Static Slicing in Probabilistic Programming: Disentangling Observation Failure and Nontermination

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 [27] is a program transformation technique to extract relevant parts of code that affect a particular computation or behavior. In essence, a slice of an original program consists of all statements in the given program that directly or indirectly influence the values at a specific point of interest, known as the slicing criterion. To identify such statements, slicing techniques primarily rely on a static analysis of data and control dependencies among variables.

It has numerous applications in software engineering such as debugging, testing, program understanding, and extraction of reusable components, to name a few [29].

In this work, we specifically focus on slicing probabilistic programs. Probabilistic programs extend ordinary programming languages by incorporating two key features: i) the ability to sample values from probability distributions (random sampling), and ii) the capability to condition variable values using observe statements (conditioning). These features enable probabilistic programs to model and reason about uncertainty in a structured and expressive manner.

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]. Additionally, they are fundamental to traditional randomized algorithms [18]. Finally, they play a crucial role in ensuring privacy, as evidenced by the concept of differential privacy [6].

Nevertheless, 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 demonstrated that traditional data and control dependence must be complemented with a new form of dependence to account for the complex and more intricate effects of conditioning to compute correct slices. 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 from several limitations. First, they adopt a program semantics that conflates observation violation with nontermination —two phenomena that, in our view, should be distinguished. Second, they support only a particular form of slicing known as *nontermination insensitive*. In order to yield potentially smaller sliced programs, this form of slicing allows nonterminating executions in the original program to "become" terminating in the sliced program. While this may be sensible for some applications, in other applications such as program verification, it is of utter relevance that the original and the sliced program share the same nontermination behavior. Finally, the form of (nontermination insensitive) slicing they support is overly restrictive, leaving out program slices that one would arguably deem valid. For example, this slicing does not extend the Weiser's slicing semantics for deterministic programs [28] to probabilistic programs. Section 2 further expands on these shortcomings, while also providing illustrative examples.

Motivated by these limitations, we develop a novel slicing technique for probabilistic programs that supports both nontermination 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. In this paper, our main contributions include the following:

A novel taxonomy of slicing for probabilistic programs. We formally define the notions of nontermination sensitive and insensitive slicing for prob-

abilistic programs (Section 5); this characterization naturally generalizes the counterpart notion for deterministic programs. We further divide nontermination 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 nontermination (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 nonterminating loops in probabilistic programs (Section 6).

Proof framework. We have developed a (bi)simulation-based proof framework that verifies the correctness of the slicing for all slice variants. Through a bisimulation argument, we show that properly combining traditional data and control dependencies with the novel form of *observe(-nontermination)* dependence yields correct program slices, for our entire slicing taxonomy (Section 7).

An algorithm for computing slices. In addition to the theoretical advancements for gaining insights into probabilistic programs, we have developed an algorithm that computes slices based on our slicing taxonomy. This algorithm performs incremental computations, resulting in better amortized complexity compared to the worst-case complexity.

Organization of the paper. The rest of the paper is organized as follows. Sec. 2 provides an overview of our slicing technique and explains how it addresses previous limitations, Sec. 3 gives background on related concepts, Sec. 4 discusses the syntax and our refined semantics of probabilistic programs, Sec. 5 presents a taxonomy with semantic characterizations of different slice types, Sec. 6 illustrates how different slice variants can be computed using a combination of data, control, and a novel form of observe(-nontermination) dependence, Sec. 7 provides a (bi)simulation based proof framework to verify the correctness of different slice variants as discussed in Sec. 6, Sec. 8 introduces an algorithm to compute all different slices and discusses the worst-case complexity of this algorithm, Sec. 9 discusses related works, and finally, Sec. 10 concludes the paper. The proofs of the theorems in Sec.. 7 are provided in appendix A.

2 Overview

In this section, we provide an overview of the main innovative features of our slicing, and explain how they address the limitations of previous approaches.

2.1 Refined Program Semantics

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

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 nontermination 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 (nontermination insensitive) slice of both.

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, thus conflating the failure to establish an observe statement with nontermination.

As argued by Bischsel *et al.* [4], we believe that failure to establish an observation and nontermination 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 standard semantics [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.

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

To address both issues mentioned above, we adopt a semantics without normalization that accumulates the probability of blocked executions. The probability of nontermination is derived from the probabilities of blocked and terminating executions. This semantic approach enables us to distinguish between program P_0 and program P_1. While the probability of any final outcome in both programs is $1/4$, the probabilities of blocked executions in these programs are $3/4$ and 0 respectively, and the probability of nontermination is 0 for P_0 and $3/4$ for P_1.

2.2 Support for Nontermination Sensitive Slicing

As a direct consequence of the semantics they adopt, [10, 2] present a slicing approach that is *insensitive* to nontermination. Program slicing typically comes in two flavors: nontermination sensitive and insensitive, differing on how nonterminating executions of the original program are treated in the sliced program (both variants treat terminating executions uniformly, strictly requiring their preservation). Loosely speaking, a nontermination *sensitive* slicing must preserve all nonterminating executions of the original program in the sliced program. On the other hand, a nontermination *insensitive* slicing allows nonterminating 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 nontermination *sensitive* slicing of P_1 would not allow removing the while-loop.

As exhibited in this example, nontermination insensitive slicing can be more aggressive, 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 nontermination.

