 6). Through a bisimulation argument, we show that properly combining traditional data and control dependencies with this novel form of dependency yields correct program slices, for our entire slicing taxonomy (Section 7).

An algorithm for computing slices. TODO: Abu, I will need your help here.REMARK: What is the novelty of this algorithm? It is only the computation of the *observe(-nontermination)* dependency or is there something else? How does it compare to previous algorithms? Do we prove it correct?

TODO: Once we have the final paper structure, we have to add a "Paper organization" paragraph.

2 Overview

We next overview the main novel features of our slicing technique, while elaborating on the limitations of previous approaches they address.

2.1 Refined Program Semantics

Program P_0 :	Program P_1 :	Program $P_1^\star$:
1: $x :\approx \mathsf{unif}[1,4]$;	1: $x :\approx \mathsf{unif}[1,4]$;	1: $x :\approx \mathsf{unif}[1,4]$;
2: $y :\approx \mathsf{unif}[1,4]$;	2: $y :\approx \mathsf{unif}[1,4]$;	2: skip;
3: observe $(y = 4)$;	3: while $(y \neq 4)$ do skip;	3: skip;
4: return x	4: return x	4: return x

Fig. 1: Semantics conflating non-termination with failure to establish observe statements: Under the semantics in [10, 2], programs P_0 and P_1 are semantically equivalent, and $P_1^\star$ is a valid (non-termination insensitive) slice of both.

To start with, let us consider the programs P_0 and P_1 in Fig. 1. Program P_0 samples two independent random integer values in the interval $[1, 4]$ and returns the first sample, observing that the second sample happens to be 4. This observation is denoted by the statement observe $(y = 4)$ in line 3. Program P_1, on the other hand, enters a diverging loop if the second sample does not happen to be 4.

From an operational perspective, P_0 admits $4 \times 1 = 4$ executions, characterized by having $y = 4$ (since the remaining $4 \times 3 = 12$ executions, where $y \neq 4$, are blocked) and each occurring with probability $1/4 \times 1/4 = 1/16$. On the other hand, P_1 admits $4 \times 4 = 16$ executions: those 4 executions where $y = 4$ are terminating and occur also with probability $1/16$, while those 12 executions where $y \neq 4$ are diverging, and therefore do not yield any (observable) program outcome.

Existing slicing approaches [10, 2] adopt a program semantics that consists in the "normalized distribution of outputs over *terminating* runs" that pass observe statements. Under these semantics, the probability that P_0 returns any specific value, say 2, is calculated by dividing the probability $1 \times 1 \times 1/16$ of all valid and terminating executions[3] where $x = 2$ by the probability $4 \times 1 \times 1/16$ of all valid and terminating executions, yielding a result of $1/4$. The probability that P_1 returns 2 is determined by following the same procedure (noting that all executions induced by P_1 are valid, but only a subset are terminating), which yields, indeed, identical numbers. This remains true for any given return value and therefore, the semantics in [10, 2] do not distinguish program P_0 from program P_1. More generally, these semantics regard programs observe b and while $\neg b$ do skip as semantically equivalent, conflating thus failure to establish an observe statement with non-termination.

As argued by Bischsel *et al.* [4], we believe that failure to establish an observation and non-termination represent conceptually different phenomena, and this distinction should be reflected by the semantics adopted for slicing.

Even more critical is the fact that the aforementioned semantics do not *conservatively* extend that of a probabilistic language without conditioning, where no normalization is applied. For example, any stantard sematics [3] would report that the probability that P_1 outputs, say 2, is simply $1 \times 1 \times 1/16$, without rescaling it by any normalization factor.

To address both issues above, we adopt a semantics akin to that of Olmedo *et al.* [17], where we normalize *w.r.t.* the probability of *all valid executions* —not only terminating ones. This semantic distinguishes program P_0 from program P_1. For instance, it reports that program P_0 returns 2 with probability $1/4$ (and terminates with probability 1), while program P_1 returns 2 with probability $1/16$ (and terminates with probability $1/4$).

As a final remark for incoming examples, observe that when determining the output distribution of observe-free programs, we can dispense of any normalization factors since they will always be 1.

2.2 Support for Non-termination Sensitive Slicing

As a direct consequence of the semantics they adopt, [10, 2] present a slicing approach that is *insensitive* to non-termination. Program slicing typically comes in two flavors: non-termination sensitive and insensitive, differing on how non-terminating executions of the original program are treated in the sliced pro-

[3] By *valid* execution, we mean executions that pass all observe statements.

Program P_2 :

1: $x :\approx \mathsf{unif}[1,4]$;
2: $y :\approx \mathsf{unif}[1,4]$;
3: while $(x = 4)$ do skip;
4: return x

Program $P_2^\bullet$:

1: $x :\approx \mathsf{unif}[1,4]$;
2: skip;
3: while $(x = 4)$ do skip;
4: return x

Program $P_2^\star$:

1: $x :\approx \mathsf{unif}[1,4]$;
2: skip;
3: skip;
4: return x

Fig. 2: Non-termination sensitive and insensitive slicing: Programs P_2 (original program), $P_2^\bullet$ (non-termination sensitive slice) and $P_2^\star$ (non-termination insensitive slice).

gram (both variants treat terminating executions uniformly, strictly requiring their preservation). Loosely speaking, a non-termination *sensitive* slicing must preserve all non-terminating executions of the original program in the sliced program. On the other hand, a non-termination *insensitive* slicing allows non-terminating executions of the original program to become terminating in the sliced program (and therefore the "termination domain" of the sliced program can be larger than that of the original program).

For example, given the program P_1 in Fig. 1, the slicing approaches in [10, 2] allow removing the while-loop together with the random assignment to y, yielding the sliced program $P_1^\star$. However, while program P_1 terminates with a probability of only $4 \times 1 \times 1/16 = 1/4$, program $P_1^\star$ terminates with probability $1 > 1/4$. In contrast to [10, 2], a non-termination *sensitive* slicing of P_1 would not allow removing the while-loop.

TODO: Confirm this for [10]

As exhibited in this example, non-termination insensitive slicing can be more agressive, leading to smaller sliced programs, which, for some applications such as program understanding and debugging, can be more desirable than having —larger— sliced programs that do preserve non-termination.

However, there exist applications of probabilistic programming such as cryptography and differential privacy, which regard non-termination as an observable phenomena (by the so-called *attacker*) and preserving non-termination in these domains becomes of paramount importance.

Notably, our slicing approach supports —and distinguishes between— both forms for of slicing. Like for deterministic programs, the key ingredient to establish this distinction is the variant of control dependence among variables considered: so-called *weak* control dependence will lead to a slicing that is non-termination insensitive, and *strong* control dependence to a slicing that is non-termination sensitive.

2.3 Natural Notion of Non-termination Insensitive Slicing

REMARK: @Abu, we need a reference here. Can we use the same as in the intro?

Weiser's original notion of non-termination insensitive slicing for *deterministic* programs requires that whenever the original program halts its execution on a given input, the sliced program also halt on that input, traversing the same execution path with equivalent values for relevant variables. Our counterpart notion for *probabilistic* programs is inspired by this, generalizing it in a quantitative manner to account for execution probabilities.

To ilustrate this, let us consider program P_2 from Fig. 2, and program $P_2^\star$ which is obtained from P_2 by slicing away the while-loop in line 3, together with the random assignment to y in line 2. Observe that P_2 admits $3 \times 4 = 12$ terminating executions, which can be partitioned into three groups of 4 executions each, according to the value of variable x, *e.g.*, the first group gathers the 4 executions where $x = 1$, and likewise for the second ($x = 2$) and third ($x = 3$) group; each group has an overall probability of $4 \times 1/16 = 1/4$.

Each of these groups of terminating executions in P_2 is "mirrored by" a terminating execution in $P_2^\star$, with equivalent value for x and occurring with the same probability ($1/4$). Therefore, we regard $P_2^\star$ as a valid non-termination insensitive slice of P_2. Furthermore, note that the non-terminating executions of P_2 (i.e. those where $x = 4$) "become" terminating in $P_2^\star$, matching the intuition behind non-termination insensitive slicing that we provide in Section 2.2.

While our notion of non-termination insensitive slice for probabilistic programs arguably captures Weiser's original intuition, previous approaches [10, 2] consider a very restricted form of non-termination insensitive slicing, which, for example, rules out $P_2^\star$ as a valid slice of P_2. This is because their normalized output distribution do not fully agree (intuitively, because P_2 outputs 4 with a null probability and $P_2^\star$ does it with a strictly positive probability). Interestingly, the only proper slice of P_2 that they deem valid is $P_2^\bullet$, where only the random assignment to y is removed.

3 Preliminaries

In this section, we present the notions related to probability distributions, bring the concept of CFG, and various program dependence relations that we use for our subsequent development.

3.1 Probability related notions

Notion of subdistribution. Given a denumerable set A, a (necessarily discrete) probability subdistribution over A is a function

$$\mu \colon A \to [0,1] \qquad \text{such that} \qquad \sum_{a \in A} \mu(a) \leq 1 \ .$$

The probability of an event $A_0 \subseteq A$, by abuse of notation also written $\mu(A_0)$, is defined as $\mu(A_0) = \sum_{a \in A_0} \mu(a)$. We use $\mathsf{w}(\mu)$ to denote the *weight* $\sum_{a \in A} \mu(a)$ of μ and $\mathsf{support}(\mu)$ to denote the its *support* $\{a \in A \mid \mu(a) > 0\}$. Furthermore, we use $\mathcal{D}^{\leq 1}(A)$ to denote the set of subdistributions over A and $\mathcal{D}^{=1}(A)$ to denote the subset of subdistributions of weight 1.

Finally, we write $\mathbf{0}$ for the *null* subdistribution that assigns probability 0 to all elements of its carrier set.

Operations. Given $\mu_1, \mu_2 \in \mathcal{D}^{\leq 1}(A)$, we define the (partial) sum $\mu_1 + \mu_2 \in \mathcal{D}^{\leq 1}(A)$ as

$$(\mu_1 + \mu_2)(a) \;=\; \mu_1(a) + \mu_2(a) \qquad \text{provided } \mathsf{w}(\mu_1) + \mathsf{w}(\mu_2) \leq 1$$

Similarly, given a scaling factor $c \in \mathbb{R}_{\geq 0}$ and a subdistribution $\mu \in \mathcal{D}^{\leq 1}(A)$ such that $c \cdot \mathsf{w}(\mu) \leq 1$, we write $c \cdot \mu$ for the subdistribution (in $\mathcal{D}^{\leq 1}(A)$ defined as $(c \cdot \mu)(a) = c \cdot \mu(a)$. Furthermore, given a distribution $\mu \in \mathcal{D}^{\leq 1}(A \times B)$ over a product space, we use $\pi_1(\mu)$ (resp. $\pi_2(\mu)$) to denote its *first* (resp. *second*) *marginal*, i.e. the distribution in $\mathcal{D}^{\leq 1}(A)$ (resp. $\mathcal{D}^{\leq 1}(B)$) defined as $\pi_1(\mu)(a) = \sum_{b \in B} \mu(a, b)$ (resp. $\pi_2(\mu)(b) = \sum_{a \in A} \mu(a, b)$).

Finally, given $\mu \in \mathcal{D}^{\leq 1}(A)$ and $A_0 \subseteq A$ we define $\mu_{|A_0} \in \mathcal{D}^{\leq 1}(A)$, the *restriction* of μ *w.r.t.* A_0 by:

$$\mu_{|A_0} \;=\; \begin{cases} \mu(a) & \text{if } a \in A_0 \\ 0 & \text{otherwise} \end{cases}$$

Note that we always have $\mu = \mu_{|A_0} + \mu_{|A \setminus A_0}$.

Relation lifting. There exists a canonical approach for lifting relations over a pair of sets to relations over distributions over such sets, in terms of so-called couplings. More concretely, given relation $R \subseteq A \times B$, relation $R^{\uparrow} \subseteq \mathcal{D}^{\leq 1}(A) \times \mathcal{D}^{\leq 1}(B)$, the *lifting* of R to $\mathcal{D}^{\leq 1}(A) \times \mathcal{D}^{\leq 1}(B)$, is defined as

$$\mu_1 \, R^{\uparrow} \, \mu_2 \;=\; \exists \mu \in \mathcal{D}^{\leq 1}(A \times B). \begin{cases} \pi_1(\mu) = \mu_1 \wedge \pi_2(\mu) = \mu_2 \\ \mathsf{support}(\mu) \subseteq R \end{cases}$$

Formally, a distribution $\mu \in \mathcal{D}^{\leq 1}(A \times B)$ where $\pi_1(\mu) = \mu_1$ and $\pi_2(\mu) = \mu_2$ is known as a *coupling* between μ_1 and μ_2. If the support of such a coupling lies within relation R, then it is a *witness* of the relation $\mu_1 \, R^{\uparrow} \, \mu_2$. In this case, we write $\mu \models \mu_1 \, R^{\uparrow} \, \mu_2$.

Order structure.