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

Notably, our slicing approach supports—and distinguishes between—both forms 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 nontermination insensitive, and *strong* control dependence to a slicing that is nontermination sensitive.

Program P_2 :	Program $P_2^\bullet$:	Program $P_2^\star$:
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: skip;	2: skip;
3: while $(x = 4)$ do skip;	3: while $(x = 4)$ do skip;	3: skip;
4: return x	4: return x	4: return x

Fig. 2: Nontermination sensitive and insensitive slicing: Programs P_2 (original program), $P_2^\bullet$ (nontermination sensitive slice) and $P_2^\star$ (nontermination insensitive slice).

2.3 Natural Notion of Nontermination Insensitive Slicing

Weiser's [28] original notion of nontermination 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 illustrate 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 nontermination insensitive slice of P_2. Furthermore, note that the nonterminating executions of P_2 (i.e. those where $x = 4$) "become" terminating in $P_2^\star$, matching the intuition behind nontermination insensitive slicing that we provide in Section 2.2.

While our notion of nontermination insensitive slice for probabilistic programs arguably captures Weiser's original intuition, previous approaches [10, 2] consider a very restricted form of nontermination 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 recall the notions related to probability distributions, control flow graphs and program dependence relations that we use in our subsequent development.

3.1 Probability distributions

Probability distributions are the cornerstone of probabilistic program semantics.

Definition 1 (Probability distribution). *Given a denumerable set A, a* probability distribution *or simply* distribution *over A is a function*

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

We use $\mathcal{D}(A)$ to denote the set of distributions over A and $\mathcal{D}^{=1}(A)$ the proper subset where $\sum_{a\in A} \mu(a) = 1$.[4]

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 its *support* $\{a \in A \mid \mu(a) > 0\}$

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

Operations. Given $\mu_1, \mu_2 \in \mathcal{D}(A)$, we define the (partial) sum $\mu_1 + \mu_2 \in \mathcal{D}(A)$ as $(\mu_1 + \mu_2)(a) = \mu_1(a) + \mu_2(a)$ provided $\mathsf{w}(\mu_1) + \mathsf{w}(\mu_2) \leq 1$. Similarly, given a scaling factor $c \in \mathbb{R}_{\geq 0}$ and a distribution $\mu \in \mathcal{D}(A)$ such that $c \cdot \mathsf{w}(\mu) \leq 1$, we write $c \cdot \mu$ for the distribution (in $\mathcal{D}(A)$) defined as $(c \cdot \mu)(a) = c \cdot \mu(a)$

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

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

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

Order structure. For any denumerable set A, the set $\mathcal{D}(A)$ of probability distributions over A can be given the structure of an ω complete partial order (ω-cpo) by taking pointwise lifting of the natural order and supremum of ω-chains over $[0,1]$ to the functional space $A \to [0,1]$. Namely, relation "$\leq$" defined as $\mu_1 \leq \mu_2$ iff $\forall a \in A.\ \mu_1(a) \leq \mu_2(a)$ is a partial order over $\mathcal{D}(A)$, and for any ω-chain $\mu_1 \leq \mu_2 \leq \mu_3 \leq \ldots$ in $\mathcal{D}(A)$, operation $\sup_{k\in\mathbb{N}} \mu_i$ defined as $(\sup_{k\in\mathbb{N}} \mu_i)(a) = \sup_{k\in\mathbb{N}} \mu_1(a)$ yields a distribution in $\mathcal{D}(A)$ that is the least upper bound of the ω-chain.

3.2 Control flow graphs

We represent (structured) programs in terms of their control flow graphs (CFGs). In particular, program semantics, dependence relations and slicing are defined over this representation.

[4] Traditionally, elements of $\mathcal{D}(A)$ are referred to as probability *sub*distributions, but for simplicity, here we refer to them as distribution.

Definition 2 (Control flow graph). *A CFG is a directed graph* $G = (N, E, start)$*, where*

1. N *is the set of nodes that represent atomic commands in the program and is partitioned into two subsets, i.e.* $N = N^S \uplus N^P$*, where* N^S *contains* statement *nodes which have at most one successor node and* N^P *contains* predicate *nodes which have one or two successors.*
2. $start \in N$ *is a distinguished node representing the starting point of program execution (the entry point of the CFG). It is the only node of in-degree 0.*
3. $E \subseteq N \times N$ *is the set of edges representing the possible flow of execution in the program.*

A node is called final or exit *if it is either a statement node with no successor, or a predicate node with only one successor. These nodes represent the succesfull termination of a program execution and we write* N^E *for the subset of such nodes.*

Given a CFG $G = (N, E, start)$ and a node $n \in N$, we use $\mathsf{succ}(n)$ and $\mathsf{pred}(n)$ to respectively denote its set of successors $\{m \in N \mid (n, m) \in E\}$ and predecessors $\{m \in N \mid (m, n) \in E\}$.

Paths. 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 $1 \leq i < k$ and $k \geq 1$. An infinite path $n_1, n_2, \ldots$ is denoted by $[n_1..]$. A path is called *non-trivial* if it contains at least two nodes, and *maximal* or *complete* if it is either an infinite path or a finite path that ends at a final node. In terms of notation, we sometimes write $\pi - \{m, n\}$ for set of CFG nodes in path π, excluding nodes m and n.

3.3 Program dependences

Data and control dependences are fundamental relations for computing dependence-based slicing; they are defined over the CFG representation of programs.

In particular, the notion of data dependence relies on the set of program variables that are defined at a given node n, written $\mathsf{def}(n)$, and referenced at a given node n, written $\mathsf{ref}(n)$.

Definition 3 (Data dependence [26]). *Given a CFG* G*, node* n *is called* data dependent *on node* m*, written* $m \xrightarrow{dd} n$*, if there is a program variable* v *such that (1)* $v \in \mathsf{def}(m) \cap \mathsf{ref}(n)$*, and (2) there exists a nontrivial path* π *in* G *from* m *to* n *such that for every node* $m' \in \pi - \{m, n\}$*, it holds that* $v \notin \mathsf{def}(m')$*.*

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

The first formal definition of standard control dependence relation is provided by Ferrante et al.[7] based on the *postdominator* [25] 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 4 (Control Dependence [7, 26]). *Node n is* control dependent *on node m (written $m \xrightarrow{wcd} n$) 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 $m \xrightarrow{wcd} n$ 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 [22, 23, 26, 24] 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. Control dependence relations are further categorized into nontermination insensitive and nontermination sensitive. The relation $\xrightarrow{wcd}$ is a *weak* form of control dependence that does not consider the nontermination scenario, and hence it is nontermination insensitive that require to have an end node. Ranganath et al. [26] extended this definition and does not require the CFG to have an end node, and further provided the definition of nontermination sensitive control dependence. We denote this dependence as $\xrightarrow{scd}$, where "scd" stands for strong control dependence, and it is defined as follows.

Definition 5 (Nontermination-Sensitive Control Dependence [26]). *In a CFG, n_j is (directly) nontermination-sensitive control dependent on node n_i, denoted $n_i \xrightarrow{scd} n_j$, iff n_i has at least two successors, n_k and n_l, such that:*

1. *for all maximal paths π from n_k, n_j always occurs in π and either $n_i = n_j$ or n_j strictly ($n_j = n_i$) precedes any occurrence of n_i in π; and*
2. *there exists a maximal path π_0 from n_l on which either n_j does not occur, or n_i strictly precedes any occurrence of n_j in π_0.*

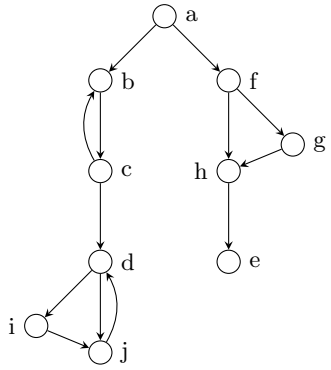
In the above definition, the term "n_j strictly precedes any occurrence of n_i in π" means that: (i) n_j occurs in π; and either (ii) n_i does not occur in π; or (iii) the first occurrence of n_j in π is earlier than the first occurrence of n_i. Fig. 3 presents an example CFG and the $\xrightarrow{scd}$ relation among the nodes in this CFG according to Def. 5.

Fig. 3 depicts the nontermination (in)sensitive control dependences on the example CFG. Some variations between these control dependences exist as shown in the figure. For example, the relation $a \xrightarrow{wcd} d$ holds due to the fact that the loop $b \to c \to b$ is assumed to be terminating. On the other hand, $a \xrightarrow{scd} d$ does not hold as the loop $b \to c \to b$ may be nonterminating which does not contain d and condition (1) in Def. 5 is violated. More details can be found in [26].

For the remainder of the paper, we use the symbol $\xrightarrow{cd}$ to denote any control dependency relation.

3.4 Closure relations

Almost all slicing algorithms need to compute the closure of data and control dependencies. Danicic et al. [5] introduced two generalizations of control



$Nodes$	$\xrightarrow{scd}$	$\xrightarrow{wcd}$
a	b, c, f, h, e	b, c, d, i, j, f, h, e
c	b, c, d, j	b, c
d	i	$-$
f	g	g

Fig. 3: An Example CFG and nontermination (in)sensitive control dependencies among its nodes [26]

dependence, termed weak and strong control closure, which account for nontermination 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 6 (Weak Control Closure). *Let N' be a subset of nodes in the CFG $G = (N, E)$. N' is weakly control closed iff for all $n \notin N'$ reachable from N' implies that 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.

Definition 7 (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

$n \xrightarrow{scd} m_i$ holds, then the weak control dependence relation $n \xrightarrow{wcd} m_i$ 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. Several efficient algorithms [16, 14, 15] have been developed to compute weak and strong control closure, even for interprocedural programs [17]. We apply these algorithms to compute the control closure for the slicing of probabilistic programs.

4 Programming Model

We next present the programming language that we use for describing probabilistic programs, including its syntax and CFG-based semantics.