Todo: Complete

3.2 Program dependences

Todo: bring the definition of CFG here

We consider the CFG representation of probabilistic programs as defined in Def. 5. The set of successors and predecessors of any CFG node n is denoted by $succ(n)$ and $pred(n)$. The notations $def(n)$ and $ref(n)$ denote the sets of program variables that are defined and referenced respectively at n. A finite CFG path $n_1, n_2, \ldots, n_k$ (or $[n_1..n_k]$ for short) is a sequence of CFG nodes such that $n_{i+1} \in succ(n_i)$ for all $1 \leq i \leq k-1$ and $k \geq 1$. An infinite path $n_1, n_2, \ldots$ is denoted by $[n_1..]$. A path is *non-trivial* if it contains at least two nodes. A *final* node is either a non-predicate node with out-degree 0 or a predicate node having a missing successor. A path is called *maximal* or *complete* if it is either an infinite path or a finite path that ends at a final node. Sometimes,

we use the notation $\pi - \{m, n\}$ to denote the set of CFG nodes in the CFG path π excluding m and n. *Data* and *control* dependences are two fundamental relations used in computing dependence-based slicing. They are defined over the CFG representation of programs.

Definition 1 (Data Dependency [22]). *Node n is* data dependent *on node m (written $m \xrightarrow{dd} n$) in the CFG G if there is a program variable v such that: (1) there exists a nontrivial path π in G from m to n such that for every node $m' \in \pi - \{m, n\}$, $v \notin def(m')$, and (2) $v \in def(m) \cap ref(n)$.*

Intuitively, $m \xrightarrow{dd} n$ denotes that n uses the value of a program variable that is set at m.

The first formal definition of standard control dependence relation is provided by Ferrante et al.[7] based on the *postdominator* [21] relation. Node n is said to *postdominate* node m if and only if every path from m to the exit node n_e goes through n. Note that this definition assumes that G has a single exit node n_e. *n strictly postdominates m* if n postdominates m and $n \neq m$. The standard *control dependency relation* can be defined as follows:

Definition 2 (Control Dependency [7, 22]). *Node n is* control dependent *on node m (written $m \xrightarrow{pcd} n$) in the CFG G if (1) there exists a nontrivial path π in G from m to n such that every node $m' \in \pi - \{m, n\}$ is postdominated by n, and (2) m is not strictly postdominated by n.*

The relation $m \xrightarrow{pcd} n$ indicates that there must be two branches originating from m, where n is consistently executed in one branch but may not be executed in the other. Several control dependency relations [18, 19, 22, 20] have been proposed to extend the standard relation, addressing various scenarios in program control flow such as CFGs with infinite loops, no end node, or multiple end nodes. Danicic et al. [5] introduced two generalizations of control dependence, termed weak and strong control closure, which account for non-termination insensitivity and nontermination sensitivity respectively. Many existing control dependencies can be seen as specific instances of these generalizations. In the following, we provide the definition of weak and strong control closure by summarizing the concepts from Danicic et al.

Definition 3 (Weak Control Closure). *Let N' be a subset of nodes in the CFG $G = (N, E)$. N' is weakly control closed iff for all $n \notin N'$:*

1. *n is reachable from N', and*
2. *all CFG paths $n_1 = n, \ldots, n_k$ from n meet at $n_k \in N'$ such that $n_i \notin N'$ for $1 < i < k$.*

While computing a subset of CFG nodes N' as the slice, if N' is not weakly control closed, there exists a CFG node $n \notin N'$ having at least two distinct CFG paths without going through a node in N' and meeting at distinct nodes $m_1, m_2 \in N'$ such that $m_1 \neq m_2$. This implies that n is a predicate node, and

based on the outcome of the condition at n during an execution, either m_1 or m_2 will be executed first. Consequently, we say that the weak control dependence relation $n \xrightarrow{wcd} m_i$ holds for $i = 1, 2$. Weak control dependence relation does not take into account the effect of nontermination. Next, we define strong control closure that will lead to the definition of the strong control dependence relation $\xrightarrow{scd}$:

Definition 4 (Strong Control Closure). *Let N' be a subset of nodes in the CFG $G = (N, E)$. N' is strongly control closed if, for every node $n \notin N'$:*

1. *n is reachable from N', and either (2) or (3) below are satisfied:*
2. *all CFG paths $n_1 = n, \ldots, n_k$ from n meet at the same node $n_k \in N'$ such that $n_i \notin N'$ for $1 < i < k$, and all complete paths from n contain at least one node from N',*
3. *no node in N' is reachable in G from n.*

In simpler terms, this definition essentially demands that the conditions for weak control closure are met. Consequently, if the control dependence relation $n \xrightarrow{wcd} m_i$ holds, then the strong control dependence relation $n \xrightarrow{scd} m_i$ also holds. Additionally, if there exists a predicate node n with two branches: one leading to a CFG path $n_1 = n, \ldots, n_k$ that meets at $n_k \in N'$ without passing through any node in N', and the other branch leading to an infinite path representing nonterminating execution without passing through any node $m \in N'$, then the relation $n \xrightarrow{scd} m$ holds. In the rest of the paper, we use the symbol $\xrightarrow{cd}$ to denote any control dependency relation.

4 Programming Model

4.1 Syntax

To describe probabilistic programs we adopt a simple imperative language extended with random assignments and observe statements, dubbed pWhile. A *program* p is a command, followed by a return expression. A *command* is either a no-op (skip), a deterministic assignment ($x \coloneqq a$), a random assignment ($x :\approx d$), a sequential composition ($c_1; c_2$) of two other commands, a conditional branching (if b then c_1 else c_2), a guarded loop (while b do c) or an observation (observe b). Formally, it is given by grammar:

p	::=	c return a	*Program*
c	::=	skip $\mid x \coloneqq a \mid x :\approx d \mid c_1; c_2 \mid$ if b then c_1 else $c_2 \mid$ while b do $c \mid$ observe b	*Commands*
a	::=	$z \mid x \mid -a \mid a_1 + a_2 \mid a_1 \times a_2 \mid \ldots$	*Arithmetic expressions*
b	::=	true $\mid$ false $\mid a_1 == a_2 \mid a_1 \leq a_2 \mid \ldots \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \ldots$	*Boolean expressions*
d	::=	distr$\{z_1 \mapsto p_1, \ldots, z_n \mapsto p_n\} \mid$ unif$[z_1, z_2] \mid$ binom$(n, p) \mid$ geom$(p) \mid \ldots$	*Distribution expressions*

Program variables and arithmetic expressions are assumed to be integer-valued. Arithmetic and Boolean expressions are rather standard. Distribution expressions represent probability distributions over the set of integers; for concreteness, we include distribution probabilities defined pointwise ($\mathsf{distr}\{z_1 \mapsto p_1, \ldots, z_n \mapsto p_n\}$), uniform distributions over an integer interval ($\mathsf{unif}[z_1, z_2]$), binomial distributions ($\mathsf{binom}(n, p)$) and geometric distributions ($\mathsf{geom}(p)$) but, in practice, distribution expressions need not be restricted to these kind of distributions.

As for commands, the class of distinguished statements are *probabilistic assignments* and observe statements. A probabilistic assignment $x :\approx d$ samples a value from distribution d and assigns it to program variable x. An observe statement observe b is just syntactic sugar for while $\neg b$ do skip. Said otherwise, it "block" executions violating b by assigning them a null probability.

TODO: Add discussion saying we follow . . . , and that observe statements do not renormalize.

We use $\mathcal{V}$ to denote the finite set of program variables (ranged over by x), c to denote the set of programs (ranged over by c), $\mathcal{AE}$ to denote de set of arithmetic expressions (ranged over by a), $\mathcal{BE}$ to denote de set of Boolean expressions (ranged over by b) and $\mathcal{DE}$ to denote de set of distribution expressions (ranged over by d).

Example 1 (Geometric distribution).

$$
\begin{array}{l}
m := 0; \\
b := 0; \\
\mathsf{while}\ (b == 0) \\
\quad b :\approx \mathsf{unif}[0, 1]; \\
\quad m := m + 1 \\
\mathsf{return}\ m
\end{array}
$$

4.2 Control flow graphs

To bridge the semantics of "original" programs and their sliced versions, we instrument program semantics to keep track the nodes of the control flow graph (CFG) of programs that their executions traverse. We thus continue by recalling the notion of control flow graph.

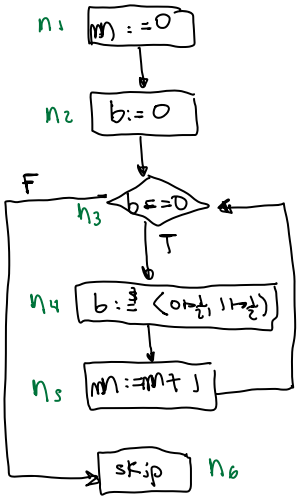
Definition 5 (Control flow graph). *A CFG is a directed graph $G = (\mathcal{N}, \mathcal{E}, start)$, where*

1. *$\mathcal{N}$ is the set of nodes that represent atomic commands in the program and is partitioned into two subsets, i.e. $\mathcal{N} = \mathcal{N}^S \uplus \mathcal{N}^P$, where $\mathcal{N}^S$ contains* statement *nodes (from no–ops, assignments, and* observe *statements) which have at most one succesor node and $\mathcal{N}^P$ contains* predicate *nodes (from conditional branching and guarded loops) which have (exactly) two successors. Moreover, we use $\mathcal{N}^E$ to denote the subset of nodes from $\mathcal{N}^S$ with no successor. These are* exit *(or end) nodes of G representing succesfull termination of a program trace.*
2. *Likewise, $start \in \mathcal{N}$ is a distinguished node representing the starting point of program execution (the entry point of the CFG).*

3. $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ *is the set of edges representing the possible flow of execution in the program. Exit nodes in* $\mathcal{N}^E$ *have no succesor, statement nodes in* $\mathcal{N}^S \setminus \mathcal{N}^E$ *have one succesor, given by function* $\mathsf{succ}(\cdot)$ *and predicate nodes in* $\mathcal{N}^P$ *have two successors, given by functions* $\mathsf{succ}_\mathsf{T}(\cdot)$ *and* $\mathsf{succ}_\mathsf{F}(\cdot)$*, and respectively referred to as the* true *and* false *successor node.*

The translation of a pWhile command into a CFG is rather standard and we refer the reader, *e.g.*, to [16] for a detailed account thereof. Note that if the pWhile command to translate ends with a guarded loop, then the node representing the guard will have no false succesor, violating the requirements in Definition 5. To accommodate these cases, we can add a spurious skip statement right after the loop.

Example 2 (Geometric distribution). Below we depict the CGF of the command in Example 1 (later on we account for the effect of the return statement):



To recover the statements (no-ops, assignments or Boolean expressions) associated to the nodes of a CFG, we assume the presence of function code. Concretely, $\mathsf{code}_G(n)$ returns the statement associated to node n by CFG G. When the underlying CFG G is understood from the context, we omit it and simply write $\mathsf{code}(n)$.

4.3 Semantics

As usual, a *store* σ is a mapping from variables to integer numbers; we use $\Sigma = \mathcal{V} \to \mathbb{Z}$ to denote the set of stores. Given a store $\sigma \in \Sigma$ and a variable $x \in \mathcal{V}$, we write $\sigma[x \mapsto z]$ for the store that is obtained from σ, by updating the value of x to z. To give semantics to programs, we assume the presence of *interpretation functions*

$$[\![\cdot]\!] \colon \mathcal{AE} \to \Sigma \to \mathbb{Z}, \qquad [\![\cdot]\!] \colon \mathcal{BE} \to \Sigma \to \mathbb{B} \quad \text{and} \quad [\![\cdot]\!] \colon \mathcal{DE} \to \Sigma \to \mathcal{D}^{=1}(\mathbb{Z}) \ ,$$

$$\frac{n \in \mathcal{N}^S \setminus \mathcal{N}^E \qquad \mathsf{code}(n) = \mathsf{skip}}{\langle n, \mu, p\rangle \;\longrightarrow\; \langle \mathsf{succ}(n), \mu, p\rangle} \text{ [Skip]}$$

$$\frac{\begin{array}{c} n \in \mathcal{N}^S \setminus \mathcal{N}^E \qquad \mathsf{code}(n) = x \coloneqq a \\ \mu'(\sigma') = \mu(\{\sigma \in \Sigma \mid \sigma[x \mapsto [\![a]\!]\,\sigma] = \sigma'\}) \end{array}}{\langle n, \mu, p\rangle \;\longrightarrow\; \langle \mathsf{succ}(n), \mu', p\rangle} \text{ [Assign]}$$

$$\frac{\begin{array}{c} n \in \mathcal{N}^S \setminus \mathcal{N}^E \qquad \mathsf{code}(n) = x :\approx d \\ \text{Assuming } \mathcal{V} = \{x, x_1, \ldots, x_n\}, \\ \mu'(x \mapsto z, x_1 \mapsto z_1, \ldots, x_n \mapsto z_n) = \sum_{\sigma \in \Sigma \mid \bigwedge_{i=1}^{n} \sigma(x_i) = z_i} \mu(\sigma) \times ([\![d]\!]\,\sigma\,z) \end{array}}{\langle n, \mu, p\rangle \;\longrightarrow\; \langle \mathsf{succ}(n), \mu', p\rangle} \text{ [Random]}$$

$$\frac{\begin{array}{c} n \in \mathcal{N}^S \setminus \mathcal{N}^E \qquad \mathsf{code}(n) = \mathsf{observe}\ b \\ \mu_T = \mu_{|\{\sigma \in \Sigma \mid [\![b]\!]\sigma = true\}} \qquad \mu_F = \mu_{|\{\sigma \in \Sigma \mid [\![b]\!]\sigma = false\}} \end{array}}{\langle n, \mu, p\rangle \;\longrightarrow\; \langle \mathsf{succ}(n), \mu_T, p + \mathsf{w}(\mu_F)\rangle} \text{ [Observe]}$$

$$\frac{\begin{array}{c} n \in \mathcal{N}^P \qquad \mathsf{code}(n) = b \\ \mu_T = \mu_{|\{\sigma \in \Sigma \mid [\![b]\!]\sigma = true\}} \end{array}}{\langle n, \mu, p\rangle \;\longrightarrow\; \langle \mathsf{succ}_{\mathsf{T}}(n), \mu_T, p\rangle} \text{ [Cond-T]} \qquad \frac{\begin{array}{c} n \in \mathcal{N}^P \qquad \mathsf{code}(n) = b \\ \mu_F = \mu_{|\{\sigma \in \Sigma \mid [\![b]\!]\sigma = false\}} \end{array}}{\langle n, \mu, p\rangle \;\longrightarrow\; \langle \mathsf{succ}_{\mathsf{F}}(n), \mu_F, p\rangle} \text{ [Cond-F]}$$

Fig. 3: Small-step semantics over CFGs.

that given an arithmetic/Boolean/distribution expression and a store returns an integer/a Boolean/a proper distribution over the set of integers. Their definitions are rather standard and are thus omitted.