4.1 Language Syntax

To describe probabilistic programs we adopt a simple imperative language extended with random assignments and observe statements, dubbed cpWhile. A *program* p is a command, followed by a return expression. A *command* is either a no-op (skip), a deterministic assignment ($x := a$), 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 the following grammar:

p	::=	c; return a	*Program*
c	::=	skip $\mid$ $x := a$ $\mid$ $x :\approx d$ $\mid$ $c_1; c_2$ $\mid$ 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	::=	unif$[z_1, z_2] \mid$ distr$\{z_1 \mapsto p_1, \ldots, z_n \mapsto p_n\} \mid \ldots$	*Distribution expressions*

Program variables and arithmetic expressions are 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 (distr$\{z_1 \mapsto p_1, \ldots, z_n \mapsto p_n\}$) and uniform distributions over an integer interval (unif$[z_1, z_2]$).

As for commands, *probabilistic assignments* and observe statements are the only "distinguished" class of statements, endowing cpWhile with a probabilistic behavior. A probabilistic assignment $x :\approx d$ samples a value from the distribution d and assigns it to the program variable x. Statement observe b blocks the

executions that violates b and renormalizes the probability of the remaining—called *valid*—executions.

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

4.2 Language 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 provide semantics for programs, we assume the following *interpretation functions*:

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

These functions take an arithmetic expression, a Boolean expression, or a distribution expression, respectively, along with a store, and return an integer, a Boolean, or a unitary distribution over the set of integers. The definitions of these functions are standard and thus are not detailed here.

The semantics of programs is instrumented to yield, besides the distribution of final stores, also the probability of blocked executions. This is required, in particular, in Section 5, to characterise our different notions of program slicing.

Formally, the semantics of programs is defined in terms of their control flow graphs. The translation of a program into a CFG is also standard, and we refer the reader, *e.g.*, to [20] for a detailed account thereof. We assume a mapping function $P = \mathsf{code}(G)$ to transform a CFG G into its corresponding program P. We also use $\mathsf{code}(n)$ to denote the program instruction for CFG node n. In latter sections, subscripts indicate different program translations from a CFG G, such as $P_1 = \mathsf{code}_1(G)$ for the original program and $P_2 = \mathsf{code}_2(G)$ for its slice. Similarly, $\mathsf{code}_i(n)$ represents the instruction for node n in program P_i.

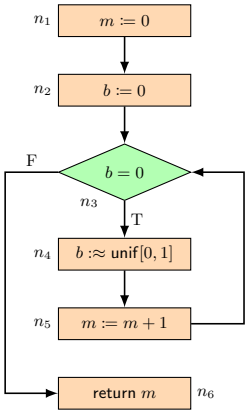
Example 1. In Fig. 4, we present a program that encodes a geometric distribution and its associated CFG. The program, called P_{geo}, repeatedly flips a coin until the first head is observed and returns the numbers of trials required for this outcome.

CFGs derived from cpWhile programs are characterized, in particular, as follows. Statement nodes correspond to no-ops, (deterministic or random) assignments, observe statements or return statements and predicate nodes correspond to the guards of while loops or conditional branchings. Since all programs end with a return statement, predicate nodes always have two successors, given, respectively, by functions $\mathsf{succ}_\mathsf{T}(\cdot)$ (the *true* successor) and $\mathsf{succ}_\mathsf{F}(\cdot)$ (the *false* successor), and all statement nodes, except for those associated to return statements, have one successor, given by function $\mathsf{succ}_\star(\cdot)$. Each CFG has a single exit node, associated to the return statement of the respective program.

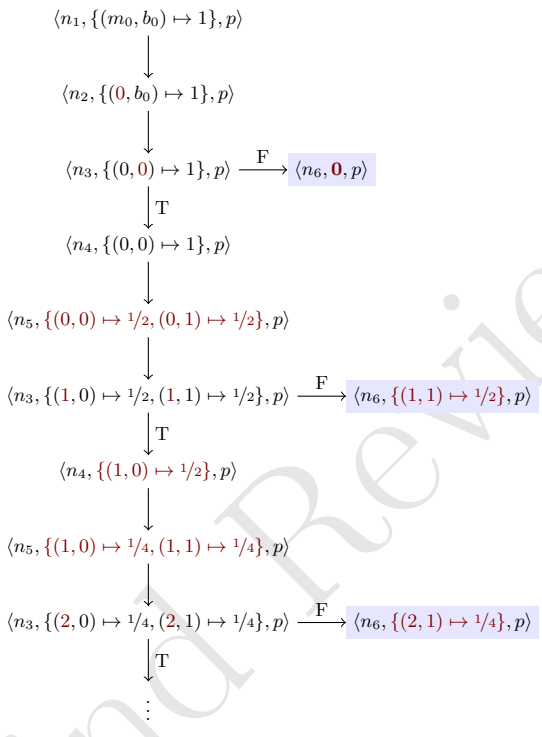
Program P_{geo} :

$m \coloneqq 0;$
$b \coloneqq 0;$
while $(b = 0)$ do
    $b :\approx \mathsf{unif}[0, 1];$
    $m \coloneqq m + 1$
return m

(a) cpWhile program



(b) Control flow graph



(c) Operational semantics

Fig. 4: Program P_{geo} encoding a geometric distribution, together with its control flow graph and an excerpt of its (small-step) operational semantics. To ease the reading, when presenting the operational semantics we color in red the variations of probabilistic states *w.r.t.* the preceding configuration.

To define the semantics of programs over their CGFs, we follow Amtoft and Banerjee [2] and assume that CFG nodes modify distributions over stores, rather than individual stores. To simplify terminology, in the reminder we refer to distributions over stores, i.e. elements of $\mathcal{D}(\Sigma)$, as *probabilistic states* or *probabilistic stores* (which are ranged over by μ).

Concretely, the operational semantics of programs relates CFG configurations (which are ranged over by Γ). A *configuration* of a CFG $G = (N, E, start)$ is a triple $\langle n, \mu, p\rangle$, where $n \in N$ is a node of G, $\mu \in \mathcal{D}(\Sigma)$ is a probabilistic state and $p \in [0, 1]$ is a probability. Intuitively, the (small-step) semantics relates a pair of configurations $\langle n, \mu, p\rangle$ and $\langle n', \mu', p'\rangle$, written $\langle n, \mu, p\rangle \longrightarrow \langle n', \mu', p'\rangle$, if when executing statement or predicate in node n from a probabilistic state μ and a cumulated probability p of violating observe statements, the program transitions, in one step, to successor node n' resulting a in a probabilistic state μ' and a cumulated probability p' of violating observe statements. (Note that

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

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

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

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

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

Fig. 5: Small-step semantics over program CFGs.

as the execution of a program progresses, the probability of violating observe statements can only remain equal or increase; therefore, in such a transition, we will always have $p' \geq p$.) The formal definition of relation $\longrightarrow$ is provided in Fig. 5. All rules are self-explanatory. In particular, we have no rule for nodes representing return statements because they are always final nodes, with no successor.

Example 2. In Fig. 4c, we depict an excerpt of the small-step semantics of program P_{geo}.

Even though transition relation $\longrightarrow$ fully describes program behaviour, we are mostly interested in describing the distribution of final stores or *final probabilistic state* reached by programs. Informally, we can construct such is final probabilistic state by adding up all probabilistic states reached within exit nodes (in our running example from Fig. 4, given by the configurations highlighted in violet). Formally, we need to define a *step-indexed* relation $\longrightarrow_k$ that collects probabilistic states reached in exit nodes within k steps, and take the "limit" of the so-defined sequence. We provide the definition of this step-indexed semantics in Fig. 6.

Example 3. Continuing with program P_{geo} from Fig. 4, for any initial probabilistic state μ_0 and any probability p, we have:

- $\langle n_1, \mu_0, p\rangle \longrightarrow_k \langle\!\langle \mathbf{0}, p\rangle\!\rangle$ for all $k = 0, \ldots, 6$

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

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

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

$$\frac{\begin{array}{c}\langle n,\mu,p\rangle \longrightarrow \langle \mathsf{succ}_\mathsf{T}(n),\mu_T,p\rangle \\ \langle \mathsf{succ}_\mathsf{T}(n),\mu_T,p\rangle \longrightarrow_k \langle\!\langle \mu'_T,p+p'_T\rangle\!\rangle \\ \langle n,\mu,p\rangle \longrightarrow \langle \mathsf{succ}_\mathsf{F}(n),\mu_F,p\rangle \\ \langle \mathsf{succ}_\mathsf{F}(n),\mu_F,p\rangle \longrightarrow_k \langle\!\langle \mu'_F,p+p'_F\rangle\!\rangle\end{array}}{\langle n,\mu,p\rangle \longrightarrow_{k+1} \langle\!\langle \mu'_T+\mu'_F,p+p'_T+p'_F\rangle\!\rangle} \quad \text{provided } k \geq 1 \text{ and } n \in N^P$$

Fig. 6: Step-indexed semantics over program CFGs.

- $\langle n_1,\mu_0,p\rangle \longrightarrow_k \langle\!\langle \{(1,1)\mapsto 1/2\}, p\rangle\!\rangle$ 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 $k = 10,\ldots,12$

As we can already notice in the above example, relation $\longrightarrow_k$ enjoys two relevant properties:

Determinism: For any configuration $\langle n,\mu,p\rangle$ and any number of steps k, there exist unique μ' and 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 $p \leq p'$.

Proof. We can establish the determinism of $\longrightarrow_k$ by a simple induction over k, exploiting the deterministic nature of relation $\longrightarrow$: if $n \in N^S \setminus N^E$, then for every μ and p there exist unique μ' and p' such that $\langle n,\mu,p\rangle \longrightarrow \langle \mathsf{succ}_\star(n),\mu',p'\rangle$; likewise, if $n \in N^P$, then for every μ and p there exist unique μ_T,μ_F and p_T,p_F such that $\langle n,\mu,p\rangle \longrightarrow \langle \mathsf{succ}_\mathsf{T}(n),\mu_T,p_T\rangle$ and $\langle n,\mu,p\rangle \longrightarrow \langle \mathsf{succ}_\mathsf{F}(n),\mu_F,p_F\rangle$.

To establish the ω-chain property, we show that $k_1 < k_2$, $\langle n,\mu,p\rangle \longrightarrow_{k_1} \langle\!\langle \mu_1,p_1\rangle\!\rangle$ and $\langle n,\mu,p\rangle \longrightarrow_{k_2} \langle\!\langle \mu_2,p_2\rangle\!\rangle$ entails that $\mu_1 \leq \mu_2$ and $p_1 \leq p_2$. The proof proceed by routine induction on the derivation of $\langle n,\mu,p\rangle \longrightarrow_{k_2} \langle\!\langle \mu_2,p_2\rangle\!\rangle$ relying, as above, on the deterministic nature of relation $\longrightarrow$.