To define the semantics of programs we follow [2] and assume that nodes of their associated CFGs modify probability distributions of over stores, rather than individual stores. Thus, in the reminder we refer to elements of $\mathcal{D}^{\leq 1}(\Sigma)$ as *probabilistic states* (which are ranged over by μ).

Let c be a pWhile program of CFG $G = (\mathcal{N}, \mathcal{E})$. A *configuration* of c is a triple $\langle n, \mu, p\rangle$, where $n \in \mathcal{N}$ is a node of G, $\mu \in \mathcal{D}^{\leq 1}(\Sigma)$ is a probabilistic state and $p \in [0, 1]$ is a probability. We introduce a small-step semantics $\longrightarrow$ that relates configurations. Loosely speaking,

$$\langle n, \mu, p\rangle \;\longrightarrow\; \langle n', \mu', p'\rangle$$

holds if, when executing command in node n from a probabilistic state μ and a cumulated probability p of violating observe statements, in one step we transition to (succesor) node n' resulting a in a probabilistic state μ' and a cumulated probability p' of violating observe statements. Note that as the execution of a program progress, the probability of violating observe statements can only remain equal or increase, we will always have $p' \geq p$. The formal definition of relation $\longrightarrow$ is provided in Figure 3.

Example 3. The small-step semantics for the CFG from Example 2 is found in Figure 5.

$$\frac{}{\langle n,\mu,p\rangle \longrightarrow_0 \langle\!\langle \mathbf{0}, p\rangle\!\rangle}$$

$$\frac{}{\langle n,\mu,p\rangle \longrightarrow_k \langle\!\langle \mu, p\rangle\!\rangle} \quad \text{provided } k \geq 1 \text{ and } n \in \mathcal{N}^E$$

$$\frac{\langle n,\mu,p\rangle \longrightarrow \langle \mathsf{succ}(n),\mu',p'\rangle \quad \langle \mathsf{succ}(n),\mu',p'\rangle \longrightarrow_k \langle\!\langle \mu'',p''\rangle\!\rangle}{\langle n,\mu,p\rangle \longrightarrow_{k+1} \langle\!\langle \mu'',p''\rangle\!\rangle} \quad \text{provided } k \geq 1 \text{ and } n \in \mathcal{N}^S \setminus \mathcal{N}^E$$

$$\frac{\begin{array}{c}\langle n,\mu,p\rangle \longrightarrow \langle \mathsf{succ_T}(n),\mu_T,p_T\rangle \quad \langle \mathsf{succ_T}(n),\mu_T,p_T\rangle \longrightarrow_k \langle\!\langle \mu'_T,p'_T\rangle\!\rangle \\ \langle n,\mu,p\rangle \longrightarrow \langle \mathsf{succ_F}(n),\mu_F,p_F\rangle \quad \langle \mathsf{succ_F}(n),\mu_F,p_F\rangle \longrightarrow_k \langle\!\langle \mu'_F,p'_F\rangle\!\rangle\end{array}}{\langle n,\mu,p\rangle \longrightarrow_{k+1} \langle\!\langle \mu'_T+\mu'_F, p'_T+p'_F\rangle\!\rangle} \quad \text{provided } k \geq 1 \text{ and } n \in \mathcal{N}^P$$

Fig. 4: Step-indexed semantics over CFGs.

Even though relation $\longrightarrow$ fully describes programs behaviour, one is usually particularly interested in describing the *distribution of final stores* or *final probabilistic state* reached by a program. Informally, we can construct this final probabilistic state by adding up all probabilistic states reached within exit nodes (the ones highlighted in blue in Figure 5) and normalizing it by the probability of all valid executions. Formally, we need to define a *step-indexed* relation $\longrightarrow_k$, which collects the final stores reached within k steps together with their respective probabilities (the larger the k, the more final states it collects and the larger the probabilities can grow); see Figure 4.

Example 4. Continuing with our running example, for any initial probabilistic state μ_0 and any probability p, we shall have:

- $\langle n_1,\mu_0,p\rangle \longrightarrow_k \langle\!\langle \mathbf{0},p\rangle\!\rangle$ for all $k = 0,\ldots,6$
- $\langle n_1,\mu_0,p\rangle \longrightarrow_k \langle\!\langle \{(1,1)\mapsto 1/2\},p\rangle\!\rangle$ for all $k = 7,\ldots,9$
- $\langle n_1,\mu_0,p\rangle \longrightarrow_k \langle\!\langle \{(1,1)\mapsto 1/2,(2,1)\mapsto 1/4\},p\rangle\!\rangle$ for all $k = 10,\ldots,12$
- $\langle n_1,\mu_0,p\rangle \longrightarrow_k \langle\!\langle \{(1,1)\mapsto 1/2,(2,1)\mapsto 1/4,(3,1)\mapsto 1/8\},p\rangle\!\rangle$ for all $k = 13,\ldots,15$

The formal definition of the step-indexed relation $\longrightarrow_k$ is provided in Figure 4. As one can already notice in Example 4, relation $\longrightarrow_k$ fulfills two important properties:

Determinism: For any configuration $\langle n,\mu,p\rangle$ and any number of steps k, there exists a single μ' and single p' such that $\langle n,\mu,p\rangle \longrightarrow_k \langle\!\langle \mu',p'\rangle\!\rangle$.

ω-chain: For any configuration $\langle n,\mu,p\rangle$, the sequence $\langle \langle\!\langle \mu_k,p_k\rangle\!\rangle \mid \langle n,\mu,p\rangle \longrightarrow_k \langle\!\langle \mu_k,p_k\rangle\!\rangle\rangle_{k\in\mathbb{N}}$ forms an ω-chain, where pairs $\langle\!\langle \mu_k,p_k\rangle\!\rangle$ are ordered componentwise, i.e. $\langle\!\langle \mu,p\rangle\!\rangle \leq \langle\!\langle \mu',p'\rangle\!\rangle$ iff $\mu \leq \mu'$ and $p \leq p'$.

The last property allows formally defining the final probabilistic state reached by a program of CFG G, when executed from initial probabilistic state μ_0. It is given by $[\![G]\!]\langle\!\langle \mu_0, 0 \rangle\!\rangle$, where

$$[\![G]\!]\langle\!\langle \mu_0, p \rangle\!\rangle \;=\; \frac{\mu'}{1-p'} \qquad \text{and} \qquad \langle\!\langle \mu', p' \rangle\!\rangle \;=\; \sup_{k \in \mathbb{N}} \langle \langle\!\langle \mu_k, p_k \rangle\!\rangle \mid \langle n, \mu, p \rangle \longrightarrow_k \langle\!\langle \mu_k, p_k \rangle\!\rangle \rangle$$

Note that if G violates observe statements with probability 1, i.e. $p' = 1$, then its semantics is undefined.

5 Slicing Taxonomy

Our contributions include novel characterizations of (and algorithmic support for) non-termination sensitive and insensitive slicing for probabilistic programs. We further divide non-termination insensitive slicing into two sub-categories: distribution sensitive and distribution insensitive. Let us present the main intuition behind these semantic notions.

REMARK: Maybe use another terminology, e.g. *portion*

Non-termination sensitive slicing. A non-termination *sensitive* slicing will preserve the probability of any possible (proper) program outcomes as well as the probability of non-termination. More formally, if P' is a subprogram of P (i.e. P' is obtained from P by replacing some of its statements by skip), we say that P' is a valid *non-termination sensitive* slice of P iff

$$\mathcal{D}_P = \mathcal{D}_{P'} \quad \text{and} \quad \mathcal{P}_P(\lightning) = \mathcal{P}_{P'}(\lightning) \;, \tag{1}$$

where $\mathcal{D}_P$ and $\mathcal{P}_P(\lightning)$ respectively denote P's outcomes probability distribution and P's probability of non-termination, and likewise for P'. We note that equation $\mathcal{P}_P(\lightning) = \mathcal{P}_{P'}(\lightning)$ in fact follows from equation $\mathcal{D}_P = \mathcal{D}_{P'}$ since the probability of non-termination $\mathcal{P}_P(\lightning)$ of any program P can be computed as one minus the probability of termination, i.e. one minus the total probability mass of $\mathcal{D}_P$.

To illustrate this class of slicing, let us consider program P_2 from Fig. 2. The probability of returning a proper value and of divergence, denoted by $\lightning$, are as follows:

$$\begin{array}{lcl} P_2(x=1) &=& 1/4 \\ P_2(x=2) &=& 1/4 \\ P_2(x=3) &=& 1/4 \\ P_2(x=4) &=& 0 \\ P_2(x=\_) &=& 0 \end{array} \qquad \text{and} \qquad P_2(\lightning) \;=\; 1/4 \;,$$

where symbol "_" stands for "otherwise", i.e., any other value not present in the preceding enumeration.

The only proper slice of P_2 that is non-termination sensitive is $P_2^{\bullet}$, whose output distribution obeys the very same equations as above.

Non-termination insensitive slicing. In a non-terminating insensitive slicing, the sliced program will preserve or increase the probability of returning any given outcome, in comparison with the original program. Intuitively, this is because some non-terminating execution in the original program may become terminating in the sliced program, and therefor, we are trading some *non-termination probability in the original program* for some *normal termination in the sliced program.* As a result, the sliced program may feature a smaller probability of non-termination. More formally, if P' is a subprogram of P, we say that P' is a valid *non-termination insensitive* slice of P iff

$$\mathcal{D}_P \leq \mathcal{D}_{P'} \quad \text{and} \quad \mathcal{P}_P(\lightning) \geq \mathcal{P}_{P'}(\lightning) \ . \tag{2}$$

Here, $\mathcal{D}_P \leq \mathcal{D}_{P'}$ (like $\mathcal{D}_P = \mathcal{D}_{P'}$ in Equation 1) should be understood pointwise, i.e. $\mathcal{D}_P(v) \leq \mathcal{D}_{P'}(v)$ for all v. By the same argument as before, note that equation $\mathcal{P}_P(\lightning) \geq \mathcal{P}_{P'}(\lightning)$ also follows from $\mathcal{D}_P \leq \mathcal{D}_{P'}$.

To ilustrate this class of slicing, let us consider the output distribution of $P_2^\star$, and compare it to that of P_2:

$$\begin{array}{lll} P_2^\star(x=1) = 1/4 \geq 1/4 & & \\ P_2^\star(x=2) = 1/4 \geq 1/4 & & \\ P_2^\star(x=3) = 1/4 \geq 1/4 & \text{and} & P_2^\star(\lightning) = 0 \leq 1/4 \\ P_2^\star(x=4) = 1/4 \geq 0 & & \\ P_2^\star(x=\_) = 0 \geq 0 & & \end{array}$$

From the above equations, we conclude that $P_2^\star$ is a non-termination insensitive slice of P_2.

In Figure 1 we can find another example of this kind of slicing: Program $P_1^\star$ is a non-termination insensitive slice of P_1:

$$\begin{array}{lll} P_1^\star(x=1) = 1/4 \geq 1/4 \times 1/4 = P_1(x=1) \\ P_1^\star(x=2) = 1/4 \geq 1/4 \times 1/4 = P_1(x=2) \\ P_1^\star(x=3) = 1/4 \geq 1/4 \times 1/4 = P_1(x=3) \\ P_1^\star(x=4) = 1/4 \geq 1/4 \times 1/4 = P_1(x=4) \\ P_1^\star(x=\_) = 0 \geq 0 \times 1/4 = P_1(x=\_) \\ \text{and} \\ P_1^\star(\lightning) = 0 \leq 1/4 = P_1(\lightning) \end{array}$$

In particular it is a *distribution sensitive* slice because it preserves the relative probabilities of the original program, or said otherwise, the output distribution of $P_1^\star$ is a scaled version of that of P_1. For this subclass of non-termination insensitive slicing, Equation 2 is refined to

$$\exists q \in [0,1].\ \mathcal{D}_P = q \cdot \mathcal{D}_{P'} \quad \text{and} \quad \mathcal{P}_P(\lightning) \geq \mathcal{P}_{P'}(\lightning) \tag{3}$$

Intuitively, the above equation holds when the removed program fragment is "probabilistically independent" of the rest of the program, and the scaling factor q —$\frac{1}{4}$ in the above example— coincides with the termination probability of the

program fragment being removed. This kind of slicing corresponds, indeed, to the one supported by Amtoft and Banerjee [2].

When the sliced program does not necessarily respect the relative probabilities of the original program, but does comply with Equation 2, we call it a *distribution insensitive* slice. This is the case, *e.g.*, for $P_2^\star$, and P_2.

6 Slicing Definition

In the context of probabilistic programs (PPs), slicing typically entails identifying segments of code that influence the evaluation of the return expression, particularly, the values of variables involved in this expression. The objective is to compute a slice that preserves only the portions relevant to the return expression, while maintaining the original program's relative distribution of return values as discussed in Sec. **??**. The slicing process is guided by a slicing criterion $C \subseteq N$. In the context of PP, C is typically a singleton set containing the CFG node representing the return statement. Although C could potentially encompass additional CFG nodes without any inherent limitation, we maintain this restriction for the sake of brevity. Dependence-based slicing algorithms typically operate at the CFG level and compute a slice set $slice_C$ as described below:

$$slice_C(G, \overset{cd}{\rightarrow}) = \bigcup_{n \in C} \{m : m(\overset{cd}{\rightarrow} \cup \overset{dd}{\rightarrow})^* n\} \tag{4}$$

where, $\overset{cd}{\rightarrow}$ is a suitable control dependence relation, $\overset{dd}{\rightarrow}$ is a data dependence relation, and $\rightarrow^*$ is the transitive reflexive closure of $\rightarrow$. This definition can precisely captures the slice of a *deterministic* program.The $slice_C$ function above is parametric to a control dependence relation $\overset{cd}{\rightarrow}$ which decides whether the slice is nontermination insensitive or sensitive. A strong (resp. weak) control dependence relation produces nontermination sensitive (resp. insensitive) slices.

However, the above equation is not sufficient and in some cases not correct in producing a correct slice of a *probabilistic* program. The ***observe*** statements and the nonterminating loops may affect the final distribution of a PP by introducing an special kind of dependency called *observe-nontermination dependence* [4] that cannot be captured by data and control dependencies only. In the next section, we illustrate this dependence through examples and provide its formal definition.

6.1 Observe-Nontermination dependence

Both an observe statement and a potentially nonterminating loop determine whether the execution should proceed beyond these instructions. An execution is either discarded if an observation failure occurs or remains stuck indefinitely if the loop does not terminate. Consequently, these two kinds of program instructions, which we collectively call observe-nontermination instruction, may

[4] may be we should come up with a better name like *distribution-aware dependence* or something else

impact the final distribution of a PP program. An observe-nontermination dependence is a conditional dependence between two CFG nodes n_0 and n_r such that n_o represents an observe-nontermination instruction, n_r is a CFG nodes in the slicing criterion C, and there exists a CFG node n that affects the execution of both n_o and n_r due to data and/or control dependencies. The following definition provides the formal treatment of observe-nontermination dependence:

Definition 6 (Observe-Nontermination dependence). *Let $G = (N, E, n_0)$ be any CFG, let $n_o \in N$ be an observe-nontermination instruction, and let $n_r \in C$ be any CFG node in the slicing criterion. An observe-Nontermination dependence relation $n_o \xrightarrow{obsntd} n_r$ holds iff there exists a CFG node $n \in N$ such that the reflexive transitive closure relations $n(\xrightarrow{cd} \cup \xrightarrow{dd})^* n_o$ and $n(\xrightarrow{cd} \cup \xrightarrow{dd})^* n_r$ hold.*

Thus, the relation $n_o \xrightarrow{obsntd} n_r$ represents a conditional dependence ▶ FO: *What do you mean by "conditional"? How is it different from simply "dependence"?* ◀ between n_o and n_r (i.e. n_o and n_r are not probabilisitcally independent) when a third CFG node n exists that affect the execution of both n_o and n_r. If the CFG node n_o in the above definition represents only the observe statement, then we call the dependence relation an observe dependence $n_o \xrightarrow{obsd} n_r$.

We demonstrate the above dependence relation with examples. Consider programs P_0, P_1 (Fig. 1) and P_3 (Fig. 2). Let n_i represents the CFG node of statement i in the given program. For programs P_0 and P_1, the relation $n_3 \xrightarrow{obsntd} n_4$ does not hold, as there is no CFG node n_j for any $1 \leq j \leq 3$ such that $n_j(\xrightarrow{cd} \cup \xrightarrow{dd})^* n_3$ and $n_j(\xrightarrow{cd} \cup \xrightarrow{dd})^* n_4$ hold. This is evident since variables x and y are independent in these programs. Consequently, we can slice out n_3 from a nontermination insensitive slice of P_0 and P_1 without affecting the final distribution modulo termination. This implication can be verified by proving that the outcome of any execution of n_3 and n_4 are probabilistically independent modulo termination ▶ FO: *What does "modulo termination" mean here? Probabilistic independence is unrelated to termination* ◀ .

Program P_0 has $4 \times 4 = 16$ distinct executions in which $4 \times 1 = 4$ executions are valid executions. The distribution of the return values of P_0 is represented by

$$\mathcal{D}_{P_0} = \{i \mapsto \frac{1}{4}, j \mapsto 0\}$$

REMARK: All calculations below are already present in Section 5. I wouldn't repeat them.

where $i, j \in \mathbb{Z}$, $1 \leq i \leq 4$, $j \neq i$. We calculate the probability

$$\mathcal{P}(x = i) = \frac{1 \times 2 \times 1/16}{8 \times 1/16} = \frac{1}{4}.$$

If we replace statement 3 with a skip statement, the probability will be altered to

$$\mathcal{P}'(x = i) = \frac{1 \times 4 \times 1/16}{16 \times 1/16} = \frac{1}{4}$$

which is equal to $\mathcal{P}(x = i)$. In case of program P_1, which is nonterminating when $y < 4$, we calculate the probability

$$\mathcal{P}(x = i) = \frac{1 \times 1 \times 1/16}{16 \times 1/16} = \frac{1}{16}.$$

The termination probability of this program is $\mathcal{P}(\lambda) = \frac{4 \times 1}{16} = \frac{1}{4}$. If the nonterminating loop is replaced by a skip statement, its probability for any value $x = i$ will be

$$\mathcal{P}'(x = i) = \frac{1 \times 4 \times 1/16}{16 \times 1/16} = \frac{1}{4},$$

and the following equality holds:

$$\mathcal{P}(x = i) = \mathcal{P}'(x = i) \times \mathcal{P}(\lambda).$$

In the case of program P_3, the relation $n_3 \xrightarrow{obsntd} n_4$ holds due to the relations $n_1 \xrightarrow{dd} n_3$ and $n_1 \xrightarrow{dd} n_4$. This implies that the outcome of any execution of n_3 and n_4 are not probabilistically independent. The probabilities of possible outcomes are calculated as follows:

$$\mathcal{P}(x = i) = \frac{1 \times 4 \times 1/16}{16 \times 1/16} = \frac{1}{4}$$

for $1 \leq i \leq 3$ and $\mathcal{P}(x \geq 4) = 0$ since P_3 does not terminate if $x = 4$ and the outcomes of it is in the range $1 \leq i \leq 3$. If we replace the nonterminating loop by a skip statement, we obtain the probabilities

$$\mathcal{P}(x = i) = \frac{1 \times 4 \times 1/16}{16 \times 1/16} = \frac{1}{4}$$

for $i = 1, \ldots, 4$. Consequently, we cannot prove the equality of the distributions (modulo termination) $\mathcal{D} = \{0 \mapsto \frac{1}{4}, \ldots, 3 \mapsto \frac{1}{4}, * \mapsto 0\}$ and $\mathcal{D}' = \{0 \mapsto \frac{1}{4}, \ldots, 4 \mapsto \frac{1}{4}, * \mapsto 0\}$. Therefore, a nontermination insensitive distribution sensitive slice of P_3 must include the nonterminating loop at statement 3. However, a nontermination insensitive distribution insensitive slice of P_3 may still slice away this nonterminating loop since if we assume that the nonterminating loop eventually terminates, then its distribution will be $\mathcal{D}'' = \{0 \mapsto \frac{1}{4}, \ldots, 4 \mapsto \frac{1}{4}, * \mapsto 0\}$ which is equal to $\mathcal{D}'$

TODO: Maybe it is a good idea to discuss also the first program in Section 10.1, to illustrate the v-shape thing for observe statements.

6.2 Computing various slices

Eq. 5 presented below extends Eq. 4 to compute the slice set for various types of slices of a given PP program:

REMARK: Relation R should be parametrized by cd.

$$slice_C(G, \xrightarrow{cd}, \xrightarrow{R}) = \bigcup_{n \in C} \{m : m(\xrightarrow{cd} \cup \xrightarrow{dd})^* n\} \cup \bigcup_{n_o \mid \exists n_r \in C.\, n_o \xrightarrow{R} n_r} \{n : n(\xrightarrow{cd} \cup \xrightarrow{dd})^* n_o\} \tag{5}$$

Slicing class	cd	R
non-termination sensitive	scd	$obsntd$
non-termination insensitive, distribution sensitive	wcd	$obsntd$
non-termination insensitive, distribution insensitive	wcd	$obsd$

Table 1: Dependence relation used for defining each class of slicing.

The equation above is parameterized by the control dependency relation $\xrightarrow{cd}$, which may manifest as either a weak or strong control dependency relation $\xrightarrow{wcd}$ or $\xrightarrow{scd}$ respectively. Additionally, it involves the relation $\xrightarrow{R}$, which can either signify the *observe-nontermination dependency* relation $\xrightarrow{obsntd}$ or only the *observe dependency* relation $\xrightarrow{obsd}$. To recover each class of slicing, relations cd and R must be instantiated as summarized in Table 1.

For the nontermination and distribution insensitive slice, we disregard the impact of nontermination entirely, as the semantics assume that all nonterminating loops eventually terminate in both the original program and the slice. Consequently, this type of slice set is determined by the relation $slice_C(G, \xrightarrow{wcd}, \xrightarrow{obsd})$. Conversely, for the nontermination insensitive distribution sensitive slice, we focus solely on nonterminating loops that influence the final distribution, disregarding those that are probabilistically independent of the return statement in a semantic context. Hence, the relation $slice_C(G, \xrightarrow{wcd}, \xrightarrow{obsntd})$ computes these slices by encompassing the following semantics: (1) the relation $\xrightarrow{wcd}$ disregards all nontermination effects while calculating the data and control dependency in Eq. 5, and (2) the relation $\xrightarrow{obsntd}$ selectively include all nonterminating instructions that affect the final distribution at the CFG node n_r. The nontermination sensitive slice is computed by the relation $slice_C(G, \xrightarrow{scd}, \xrightarrow{obsntd})$, which captures the effect of nontermination during the computation of the normal data and control dependency relation as well as during capturing the effect of the dependency due to *observe-nontermination* instruction.

Todo: For the sake of self-containedness, *wcd*, *scd* and *dd* should be defined (I guess these defs are used in the soundness proof, too.)

7 Slicing Correctness

In this section, we develop theories and a proof framework for proving the correctness of the slice P_2 of a probabilistic program P_1. We formally express this correctness criteria by introducing some mathematical notations.

Let $\Gamma \in \mathcal{N} \times D^{\leq 1}(\Sigma)$ be a program configuration expressing the fact that the execution is at CFG node n having the probabilistic store $\mu \in D^{\leq 1}(\Sigma)$ for

any $\Gamma = (n, \mu)$. We sometimes write $node(\Gamma) = n$ and $store(\Gamma) = \mu$ for any $\Gamma = (n, \mu)$. Let $V \subseteq \mathcal{V}$ be a subset of variables and let $\sigma \in \Sigma$. The restriction of the store σ on V denoted $\sigma|_V$ is defined as $\sigma|_V = \cup_{x \in V}\{x \mapsto \sigma(x)\}$. Let $\Sigma_V : V \to \mathbb{Z}$ be a restriction of Σ such that $\sigma|_V \in \Sigma_V$ is the restriction of $\sigma \in \Sigma$. The projection of μ on V denoted $\mu|_V$ is defined as $\mu|_V(\sigma_V) = \sum_{\substack{\sigma \in \Sigma \\ \sigma_V = \sigma|_V}} \mu(\sigma)$

In what follows, we assume that P_1 is the original program and P_2 is the slice computed according to Def. 13 from the slice set S_C which is computed using any dependence-based slicing algorithm such as Alg. 3. We say that S_C is closed under the relations $\xrightarrow{dd}$, $\xrightarrow{cd}$, and $\xrightarrow{R}$ when it is computed according to Eq. 5.