The ω-chain property allows formally defining program outcomes and the probability of blocked executions:

Definition 8 (Program semantics). *The* distribution transformer *semantics of a* cpWhile *program of CFG $G = (N,E,start)$ is given by function $[\![G]\!]_{\mathsf{dt}} \colon N \times$*

$\mathcal{D}(\Sigma) \times [0,1] \to \mathcal{D}(\Sigma) \times [0,1]$, *defined as*

$$[\![G]\!]_{\mathsf{dt}}\,\langle n,\mu,p\rangle \;=\; \sup_{k\in\mathbb{N}} \big\langle \langle\!\langle \mu_k, p_k\rangle\!\rangle \mid \langle n,\mu,p\rangle \;\longrightarrow_k\; \langle\!\langle \mu_k, p_k\rangle\!\rangle \big\rangle$$

The (unnormalized) final probabilistic state $[\![G]\!]\,\mu_0$ *reached by the program, when executed from initial probabilistic state* μ_0*, and the* probability of blocked executions $\mathsf{Pr}[G(\mu_0) \in \lightning]$ *are respectively defined as*

$$[\![G]\!]\,\mu_0 \;=\; \mu' \qquad \textit{and} \qquad \mathsf{Pr}[G(\mu_0) \in \lightning] \;=\; p',$$

where $\langle\!\langle \mu', p'\rangle\!\rangle \;=\; [\![G]\!]_{\mathsf{dt}}\,\langle \mathit{start}, \mu_0, 0\rangle$.

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.

5.1 Nontermination

We define the probability of nontermination of a program as the complement of the sum between the probability of blocked executions and the probability of valid terminating executions:

$$\mathsf{Pr}[G(\mu_0) \in \Uparrow] \;=\; 1 - \big(\mathsf{Pr}[G(\mu_0) \in \lightning] + \mathsf{w}([\![G]\!]\,\mu_0)\big)$$

Since our program semantics separate blocked executions from the terminating ones (see the [OBSERVE] rule in Fig. 5) and both are mutually exclusive with nonterminating executions, their combined total probability equals 1. By calculating the marginal distribution of nontermination as above, we can fairly compare the nonterminating behavior of the original program and its slice.

5.2 Taxonomy

Non-termination sensitive slicing. A non-termination *sensitive* slicing will preserve the program outcome as well as the probability of nontermination. Let P' be a subprogram of P (i.e. P' is obtained from P by replacing some of its statements by skip). We formally state that P' is a valid *non-termination sensitive* slice of P *w.r.t.* a set of variables V that are used in the return expression iff for every initial probabilistic state μ_0 of P and $\mu_0' = \mu_0|_V$ of P', the following equation hold:

$$\exists q \in [0,1].\ ([\![P]\!]\mu_0)|_V = q \cdot ([\![P']\!]\mu_0')|_V \text{ and } \mathsf{Pr}[P(\mu_0) \in \Uparrow] = \mathsf{Pr}[P'(\mu_0') \in \Uparrow] \tag{1}$$

The first equality in the above equation states the (relative) equivalence of the probabilistic states of P and P'. While we require the equivalence of the probability of nontermination of both programs, we only require the equality for the final probabilistic stores upto a scaling factor q. 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 are as follows:

$$\begin{array}{lcl}(\llbracket P_2 \rrbracket \mu_0)|_{\{x\}} & = & \{1 \mapsto 1/4, 2 \mapsto 1/4, 3 \mapsto 1/4, 4 \mapsto 0, \_ \mapsto 0\} \\ \mathsf{Pr}[P_2(\mu_0) \in \Uparrow] & = & 1/4\end{array}$$

where symbol "_" stands for "otherwise", i.e., any other value not present in the preceding enumeration. The only proper slice of P_2 *w.r.t.* x 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 therefore, 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 *w.r.t.* a set of variables V that are used in the return expression iff for every initial probabilistic state μ_0,

$$(\llbracket P \rrbracket \mu_0)|_V \;\leq\; (\llbracket P' \rrbracket \mu_0)|_V \text{ and } \mathsf{Pr}[P(\mu_0) \in \Uparrow] \geq \mathsf{Pr}[P'(\mu_0') \in \Uparrow] \qquad (2)$$

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

$$\begin{array}{lcl}(\llbracket P_2^\star \rrbracket \mu_0)|_{\{x\}} & = & \{1 \mapsto 1/4, 2 \mapsto 1/4, 3 \mapsto 1/4, 4 \mapsto 1/4, \_ \mapsto 0\} \\ \mathsf{Pr}[P_2^\star(\mu_0) \in \Uparrow] & = & 0\end{array}$$

From the above equations, we conclude that $P_2^\star$ is a non-termination insensitive slice of P_2 *w.r.t.* variable x.