7.1 Correctness Theorems

Definition 7 (Equivalence of probabilistic stores). *Let $V \subseteq \mathcal{V}$ be a subset of variables, and let $\mu_1, \mu_2 \in \mathcal{D}^{\leq 1}(\Sigma)$ be pairs of probabilistic stores. Let $\simeq$ be any of the following relations over probability stores: (i) equality relation (i.e. $\mu_1 = \mu_2$), (ii) equality upto a scaling factor $q \in \mathbb{Q}$ denoted $\mu_1 =_q \mu_2$ if $\mu_1 = q \times \mu_2$), and (iii) less or equal upto a scaling factor q denoted $\mu_1 \leq_q \mu_2$ if $\mu_1 \leq q \times \mu_2$. We say that μ_1 and μ_2 are* equivalent modulo relation $\simeq$ with respect to V, *written $\mu_1 \simeq^V \mu_2$, iff $\mu_1|_V \simeq \mu_2|_V$ holds.*

We now introduce the notion of *next observable nodes* which was originally introduced within the realm of non-probabilistic programs. As we traverse the CFG and encounter a CFG node, it becomes relevant to identify the first reachable CFG nodes from the slice set along any CFG path. From an execution perspective, this translates to determining the potential program instructions to be executed next in the slice. The *next observable nodes* can be renamed to the *next sliced-node to be visited*. However, we keep the *next observable* term for historical reason and ask the readers not to be confused with the ***observe*** instruction. The formal definition of next observable nodes is as follows:

Definition 8 (Next Observable). *Let n be a node in CFG G, and let S_C be a slice set. The set of next* observable nodes *$obs_{S_C}(n)$ contains all nodes $m \in S_C$ such that there exists a valid CFG path $[n_1..n_k]$ with $n_1 = n$, $n_k = m$, and we must have $n_i \notin S_C$ for $1 \leq i \leq k-1$.*

For every semantic transition $P_i \vdash \Gamma \to \Gamma'$ for $i = 1, 2$, we establish labeled transitions $P_i \vdash \Gamma \xrightarrow{l} \Gamma'$, where the label l is either a next observable node n as defined in Def. 8 or the symbol τ. In the former case, P_i signifies an observable move, while in the latter case, it denotes a silent move.

Definition 9 (Labeled Transition). *Let S_C be the slice set for the slice P_2 of the original program P_1. For all configurations Γ_1 and Γ_2 of program P_i for $i = 1, 2$ such that $P_i \vdash \Gamma_1 \to \Gamma_2$, we define*

- $P_i \vdash \Gamma_1 \xrightarrow{n} \Gamma_2$ *if $n = node(\Gamma_1)$ and $n \in S_C$*
- $P_i \vdash \Gamma_1 \xrightarrow{\tau} \Gamma_2$ *otherwise.*

We write:

- $P_i \vdash \Gamma_1 \overset{\tau}{\Rightarrow} \Gamma_2$ *for the reflexive transitive closure of* $P_i \vdash \Gamma_1 \overset{\tau}{\rightarrow} \Gamma_2$
- $P_i \vdash \Gamma_1 \overset{n}{\Rightarrow} \Gamma_2$ *if there exists a configuration* Γ *such that* $P_i \vdash \Gamma_1 \overset{\tau}{\Rightarrow} \Gamma$ *and* $P_i \vdash \Gamma \overset{n}{\rightarrow} \Gamma_2$

The observable transition $\overset{n}{\rightarrow}$ requires that $n \in S_C$ and thus it affects the slicing criterion. We can now use the definition of labeled transition to define the weak (bi)simulation as follows:

Definition 10 (Weak Simulation and Bisimulation). *Consider the following properties for relation* Φ*:*

(i) if $\Gamma_1 \Phi \Gamma_2$ *and* $P_1 \vdash \Gamma_1 \overset{n}{\Rightarrow} \Gamma_1'$*, then there exists* Γ_2' *such that* $\Gamma_1' \Phi \Gamma_2'$ *and* $P_2 \vdash \Gamma_2 \overset{n}{\Rightarrow} \Gamma_2'$*.*

(ii) if $\Gamma_1 \Phi \Gamma_2$ *and* $P_2 \vdash \Gamma_2 \overset{n}{\Rightarrow} \Gamma_2'$*, then there exists* Γ_1' *such that* $\Gamma_1' \Phi \Gamma_2'$ *and* $P_1 \vdash \Gamma_1 \overset{n}{\Rightarrow} \Gamma_1'$*.*

Φ *is a* weak simulation *if (i) holds, and a* weak bisimulation *if both (i) and (ii) hold.*

We now define the relation $\overset{seq}{\simeq}$ between the configurations of an original program P_1 and its slice P_2 as follows:

Definition 11 ($\overset{seq}{\simeq}$). *Let* S_C *be the slice set for the slice* P_2 *of the original program* P_1*, and let* V *represent the set of variables of* P_2*, which is a subset of variables of* P_1*. Suppose* $\Gamma_1 = (n_1, \mu_1)$ *and* $\Gamma_2 = (n_2, \mu_2)$ *are valid configurations of programs* P_1 *and* P_2 *respectively. The relation* $\Gamma_1 \overset{seq}{\simeq} \Gamma_2$ *holds if the following conditions are met:*

1. $obs(n_1) = obs(n_2)$*, and*
2. $\mu_1 \simeq^V \mu_2$*, where the relation* $\simeq$ *is specified in Def. 7.*

Theorem 1 below states that $\overset{seq}{\simeq}$ is either a weak simulation or a weak bisimulation, depending on the dependece relations used to compute the slice set S_C:

Theorem 1 (Correctness Condition). *Assume that* S_C *is computed according to Eq. 5. The relation* $\overset{seq}{\simeq}$ *is a weak bisimulation relation if* S_C *is closed under* $\xrightarrow{dd}$*,* $\xrightarrow{scd}$*, and* $\xrightarrow{obsntd}$*, and it is a weak simulation relation otherwise.*

Theorem 2 stated below ensures that the slice P_2 is a correct nontermination (in)sensitive distribution (in)sensitive slice of P_1 if $\overset{seq}{\simeq}$ is a weak (bi)simulation due to the slice set S_C computed according to the conditions stated in Theorem 1.

Theorem 2 (Correctness).

1. P_2 *is a nontermination sensitive slice of* P_1 *if* S_C *is closed under* $\overset{dd}{\rightarrow}$*,* $\overset{scd}{\rightarrow}$ *and* $\xrightarrow{obsntd}$*.*

2. P_2 *is a nontermination insensitive distribution sensitive slice of* P_1 *if* S_C *is closed under* $\xrightarrow{dd}$, $\xrightarrow{wcd}$ *and* $\xrightarrow{obsntd}$.
3. P_2 *is a nontermination insensitive distribution insensitive slice of* P_1 *if* S_C *is closed under* $\xrightarrow{dd}$, $\xrightarrow{wcd}$ *and* $\xrightarrow{obsd}$.

7.2 Proof of theorems

We need some auxiliary lemmas to prove the theorems.

Lemma 1. *Given a slice set* S_C *closed under* $\xrightarrow{cd}$, *the* $obs(n)$ *set of any CFG node* n *contains at most one element.*

Proof. According to Def. 8, $obs(n) = \{n\}$ if $n \in S_C$, $obs(n) = \emptyset$ if no CFG path from n includes a node from S_C, and the lemma is satisfied in both cases.

Assume $n \notin S_C$ and suppose, contrary to the lemma, that there are distinct nodes n_k and n_l such that both belong to $obs(n)$. This implies the existence of two distinct CFG paths: $[n_1 = n..n_k]$ and $[n^1 = n..n^l = n_l]$ such that $n_i, n^j \notin S_C$ for each $1 \leq i < k$ and $1 \leq j < l$, and $n_k, n_l \in S_C$. Node n_k (resp. n_l) postdominates all nodes between n_1 and n_k (resp. n_l). Given that n has another branch to node n_l (resp. n_k) and thus n is not strictly postdominated by either n_k or n_l, either $n \xrightarrow{cd} n_k$ or $n \xrightarrow{cd} n_l$ holds. In any case, we get the contradiction that $n \in S_C$ since S_C is closed under $\xrightarrow{cd}$. Therefore, our initial assumption that $obs(n)$ contains more than one element when $n \notin S_C$ is false, and the lemma is proven. □

In what follows, we will abuse the notation and write $obs(n) = m$ for $obs(n) = \{m\}$.

Lemma 2. *Let* $P \vdash \langle n_1, \mu_1\rangle \rightarrow \langle n_2, \mu_2\rangle$ *be a semantic transition of program* P. *If* $code(n_1)$ *is* ***skip****, true, or false , then we must have* $\mu_1 = \mu_2$.[5]

Proof. According to the SKIP, TRUE, and FALSE semantic rules in Fig. 3, $\mu_1 = \mu_2$ trivially holds. □

Lemma 3. *Let* $P_1 \vdash \langle n, \mu_1\rangle \xrightarrow{n} \langle n', \mu_1'\rangle$ *and* $P_2 \vdash \langle n, \mu_2\rangle \xrightarrow{n} \langle n', \mu_2'\rangle$ *be two labelled transitions of* P_1 *and its slice* P_2, *and let* V *be the set of program variables in* P_2. *If* $\mu_1 \simeq^V \mu_2$ *holds, then* $\mu_1' \simeq^V \mu_2'$ *holds as well.*

Proof. [6] It is evident from the labelled transitions in the premise of the lemma that $n \in S_C$. Thus, $code_1(n) = code_2(n)$. We infer the relation between μ_1' and μ_2' from the relation $\mu_1 \simeq_V \mu_2$ by analyzing the semantic rules in Fig. 3 as follows:

– $code_1(n)$ is a normal or probabilistic assignment to a variable $x \in V$. According to the ASSIGN or DIST rules, $\mu_1' \simeq^V \mu_2'$ immediately follows from $\mu_1 \simeq^V \mu_2$.

[5] to be checked by Federico

[6] TODO Federico: it's a proof sketch, we need to complete this proof.

– Let $code_1(n) =$ ***Observe*** b. If the probabilistic store μ_1 evaluates b true, then so is P_2 due to the relation $\mu_1 \simeq^V \mu_2$. Consequently, $\mu_1' \simeq^V \mu_2'$ holds according to the OBSERVE Rule in Fig. 3.
– $code_1(n) = b$. $\mu_1' \simeq^V \mu_2'$ follows from $\mu_1 \simeq^V \mu_2$ since μ_1' is a subset of μ_i for $i = 1, 2$.
– $code_1(n) =$***skip***. Then, $\mu_1' \simeq^V \mu_2'$ trivially holds since $\mu_1 = \mu_1'$ and $\mu_2 = \mu_2'$

□

Lemma 4. *Let S_C be the slice set which is closed under $\overset{cd}{\rightarrow}$, and let Γ_1 be any configuration of program P_2 at CFG node n_1. If $obs(n_1) = n_k$, then there exists configuration Γ_k of node n_k such that $P_2 \vdash \Gamma_1 \overset{\tau}{\Rightarrow} \Gamma_k$.*

Proof. Suppose $obs(n_1) = n_k$. Consequently, there exists a CFG path $n_1, \ldots, n_k$ such that $n_k \in S_C$ and $n_i \notin S_C$ for $1 \leq i < k$. Let's assume, without loss of generality, that this path is the smallest one. Since $n_i \notin S_C$ for $1 \leq i < k$, $code_2(n_i)$ must be ***skip***, *true*, or *false* as per Def. 13. If $code_2(n_i)$ is *true* (or *false*), then node n_{i+1} is in the true (resp. false) branch of node n_i, i.e., $succ_T(n_i) = n_{i+1}$ (resp. $succ_F(n_i) = n_{i+1}$). According to the semantic rules illustrated in Fig. 3, there exists a sequence of configurations $\Gamma_1, \ldots, \Gamma_k$ for the nodes $n_1, \ldots, n_k$ exists such that $P_2 \vdash \Gamma_i \Rightarrow \Gamma_{i+1}$ for $1 \leq i < k$. As $n_i \notin S_C$ for $1 \leq i < k$, we obtain the labelled transition $P_2 \vdash \Gamma_i \overset{\tau}{\rightarrow} \Gamma_{i+1}$ for $1 \leq i < k$. Consequently, $P_2 \vdash \Gamma_1 \overset{\tau}{\Rightarrow} \Gamma_k$ holds. □

Lemma 5. *Let Γ_1 and Γ_2 be valid configurations of programs P_1 and P_2 such that $\Gamma_1 \overset{seq}{\simeq} \Gamma_2$. If there exists a transition $P_1 \vdash \Gamma_1 \overset{n_1}{\rightarrow} \Gamma_3$, then a transition $P_2 \vdash \Gamma_2 \overset{n_1}{\Rightarrow} \Gamma_4$ also exists such that $\Gamma_3 \overset{seq}{\simeq} \Gamma_4$.*

Proof. Let $\Gamma_i = \langle n_i, \mu_i \rangle$ for $1 \leq i \leq 4$. First, we prove that the labeled transition $P_2 \vdash \Gamma_2 \overset{n_1}{\Rightarrow} \Gamma_4$ exists. The transition $P_1 \vdash \Gamma_1 \overset{n_1}{\rightarrow} \Gamma_3$ yields $n_1 \in S_C$, and consequently, $obs(n_1) = n_1$. The relation $\Gamma_1 \overset{seq}{\simeq} \Gamma_2$ yields $obs(n_1) = obs(n_2) = n_1$.

Let $n_2 \neq n_1$. By Lemma 4, $obs(n_2) = n_1$ indicates the existence of a configuration Γ^k of n_1 such that $P_2 \vdash \Gamma_2 \overset{\tau}{\Rightarrow} \Gamma^k$. Thus, a sequence of configurations $\Gamma^1 = \langle n^1, \mu^1 \rangle, \ldots, \Gamma^k = \langle n^k, \mu^k \rangle$ of P_2 exists where $\Gamma^1 = \Gamma_2$ and $n^k = n_1$. The relation $P_2 \vdash \Gamma^i \overset{\tau}{\rightarrow} \Gamma^{i+1}$ holds for all $i = 1, \ldots, k-1$, implying $n^i \notin S_C$ and $code_2(n^i)$ is ***skip***, *true*, or *false* as per Def. 13. Consequently, $\mu^i = \mu^{i+1}$ for all $1 \leq i \leq k$ due to Lemma 2. As $\Gamma_1 \overset{seq}{\simeq} \Gamma_2$, we have $\mu_1 \simeq^V \mu_2$ where V is the set of program variables in P_2 which is a subset of program variables in P_1. Thus, $\mu_1 \simeq^V \mu^k$. If $n_2 = n_1$, then $\Gamma_2 = \Gamma^k$ and $\mu_1 \simeq^V \mu^k$ trivially hold due to the precondition of the lemma. Hence, either $\mu_1|_V = \mu^k|_V$ or $\mu_1|_V \leq \mu^k|_V$.

Therefore, the execution of P_1 and P_2 converge at n_1, and the relative probability distribution $\mu^k|_V$ of P_2 at n_1 is at least equal to the relative probability distribution $\mu_1|_V$ of P_1. Consequently, P_2 must take the same path as P_1 from n_1 even if n_1 is a predicate node or represents an ***observe*** statement, as P_2 must evaluate any conditional expression at n_1 to the same values as P_1 due to the relation $\mu_1|_V \leq \mu^k|_V$. Thus, if $P_1 \vdash \Gamma_1 \overset{n_1}{\rightarrow} \Gamma_3$ exists in P_1, a configuration

$\Gamma_4 = \langle n_3, \mu_4 \rangle$ also exists such that $n_4 = n_3$, and we get the transition $\Gamma^k \to \Gamma_4$ in P_2. Since $n_1 \in S_C$, we thus have $P_2 \vdash \Gamma^k \overset{n_1}{\to} \Gamma_4$, and consequently, we get the transition $P_2 \vdash \Gamma_2 \overset{n_1}{\Rightarrow} \Gamma_4$ in P_2.

As (i) $obs(n_3) = obs(n_3)$ holds trivially, and (ii) $\mu_3 \simeq^V \mu_4$ according to Lemma 3, the relation $\Gamma_3 \overset{seq}{\simeq} \Gamma_4$ holds according to the definitions of $\overset{seq}{\simeq}$. □

Lemma 6. *Assume that S_C is closed under $\xrightarrow{scd}$ and $\xrightarrow{obsntd}$. Let Γ_1 be any configuration of program P_1 at CFG node n_1, and let S_C be closed under the relation $\overset{scd}{\to}$. If $obs(n_1) = n_k$ and $n_1 \neq n_k$, then there exists configuration Γ_k of node n_k such that $P_1 \vdash \Gamma_1 \overset{\tau}{\Rightarrow} \Gamma_k$ and $store(\Gamma_1) =^V store(\Gamma_k)$ where V is the set of program variables at P_2.*

Proof. Assume that $obs(n_1) = n_k$. Thus, there exists a CFG path $n_1, \ldots, n_k$ such that $n_i \notin S_C$ for all $1 \leq i < k$. If there exists a configuration Γ_i of node n_i which is an assignment or skip instruction, then there always exists a configuration Γ_{i+1} of n_{i+1} such that $P_1 \vdash \Gamma_i \Rightarrow \Gamma_{i+1}$. In cases where n_i represents an observe instruction, there exists a transition $P_1 \vdash \Gamma_i \Rightarrow \Gamma_{i+1}$ for some valuation of the program variables in P_1; otherwise, the observe instruction is equivalent to ***Observe****(false)*, leading to a final distribution $\frac{0}{0}$, which renders **program** P_1 **nonsensical.** If n_i represents a predicate node having two branches n_{i+1} and n', then n' does not lead to nontermination and all CFG paths from n' must converge at n^j for some $i + 1 < j \leq k$; otherwise, n_i would be in S_C due to the relation $\xrightarrow{scd}$. Therefore, we must have the transition $P_1 \vdash \Gamma_i \Rightarrow \Gamma_j$ where either $j = i + 1$ or $i + 1 < j \leq k$. This implies that we have the labelled transition $P_1 \vdash \Gamma_1 \overset{\tau}{\Rightarrow} \Gamma_k$. Also, $store(\Gamma_1) =^V store(\Gamma_k)$ due to the following: (i) any (random) assignment to a variable in V at any node n_i that is used in n_k or any other node in S_C would create a data dependency, and n_i would be included in S_C, (ii) in instances where node n_i modifies a variable in V, it is subsequently overwritten at a node within S_C; consequently, any alteration in the store becomes inconsequential, and the update in the store can be disregarded, and (iii) if n_i represents an ***Observe***(e) statement, then variables in e are not part of V, thereby not impacting the distributions in $store(\Gamma_k)|_V$, as otherwise n_i would be included in S_C due to the relation $\xrightarrow{obsntd}$. □

Lemma 7. *Assume that S_C is closed under $\overset{scd}{\to}$ and and $\xrightarrow{obsntd}$. Let Γ_1 and Γ_2 be valid configurations of programs P_1 and P_2 such that $\Gamma_1 \overset{seq}{\simeq} \Gamma_2$. If there exists a transition $P_2 \vdash \Gamma_2 \overset{n_2}{\to} \Gamma_4$, then a transition $P_1 \vdash \Gamma_1 \overset{n_2}{\Rightarrow} \Gamma_3$ also exists such that $\Gamma_3 \overset{seq}{\simeq} \Gamma_4$.*

Proof. Let $\Gamma_i = \langle n_i, \mu_i \rangle$ for $1 \leq i \leq 4$. First, we prove that the labeled transition $P_1 \vdash \Gamma_1 \overset{n_2}{\Rightarrow} \Gamma_3$ exists. The transition $P_2 \vdash \Gamma_2 \overset{n_2}{\to} \Gamma_4$ yields $n_2 \in S_C$, and consequently, $obs(n_2) = n_2$. The relation $\Gamma_1 \overset{seq}{\simeq} \Gamma_2$ yields $obs(n_1) = obs(n_2) = n_2$.

Assume that $n_1 \neq n_2$. As $obs(n_1) = n_2$, by Lemma 6, there exists a configuration Γ^k of n_2 such that the labelled transition $P_1 \vdash \Gamma_1 \overset{\tau}{\Rightarrow} \Gamma^k$ holds. Moreover,

$\mu_1 =^V store(\Gamma^k)$, and consequently, $store(\Gamma^k) \simeq^V \mu_2$ holds due to $\mu_1 \simeq^V \mu_2$ where V is the set (and a subset) of program variables in P_2 (resp. P_1). If $n_2 = n_1$, then $\Gamma_1 = \Gamma^k$ and $store(\Gamma^k) \simeq^V \mu_2$ holds due to the precondition of the lemma. Therefore, the execution of P_1 and P_2 converge at n_2, and the relative probability distribution $\mu^k|_V$ of P_1 at n_2 is at least equal to the relative probability distribution $\mu_2|_V$ of P_2. Consequently, P_1 must take the same path as P_2 from n_2 even if n_2 is a Cond node or represents an ***Observe*** statement, as P_1 must evaluate any conditional expression at n_2 to the same values as P_2 due to the relation $\mu_1|_V = \mu^k|_V$. Thus, $P_1 \vdash \Gamma_1 \xrightarrow{n_2} \Gamma_3$ exists in P_1, and a configuration $\Gamma_3 = \langle n_3, \mu_3 \rangle$ also exists such that $n_3 = n_4$, and we get the transition $\Gamma^k \rightarrow \Gamma_3$ in P_1. Since $n_2 \in S_C$, we thus have $P_1 \vdash \Gamma^k \xrightarrow{n_2} \Gamma_3$, and consequently, we get the transition $P_1 \vdash \Gamma_1 \overset{n_2}{\Rightarrow} \Gamma_3$.

As (i) $obs(n_3) = obs(n_3)$ holds trivially, and (ii) $\mu_3 \simeq^V \mu_4$ according to Lemma 3, the relation $\Gamma_3 \overset{seq}{\simeq} \Gamma_4$ holds according to the definitions of $\overset{seq}{\simeq}$. □

Lemma 8. *Assume that S_C is closed under $\xrightarrow{scd}$ and and $\xrightarrow{obsntd}$. Let Γ_1 and Γ_2 be valid configurations of programs P_1 and P_2 such that $\Gamma_1 \overset{seq}{\simeq} \Gamma_2$. If there exists a transition $P_2 \vdash \Gamma_2 \overset{n}{\Rightarrow} \Gamma_4$, then a transition $P_1 \vdash \Gamma_1 \overset{n}{\Rightarrow} \Gamma_3$ also exists such that $\Gamma_3 \overset{seq}{\simeq} \Gamma_4$.*

Proof. If $n_2 = n$, then the lemma holds due to Lemma 7. Assume that $n_2 \neq n$ and $P_2 \vdash \Gamma_2 \overset{n}{\Rightarrow} \Gamma_4$. Thus, a sequence of silent transitions $P_2 \vdash \Gamma^i \xrightarrow{\tau} \Gamma^{i+1}$ for $1 \leq i < k$ exists, along with the transition $P_2 \vdash \Gamma^k \xrightarrow{n} \Gamma_4$ such that $\Gamma^1 = \Gamma_1$. Let $\Gamma_i = \langle n_i, \mu_i \rangle$ for $1 \leq i \leq 4$, and let $\Gamma^i = \langle n^i, \mu^i \rangle$ for $1 \leq i \leq k$. Firstly, we establish the relation $\Gamma_1 \overset{seq}{\simeq} \Gamma^k$.

The silent transition $P_2 \vdash \Gamma^i \xrightarrow{\tau} \Gamma^{i+1}$ implies $n^i \notin S_C$. The transition $P_2 \vdash \Gamma^k \xrightarrow{n} \Gamma_4$ indicates that $n^k = n \in S_C$. Consequently, $obs(n_2) = obs(n^k) = n$. The relation $\Gamma_1 \overset{seq}{\simeq} \Gamma_2$ implies $obs(n_1) = obs(n_2)$, and hence $obs(n_1) = obs(n^k)$. This proves the first precondition for the relation $\Gamma_1 \overset{seq}{\simeq} \Gamma^k$. We now establish the relation $\mu_1 \simeq^V \mu^k$ where V is the set (and a subset) of program variables in P_2 (resp. P_1).

Since $obs(n_2) = n = n^k$, all CFG paths from n_2 must converge at n^k without going through S_C. For any CFG node n' in any such path, $code_2(n')$ is ***skip***, *true* , or *false* . By successive application of Lemma 2, we obtain $\mu_2 = \mu^k$. Therefore, $\mu_1 \simeq^V \mu^k$ yields from the relation $\mu_1 \simeq^V \mu_2$, and consequently, $\Gamma_1 \overset{seq}{\simeq} \Gamma^k$ holds. As $P_2 \vdash \Gamma^k \xrightarrow{n} \Gamma_4$ exists, the lemma is proven by lemma 7.

□

Corollary 1. *Assume the transitions $P_2 \vdash \Gamma_2 \overset{n}{\Rightarrow} \Gamma_4$ and $P_1 \vdash \Gamma_1 \overset{n}{\Rightarrow} \Gamma_3$ such that $\Gamma_1 \overset{seq}{\simeq} \Gamma_2$ and $\Gamma_3 \overset{seq}{\simeq} \Gamma_4$ hold. If $store(\Gamma_1) =^V store(\Gamma_2)$ and S_C is closed under $\xrightarrow{scd}$ and and $\xrightarrow{obsntd}$, then $store(\Gamma_3) =^V store(\Gamma_4)$.*

Lemma 9. *Assume that S_C is closed under $\xrightarrow{dd}$, $\xrightarrow{wcd}$, and either $\xrightarrow{obsntd}$ or $\xrightarrow{obsd}$. Let Γ_1 and Γ_2 be valid configurations of programs P_1 and P_2 such that*

$\Gamma_1 \overset{seq}{\simeq} \Gamma_2$. *If there exists a transition* $P_1 \vdash \Gamma_1 \overset{n}{\Rightarrow} \Gamma_3$, *then a transition* $P_2 \vdash \Gamma_2 \overset{n}{\Rightarrow} \Gamma_4$ *also exists such that* $\Gamma_3 \overset{seq}{\simeq} \Gamma_4$.

Proof. Let's consider that the transition $P_1 \vdash \Gamma_1 \overset{n}{\Rightarrow} \Gamma_3$ exists. It implies that a sequence of silent transitions $P_1 \vdash \Gamma^i \overset{\tau}{\rightarrow} \Gamma^{i+1}$ for $1 \leq i < k$ also exist, along with the transition $P_1 \vdash \Gamma^k \overset{n}{\rightarrow} \Gamma_3$ such that $\Gamma^1 = \Gamma_1$. Let $\Gamma_i = \langle n_i, \mu_i \rangle$ for $1 \leq i \leq 4$, and let $\Gamma^i = \langle n^i, \mu^i \rangle$ for $1 \leq i \leq k$. Firstly, we establish the relation $\Gamma^k \overset{seq}{\simeq} \Gamma_2$.