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 *w.r.t.* x:

$$\begin{array}{lcl}(\llbracket P_1 \rrbracket \mu_0)|_{\{x\}} & = & 1/4 \cdot \{1 \mapsto 1/4, 2 \mapsto 1/4, 3 \mapsto 1/4, 4 \mapsto 1/4, \_ \mapsto 0\} \\ \mathsf{Pr}[P_1(\mu_0) \in \Uparrow] & = & 1/4 \\ (\llbracket P_1^\star \rrbracket \mu_0)|_{\{x\}} & = & \{1 \mapsto 1/4, 2 \mapsto 1/4, 3 \mapsto 1/4, 4 \mapsto 1/4, \_ \mapsto 0\} \\ \mathsf{Pr}[P_1^\star(\mu_0) \in \Uparrow] & = & 0\end{array}$$

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

$$\exists q \in [0,1].\ (\llbracket P \rrbracket \mu_0)|_V \ = \ q \cdot (\llbracket P' \rrbracket \mu_0)|_V \text{ and } \mathsf{Pr}[P(\mu_0) \in \Uparrow] \geq \mathsf{Pr}[P'(\mu_0') \in \Uparrow] \tag{3}$$

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.

Note that a nontermination sensitive slice is also distribution sensitive, as it always preserves the relative distribution of the original program. Furthermore, a nontermination insensitive distribution sensitive slice may also be nontermination sensitive, e.g., if the original program is always terminating, since in that case both Eq. 1 and Eq. 3 are satisfied. For example, program $P_1^\star$ in Fig. 1 is both a nontermination sensitive and nontermination insensitive distribution sensitive slice of P_0. Moreover, a nontermination insensitive distribution sensitive slice may also be nontermination insensitive distribution insensitive when both Eq. 2 and Eq. 3 are satisfied. For example, program $P_1^\star$ in Fig. 1 is a nontermination insensitive distribution (in)sensitive slice of P_1.

6 Slicing Definition

In the context of probabilistic programs, 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. 5. 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* [5] 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 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 9 (Observe-Nontermination dependence). *Let $G = (N, E, n_\star)$ 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 between n_o and n_r, denoted $n_o \xrightarrow{obntd} n_r$, holds iff there exists a CFG node $n \in N$ such that the reflexive transitive closure relations $n(\overset{wcd}{\rightarrow} \cup \overset{dd}{\rightarrow})^* n_o$ and $n(\overset{wcd}{\rightarrow} \cup \overset{dd}{\rightarrow})^* n_r$ hold.*

Thus, the relation $n_o \xrightarrow{obntd} n_r$ ensures the presence of a node n creating a V-structure of dependences between (n, n_r) and (n, n_o), implying that n_r and n_o are not probabilistically independent. If n_o in the above definition represents solely an ***observe*** statement, then we refer to the dependence relation as an **observe dependence**, denoted $n_o \xrightarrow{obsd} n_r$. Note that we have used the weak form of control dependence relation $\overset{wcd}{\rightarrow}$ in the above definition to avoid trivial relation. If n_o represents a predicate node representing the loop header of a nonterminating loop, and we employ the strong form of nontermination sensitive control dependence relation $\overset{scd}{\rightarrow}$, then $n_o(\overset{wcd}{\rightarrow} \cup \overset{dd}{\rightarrow})^* n_o$ and $n_o(\overset{wcd}{\rightarrow} \cup \overset{dd}{\rightarrow})^* n_r$ always hold. Consequently, $n_o \xrightarrow{obntd} n_r$ trivially holds, which is undesirable.

We illustrate the aforementioned dependence relation with examples. Consider programs P_0, P_1, and P_2 in Fig. 1 and 2. Let n_i represents the CFG node

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

Slicing class	$\overset{cd}{\rightarrow}$	$\overset{R}{\rightarrow}$
non-termination sensitive	$\xrightarrow{scd}$	$\xrightarrow{obntd}$
non-termination insensitive, distribution sensitive	$\xrightarrow{wcd}$	$\xrightarrow{obntd}$
non-termination insensitive, distribution insensitive	$\xrightarrow{wcd}$	$\xrightarrow{obsd}$

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

of statement i in the given program. For programs P_0 and P_1, the relation $n_3 \xrightarrow{obntd} n_4$ does not hold, as there is no CFG node n such that $n(\overset{wcd}{\rightarrow} \cup \overset{dd}{\rightarrow})^* n_3$ and $n(\overset{wcd}{\rightarrow} \cup \overset{dd}{\rightarrow})^* n_4$ hold for any $n \in \{n_1, n_2, n_3, n_4\}$. This is evident since the variables x and y are independent in these programs. Consequently, n_3 does not affect the final (relative) distribution of these programs, and it can be removed (or replaced by an ***skip*** statement) from some slices that satisfy any of Eq. (1)- (3). This is evident, as we have seen in Sec. 5, that $(\llbracket P_i \rrbracket \mu_0)|_{\{x\}} = \frac{3}{4} \times (\llbracket P_1^\star \rrbracket \mu_0')|_{\{x\}}$ for $i = 0, 1$ and $\mu_0' = \mu_0|_{\{x\}}$. In the case of program P_2, the relation $n_3 \xrightarrow{obntd} n_4$ holds due to the relations $n_1 \overset{dd}{\rightarrow} n_3$ and $n_1 \overset{dd}{\rightarrow} n_4$. This implies that n_3 affects the final distribution of P_2. As illustrated in Sec. 5, there exists no $q \in [0, 1]$ such that $(\llbracket P_2 \rrbracket \mu_0)|_{\{x\}} = q \times (\llbracket P_2^\star \rrbracket \mu_0')|_{\{x\}}$ holds for $\mu_0' = \mu_0|_{\{x\}}$. Thus, n_3 cannot be sliced away from any distribution sensitive slice of P_2. However, a nontermination insensitive distribution insensitive slice of P_2 may still slice away the nonterminating loop at n_3 when Eq. 2 is satisfied.