The silent transition $P_1 \vdash \Gamma^i \overset{\tau}{\rightarrow} \Gamma^{i+1}$ implies $n^i \notin S_C$. The transition $P_1 \vdash \Gamma^k \overset{n}{\rightarrow} \Gamma^3$ indicates that $n^k = n \in S_C$. Consequently, $obs(n_1) = obs(n^k) = n$. The relation $\Gamma_1 \overset{seq}{\simeq} \Gamma_2$ implies $obs(n_1) = obs(n_2)$, and hence $obs(n^k) = obs(n_2)$. This proves the first precondition for the relation $\Gamma^k \overset{seq}{\simeq} \Gamma_2$. We now establish the relation $\mu^k \simeq^V \mu_2$ where V is the set (and a subset) of program variables in P_2 (resp. P_1).

Assume that no predicate node n^i exists in the path $n^1, \ldots, n^k$. Then, $store(\Gamma^1) =^V store(\Gamma^k)$ holds due to the following: (i) any (random) assignment to a variable in V at any node n^i that is used in n^k or any other node in S_C would create a data dependency, and n^i would be included in S_C, (ii) in instances where node n^i modifies a variable in V, it is subsequently overwritten at a node within S_C; consequently, any alteration in the store becomes inconsequential, and the update in the store can be disregarded, (iii) if n^i represents an ***Observe***(e) statement, then variables in e are not part of V, thereby not impacting the distributions in $store(\Gamma^k)|_V$, as otherwise n^i would be included in S_C due to the relation $\xrightarrow{obsntd}$ or $\xrightarrow{obsd}$.

In cases where node n^i represents a predicate node with successors n^{i+1} and n', the branch from n' either diverges or both branches converge at a node n^j for any $i+1 \leq j \leq k$ without going through a node in S_C, as otherwise we would have $n^i \in S_C$ due to the relation $\xrightarrow{cd}$. Then, one of the following must hold:

- If both the branches converge at n^j, and since $n^i \notin S_C$, $store(\Gamma^1) =^V store(\Gamma^k)$ holds.
- If S_C is closed under $\xrightarrow{scd}$ and there exists a diverging branch at n^i, this divergence would be captured by including n^i in S_C. As a result, we would obtain $i = k$ and $store(\Gamma^1) =^V store(\Gamma^k)$ holds.
- If S_C is closed under $\xrightarrow{wcd}$ and $\xrightarrow{obsntd}$, then the divergence will be captured only by including n^i in S_C if the diverging loop condition is affected by a node in S_C due to $(\xrightarrow{wcd} \cup \xrightarrow{dd})^*$. If all such divergences are captured, $store(\Gamma^k) =^V store(\Gamma^1)$ holds; otherwise, some distributions diminishes in the diverging branch and $store(\Gamma^k) \leq^V store(\Gamma^1)$ holds. In any case, the relation $store(\Gamma^k) =^V_q store(\Gamma^1)$ holds for any $q \in [0, 1]$.
- If S_C is closed under $\xrightarrow{wcd}$ and $\xrightarrow{obsd}$ and there exists a diverging branch at n^i, we obtain $store(\Gamma^k) \leq^V store(\Gamma^1)$.

So, any change in the probabilistic stores in μ^i along the CFG path $n^1, \ldots, n_3$ may influence the probabilistic store at μ_3 for the subset V of program variables in P_1 if there exists a diverging branch that is not captured in S_C. Nevertheless, the relation $store(\Gamma^k) \simeq^V store(\Gamma^1)$ holds which yields the relation $\mu^k \simeq^V \mu_2$ from the relation $\mu_1 \simeq^V \mu_2$. Consequently, $\Gamma^k \overset{seq}{\simeq} \Gamma_2$ holds. As $P_1 \vdash \Gamma^k \overset{n}{\to} \Gamma_3$ exists, the lemma is proven by lemma 5. □

Corollary 2. *Assume the transitions $P_2 \vdash \Gamma_2 \overset{n}{\Rightarrow} \Gamma_4$ and $P_1 \vdash \Gamma_1 \overset{n}{\Rightarrow} \Gamma_3$ such that $\Gamma_1 \overset{seq}{\simeq} \Gamma_2$ and $\Gamma_3 \overset{seq}{\simeq} \Gamma_4$ hold. Then, the following must hold:*

1. *If $store(\Gamma_1) \leq^V store(\Gamma_2)$ and S_C is closed under $\overset{wcd}{\to}$ and $\xrightarrow{obsd}$, then $store(\Gamma_3) \leq^V store(\Gamma_4)$.*
2. *If $store(\Gamma_1) =^V_q store(\Gamma_2)$ for some $q \in [0,1]$ and S_C is closed under $\overset{wcd}{\to}$ and $\xrightarrow{obsntd}$, then $store(\Gamma_3) = q'^V store(\Gamma_4)$ for some $q' \in [0,1]$.*

Proof (Theorem 1). Let Γ_1 and Γ_2 be valid configurations of programs P_1 and P_2, and let $\Gamma_1 \overset{seq}{\simeq} \Gamma_2$ holds.

1. Assume that there exists a labeled transition $P_1 \vdash \Gamma_1 \overset{n}{\Rightarrow} \Gamma_1'$. By lemma 9, $P_2 \vdash \Gamma_2 \overset{n}{\Rightarrow} \Gamma_2'$ also exists such that $\Gamma_1' \overset{seq}{\simeq} \Gamma_2'$. This proves that $\overset{seq}{\simeq}$ is a simulation relation.
2. The proof in (1) is one direction of the bisimulation as it also holds when S_C is closed under $\xrightarrow{scd}$, and $\xrightarrow{obsntd}$. For the other direction, let us assume that the labeled transition $P_2 \vdash \Gamma_2 \overset{n}{\Rightarrow} \Gamma_4$. By Lemma 8, a transition $P_1 \vdash \Gamma_1 \overset{n}{\Rightarrow} \Gamma_3$ also exists such that $\Gamma_3 \overset{seq}{\simeq} \Gamma_4$.

□

Proof (Theorem 2). Let Γ_0 and Γ_k (Γ_0' and Γ_k') be the initial and final configurations of P_1 (resp. P_2). Let V be the set (and subset) of program variables in P_2 (resp. P_1).

1. Let's assume that S_C is closed under $\overset{dd}{\to}$, $\overset{scd}{\to}$ and $\xrightarrow{obsntd}$. Then, according to Theorem 1, $\overset{seq}{\simeq}$ is a weak bisimulation relation. This implies that for any finite sequence of configurations $\Gamma_0, \Gamma_1, \ldots, \Gamma_k$ such that $P_1 \vdash \Gamma_i \overset{n_i}{\Rightarrow} \Gamma_{i+1}$ holds for $0 \leq i < k$, there exists a corresponding sequence $\Gamma_0', \Gamma_1', \ldots, \Gamma_k'$ where $P_2 \vdash \Gamma_i' \overset{n_i}{\Rightarrow} \Gamma_{i+1}'$ holds for $0 \leq i < k$, and vice versa. Thus, P_1 and P_2 share the same probability p of all valid executions. Moreover, if $store(\Gamma_0) =^V store(\Gamma_0')$, then, according to Corollaries 1 and 2, we deduce $store(\Gamma_k) =^V store(\Gamma_k')$. Hence, we establish the equality $\mathcal{D}_{P_1} = \mathcal{D}_{P_2}$ where $\mathcal{D}_{P_1}$ and $\mathcal{D}_{P_2}$ are derived by normalizing the distributions $store(\Gamma_k)$ and $store(\Gamma_k')$ with p.
2. Let's assume that S_C is closed under $\overset{dd}{\to}$, $\overset{wcd}{\to}$ and $\xrightarrow{obsntd}$. Then, according to Theorem 1, $\overset{seq}{\simeq}$ is a weak simulation relation. This implies that for any finite sequence of configurations $\Gamma_0, \Gamma_1, \ldots, \Gamma_k$ such that $P_1 \vdash \Gamma_i \overset{n_i}{\Rightarrow} \Gamma_{i+1}$ holds for $0 \leq i < k$, there exists a corresponding sequence $\Gamma_0', \Gamma_1', \ldots, \Gamma_k'$

where $P_2 \vdash \Gamma'_i \overset{n_i}{\Rightarrow} \Gamma'_{i+1}$ holds for $0 \leq i < k$. According to Corollary 2, we can deduce that there exists $q, q' \in [0,1]$ such that if $store(\Gamma_0) =^V_q store(\Gamma'_0)$, then $store(\Gamma_k) =^V_{q'} store(\Gamma'_k)$.
Any ***Observe*** statement that does not belong to S_C in P_1 is replaced by a ***skip*** statement in P_2. Thus, if p_1 and p_2 are the probabilities of all valid executions in P_1 and P_2 respectively, then we must have $p_1 \leq p_2$. Hence, we establish the relation $\mathcal{D}_{P_1} = p \times \mathcal{D}_{P_2}$ from the equality $store(\Gamma_k) =^V_{q'} store(\Gamma'_k)$, where $p = q' \times p_2/p_1$, $p \in [0,1]$ (as if $q' = 1$ then $p_2 = p_1$), and $\mathcal{D}_{P_1}$ and $\mathcal{D}_{P_2}$ are the normalized distributions of P_1 and P_2 by p_1 and p_2 (i.e. $\mathcal{D}_{P_1} = store(\Gamma_k)/p_1$, and similarly for $\mathcal{D}_{P_2}$).

3. Let's assume that S_C is closed under $\overset{dd}{\rightarrow}$, $\overset{wcd}{\rightarrow}$ and $\xrightarrow{obsd}$. We can prove as per (2) above that if $store(\Gamma_0) \leq^V store(\Gamma'_0)$, then $store(\Gamma_k) \leq^V store(\Gamma'_k)$. This implies that $\mathcal{D}_{P_1} \leq \mathcal{D}_{P_2}$ where $\mathcal{D}_{P_1}$ and $\mathcal{D}_{P_2}$ are the normalized distributions of P_1 and P_2 by p_1 and p_2 (i.e. $\mathcal{D}_{P_1} = store(\Gamma_k)/p_1$, and similarly for $\mathcal{D}_{P_2}$) since $p_1 \leq p_2$.

□

8 Slicing Algorithm

In this section, we develop algorithms to compute various slices according to Eq. 5. Our slicing algorithm is based on computing a partial slice set according to Eq. 4 which is subsumed by Eq. 5. The algorithm requires computing data and control dependencies iteratively. Even though there exist efficient algorithms to compute data dependencies, we provide Alg. 1 which performs incremental computation of data dependencies with improved amortized complexity.

Alg. 1 computes the set of CFG nodes dX influenced by data dependencies from a given set of nodes X in the CFG G. It identifies nodes n in G such that there exists a node m in the set X where a data dependency relation $n \xrightarrow{dd} m$ exists. The computation is based on determining the set of relevant variables that impact the computation in X. In other words, the variables referenced by any node n in X are affected by the execution of the statements represented by the nodes in dX. Let us first define RVs concerning a set X of CFG nodes.

Definition 12 (Relevant Variables). *Let G be the CFG of a given program and let X be any set of CFG nodes. The set of* relevant variables *denoted $rv^G(n)$ at any node n in the CFG G comprises all variables v specified in Cases (1) and (2) below:*

1. *$v \in ref(n)$ if any of the conditions (a)-(b) below are satisfied on the CFG node n:*
   (a) ***Initialization:*** *$n \in X$.*
   (b) ***Data dependency:*** *$def(n) \cap rv^G(m) \neq \emptyset$ for some $m \in succ(n)$*
2. ***Continuation criteria:*** *Additionally, $v \in rv^G(m)$ is also considered a RV at n (i.e., $v \in rv^G(n)$) if there exists a CFG path $n = n_1, \ldots, m = n_k$ with $k > 1$, such that $v \notin def(n_i)$ for all $1 \leq i \leq k-1$.*

Algorithm 1: computeDD

Input : $CFGG, RV(n)$ for all $n \in N$, and the set X of CFG nodes
Output: dX - set of CFG nodes affecting the computation at X

```
1  W = X, dX = ∅
2  while (W ≠ ∅) do
3  |  remove an element m from W
4  |  forall (n ∈ pred(m)) do
5  |  |  if (def(n) ∩ RV(m) ≠ ∅) then
6  |  |  |  dX = dd_X ∪ {n}
7  |  |  |  Z = RV(n) ∪ (RV(m) \ def(n))
8  |  |  end
9  |  |  else
10 |  |  |  Z = RV(m)
11 |  |  end
12 |  |  if (Z ⊈ RV(n)) then
13 |  |  |  RV(n) = RV(n) ∪ Z
14 |  |  |  W = W ∪ {n}
15 |  |  end
16 |  end
17 end
18 return dX
```

The above definition is inherently recursive. As per condition 1(a) in the definition, the initial sets of RVs are established from $ref(n)$ in Alg. 1 for all $n \in X$. These initial RVs are propagated backward through CFG due to the continuation criteria (2) specified in the definition. New RVs are generated based on condition 1(b) in the definition, stemming from data dependencies, which are subsequently propagated following the continuation criteria. Alg. 1 computes these RVs, and as new RVs are generated at any CFG node m due to condition 1(b), m affects the computation at nodes in X due to data dependencies and consequently m is included in the set dX of CFG nodes. Any update to the set RV in the algorithm is considered global.

Alg. 2 calculates a partial slice set, denoted as S_p, in accordance with Eq. 4, utilizing a specified slicing criterion, C_p, and the CFG G. At Line 4, it determines the set of CFG nodes influenced by data dependencies, leveraging the computation of RVs carried out in Alg. 1. The *control-closure* operation at Line 5 establishes the closure of control dependencies, employing either weak or strong control closure algorithms.

Algorithm 2: computePartialSliceSet

Input : CFG G, set of CFG nodes C_p, sets of relevant variables RV, and control closure algorithm *control-closure*
Output: partial slice set S_p

$S_p = Cp$
$S = \emptyset$
while $S_p \neq S$ **do**
  $S = computeDD(G, RV, S_p)$
  $S_p = control\text{-}closure(G, S)$
end
return S_p

Algorithm 3: computeSliceSet

Input : CFG G, slicing criterion C, slice type T
Output: slice set S_C

$cc = WCC$
$obsNTSet = \{n : n \in N, n$ represents an *observe* node$\}$
if (T *asks for nontermination-sensitive slice*) **then**
  $cc = SCC$
end
if (T *asks for distribution-sensitive or nontermination-sensitive slice*) **then**
  Let $C' = C \cup \{n_{start}\}$ where n_{start} is the start node in the CFG
  $obsNTSet = SCC(G, C') \setminus WCC(G, C')$
end
$RV(n) = \emptyset$ for all $n \in N$
$S_C = computePartialSliceSet(G, C, RV, cc)$
forall ($n \in obsNTSet$) **do**
  $RV'(n) = \emptyset$ for all $n \in N$
  $S_p = computePartialSliceSet(G, \{n\}, RV', cc)$
  if ($\exists m \in S_C \cap S_p$ *such that* $RV(m) \cap RV'(m) \neq \emptyset$) **then**
    $S_C = S_C \cup S_p$
  end
end
return S_C

Algorithm 3 computes the slice set in accordance with Equation 5. Initially, it determines the appropriate control-closure algorithm to compute the desired slice type, denoted as T. Subsequently, the set $obsNTSet$ is established, containing the designated observe-nontermination instructions determined by the slice type T. This set exclusively encompasses the observe nodes if the aim is to compute a nontermination and distribution insensitive slice. Otherwise, both weak and strong control closure of the set $C' = C \cup \{n_{start}\}$ is computed, and the set $SCC(G, C')$ $WCC(G, C')$ encompasses all CFG nodes representing nonterminating instructions that could influence the computation in C.

Then, the partial slice set S_C is computed which corresponds to the left part of the right hand side in Equation 5. For the rightmost part of that equation, Alg. 2 is invoked again to compute the partial slice set S_p for the slicing criterion $\{n\}$ for each node $n \in obsNTSet$. Sets of RVs RV' are calculated during this process. Line 14 of the algorithm determines the presence of a conditional dependency between the sets S_C and S_p, and S_C is expanded by S_p if such a dependency exists.

After computing the slice set S_C, we can compute the slice P_2 of the original program P_1 by modifying the function $code_2$ as follows:

Definition 13 (Slice). *Let S_C be the slice set of an original program P_1 represented as the CFG $G = (N, E)$ with an associated function $code_1$, and let $n \in N$. We obtain the associated function $code_2$ of the sliced program P_2 as follows:*

1. $code_2(n) = code_1(n)$ *if* $n \in S_C$.
2. $code_2(n) =$***skip*** *if* $n \notin S_C$ *and* $code_1(n)$ *is either an* **observe** *statement or a probabilistic/non-probabilistic assignment statement.*
3. $code_2(n) =$*true if* $n \notin S_C$, $code_1(n)$ *is a predicate statement, and* $|[n_T..n_k]| < |[n_F..n_k]|$ *where* n_T *and* n_F *are the true and false successors of node* n *and* $obs(n) = \{n_k\}$. $code_2(n) =$*false otherwise.*

In a postprocessing phase, we can optimize the sliced program P_2 by removing all ***skip*** statements followed by removing all predicate staments that are *true* /*false* with an empty body.

Efficiency This algorithm facilitates the incremental computation of data dependencies with an improved amortized complexity.

Note 1. to be competed...

9 Related Work

Although numerous studies have delved into various aspects of slicing deterministic programs [13, 11, 24], relatively few have explored slicing in the context of probabilisitc programs.

Hur et al. [10] were the first to demonstrate the inadequacy of conventional dependence-based slicing methods for probabilistic programs. They introduced a novel dependence relation called *observe dependence* to account for the impact of ***Observe*** statements on the slicing criteria. Their approach involves computing slices by considering both conventional data and control dependences, alongside observe dependence. They employ a denotational-style semantics of probabilistic programs and offer mathematical proofs to establish the correctness of their algorithm. Following this work, Amtoft and Banerjee [2, 1] investigated the slicing of structured imperative probabilistic programs by representing them as probabilistic control flow graphs. They distinguish between the slicing specification and the slicing algorithm, ensuring that for any correct slice of a given

program, another slice exists where the program variables are probabilistically independent. They present an algorithm to compute least slices and validate its correctness by adopting the denotational style semantics of Kozen [12].

Navarro and Olmedo [15] adopted a novel approach to slicing probabilistic programs, employing the slicing criterion defined by probabilistic assertions unlike traditional slicing using program variables at designated program points. They utilized the greatest pre-expectation transformer, analogous to Dijkstra's weakest pre-condition transformer, to retroactively propagate post-conditions in backward slice computation. This specification based slicing approach yields smaller-sized slices but demands increased computational overhead.

While prior methods, with the exception of Navarro and Olmedo's, failed to differentiate between observation failure and nontermination, they all computed only nontermination-insensitive, distribution-sensitive slices. In contrast, our approach computes both nontermination-insensitive and nontermination-sensitive slices. We employ an operational-style semantics for probabilistic programs and validate the correctness of our slicing technique using (bi)simulation, akin to the approach outlined in Masud et al. [21].

References

1. Torben Amtoft and Anindya Banerjee. A theory of slicing for probabilistic control flow graphs. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures*, pages 180–196, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
2. Torben Amtoft and Anindya Banerjee. A theory of slicing for imperative probabilistic programs. *ACM Trans. Program. Lang. Syst.*, 42(2), April 2020.
3. Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva. *Foundations of probabilistic programming*. Cambridge University Press, 2020.
4. Benjamin Bichsel, Timon Gehr, and Martin Vechev. 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*, pages 145–185. Springer, 2018.
5. S. Danicic, R. Barraclough, M. Harman, J. D. Howroyd, Á. Kiss, and M. Laurence. A unifying theory of control dependence and its application to arbitrary program structures. *Theoretical Computer Science*, 412(49):6809–6842, 2011.
6. Cynthia Dwork. Differential privacy. In *Proceedings of the 33rd International Conference on Automata, Languages and Programming - Part II*, ICALP'06, pages 1–12. Springer, 2006.
7. Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, jul 1987.
8. Zoubin Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452–459, 2015.
9. Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Sys. Sci.*, 28(2):270–299, 1984.
10. Chung-Kil Hur, Aditya V. Nori, Sriram K. Rajamani, and Selva Samuel. Slicing probabilistic programs. *SIGPLAN Not.*, 49(6):133–144, June 2014.

11. Husni Khanfar, Björn Lisper, and Abu Naser Masud. Static backward program slicing for safety-critical systems. In Juan Antonio de la Puente and Tullio Vardanega, editors, *Reliable Software Technologies - Ada-Europe 2015 - 20th Ada-Europe International Conference on Reliable Software Technologies, Madrid Spain, June 22-26, 2015, Proceedings*, volume 9111 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2015.
12. Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.
13. Björn Lisper, Abu Naser Masud, and Husni Khanfar. Static backward demand-driven slicing. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, PEPM '15, page 115–126, New York, NY, USA, 2015. Association for Computing Machinery.
14. Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
15. Marcelo Navarro and Federico Olmedo. Slicing of probabilistic programs based on specifications. *Science of Computer Programming*, 220:102822, 2022.
16. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999.
17. Federico Olmedo, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Annabelle Mciver. Conditioning in probabilistic programming. *ACM Trans. Program. Lang. Syst.*, 40(1), jan 2018.
18. Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGSOFT Softw. Eng. Notes*, 9(3):177–184, April 1984.
19. Keshav Pingali and Gianfranco Bilardi. Optimal control dependence computation and the roman chariots problem. *ACM Trans. Program. Lang. Syst.*, 19(3):462–491, May 1997.
20. A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Softw. Eng.*, 16(9):965–979, September 1990.
21. Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), page 133–138, New York, NY, USA, 1959. Association for Computing Machinery.
22. Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Trans. Program. Lang. Syst.*, 29(5):27–es, aug 2007.
23. Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE'81, page 439–449. IEEE Press, 1981.
24. Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 439–449. IEEE Press, 1981.
25. Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.

10 Leftovers

10.1 Account for Observe Dependences

As first observed by Hur *et al.* [10], traditional notions of data and control dependences are not enough to produce semantic preserving slices of probabilistic programs *with conditioning*. To illustrate this, consider the program below:

```
1: x :≈ unif[1, 4];
2: y :≈ unif[1, 4];
3: observe (x + y = 3);
4: return (4 − x)
```

The conditioning in line 3 constrains the values of x and y, *e.g.*, prohibiting them be 4 (or even 3). Now, since we are interested in preserving the distribution of final values of x, the observe statement must be preserved in the slice program and, as a consequence, also the random assignment to y. In other words, the only valid slice of the program is the very same program.

This new class of dependence induced by observe statement falls out of the scope of data and control dependences, and must thus be captured by an ad-hoc, additional dependence, dubbed *observe dependence* by Hur *et al.* [10]. To compute program slices we thus take the transitive closure of the union of data, control and observe dependence relations TODO: @Abu, will this be so? Or more like Hur, who somehow "extends" data and control dependance to include also observe dependences?

We now illustrate the effect of observe dependences through a more complex example, adapted from [10]:

```
1: i := 1;
2: b :≈ unif[0, 1];
3: odd := 0;
4: while (b == 0)
5:       i := i + 1;
6:       odd := 1 − odd;
7:       b :≈ unif[0, 1]
8: observe (odd == 0);
9: return i
```

The program basically encodes a geometric distribution, which is conditioned on observing the first success after an odd number of trials. Said otherwise, the observe statement in line 8 blocks all executions where variable i ends up with an even value. Therefore, the observe statement together with the assignments to variable *odd* in lines 3 and 6 must be preserved, and the only correct slicing of the program is the very same program (in either the non-termination sensitive or insensitive flavour, as the only non-terminating execution of the loop occurs with probability 0). In our slicing approach, observe dependences crisply captures this dependence.

10.2 Differentiates conditioning failure and non-termination

TODO: Present, informally, our adopted semantics

Our more fine-grained semantics distinguishes between conditioning failure and non-termination, and this is reflected in our slicing technique. For example, let us consider programs P_0 and P_1 from Fig. 1 and apply them a non-termination sensitive slicing. While our slicing technique will slice away the observe statement from program P_0 (together with the random assignment to y), it will preserve the loop from program P_1.

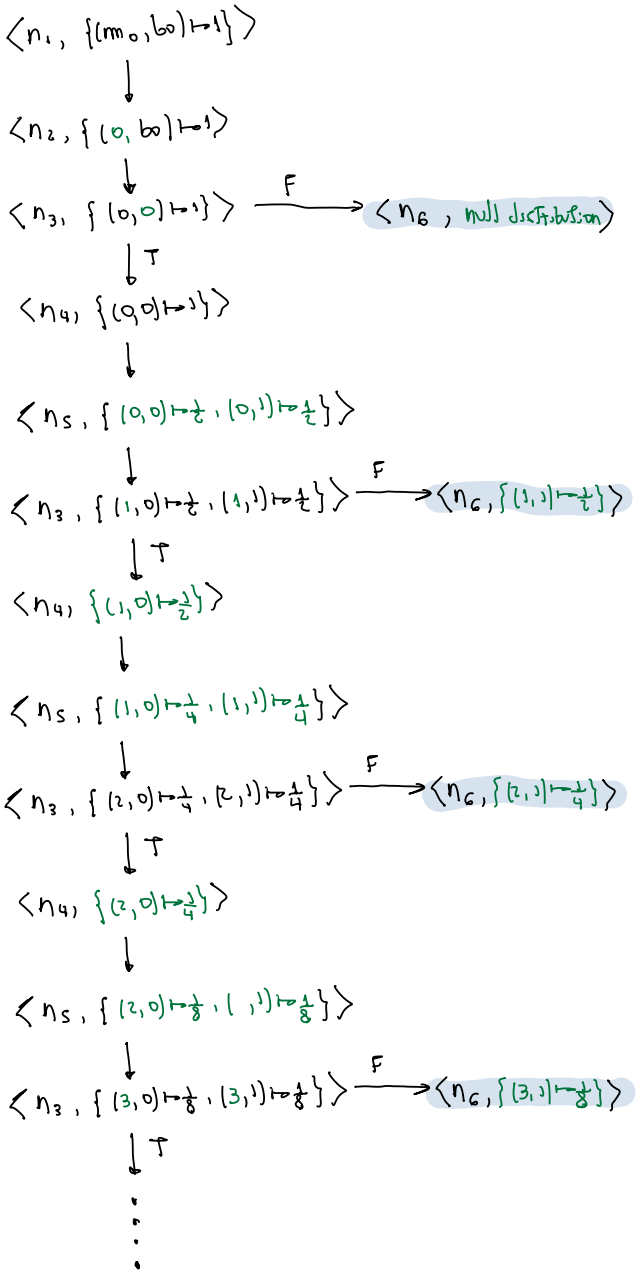


Fig. 5: Small-step semantics for the CFGs from Example 2.