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:

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

The equation above is parameterized by the control dependency relation $\overset{cd}{\rightarrow}$, which may manifest as either a weak or strong control dependency relation $\xrightarrow{wcd}$ or $\xrightarrow{scd}$ respectively. Additionally, it involves the relation $\overset{R}{\rightarrow}$, which can either signify the *observe-nontermination dependency* relation $\xrightarrow{obntd}$ or only the *observe dependency* relation $\xrightarrow{obsd}$. To calculate each class of slicing, relations $\overset{cd}{\rightarrow}$ and $\overset{R}{\rightarrow}$ must be instantiated as specified 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 $slice_C(G, \xrightarrow{wcd}, \xrightarrow{obntd})$ 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 $\xrightarrow{obntd}$ 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 $slice_C(G, \xrightarrow{scd}, \xrightarrow{obntd})$, 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* instructions.

Next, we introduce the concept of *next observable nodes*, which was originally developed within the realm of non-probabilistic programs. As we traverse the CFG and encounter a node, it becomes important to identify the first reachable 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 referred to as the *next sliced-node to be visited*. However, we retain the term *next observable* for historical reasons and ask readers not to confuse it with the ***observe*** instruction. This concept is employed in building the sliced program from the CFG and the slice set (see Def. 11), and serves as a crucial tool in formulating the proof framework for verifying the correctness of program slicing (Sec. 7). The formal definition of next observable nodes is as follows:

Definition 10 (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 $1 \leq i \leq k-1$.*

After computing the slice set $S_C = slice_C(G, \overset{cd}{\rightarrow}, \overset{R}{\rightarrow})$, we can compute the slice $P_2 = \mathsf{code}_2(G)$ according to the following definition.

Definition 11 (Slice). *Let $G = (N, E, start)$ be the CFG of an original program $P_1 = \mathsf{code}_1(G)$, let S_C be the slice set, and let $n \in N$. We obtain the slice $P_2 = \mathsf{code}_2(G)$ by constructing the function code_2 as follows:*

1. $\mathsf{code}_2(n) = \mathsf{code}_1(n)$ *if* $n \in S_C$.
2. $\mathsf{code}_2(n) =$***skip*** *if* $n \notin S_C$ *and* $\mathsf{code}_1(n)$ *is either an* ***observe*** *statement or a (probabilistic) assignment statement.*
3. $\mathsf{code}_2(n) =$*true if* $n \notin S_C$, $\mathsf{code}_1(n)$ *is a predicate node,* $|[\mathsf{succ}_\mathsf{T}(n)..n_k]| < |[\mathsf{succ}_\mathsf{F}(n)..n_k]|$ *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.

7 Slicing Correctness

In this section, we develop theories and a proof framework to verify the correctness of program slicing. We need some additional mathematical notations to state the theorems and their proofs specifying the correctness criteria.

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. 11 from the slice set S_C. 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 12 (Partial-equivalence of probabilistic stores). *Let $V \subseteq \mathcal{V}$ be a subset of variables, and let $\mu_1, \mu_2 \in \mathcal{D}(\Sigma)$ be pairs of probabilistic stores. The relation $\mu_1 \preccurlyeq_V \mu_2$ is defined as follows:*

$$\mu_1 \preccurlyeq_V \mu_2 \stackrel{def}{\equiv} \exists q \in [0,1] \cdot \mu_1|_V = q \cdot \mu_2|_V.$$

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. 10 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 13 (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:

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

The observable transition $\xrightarrow{n}$ 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 14 (Weak Simulation and Bisimulation). *Consider the following properties for relation Φ:*

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

(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.*

The concept of relevant variables (RVs) is fundamental in program slicing theories. It helps understanding the dependencies between values of program variables at different program locations. Informally, a RV at a node is a program variable that may affect the value of some variable in the slicing criterion. Formally, we define RVs in our probabilistic context as follows:

Definition 15 (Relevant Variables). *Let G be the CFG of a given program, let C be the slicing criterion, and let $S_C = slice_C(G, \overset{cd}{\rightarrow}, \overset{R}{\rightarrow})$ be the slice set. The set of RVs at any node n in the CFG G, denoted $\mathsf{rv}(n)$, comprises all variables v specified in Cases (1) and (2) below:*

1. *$v \in \mathsf{ref}(n)$, and consequently, $n \in S_C$, if any of the conditions (a)-(d) below are satisfied on the CFG node n:*
   (a) *Initialization: $n \in C$.*
   (b) *Data dependency: $\mathsf{def}(n) \cap \mathsf{rv}(m) \neq \emptyset$ for some $m \in succ(n)$.*
   (c) *Control dependency: $n \xrightarrow{cd} m$ holds for some $m \in S_C$.*
   (d) *Observe(-nontermination) dependency: $n \xrightarrow{R} n_r$ holds for some $n_r \in C$, where $\xrightarrow{R}$ is specified in Table 1.*
2. *Continuation criteria: Additionally, $v \in \mathsf{rv}(m)$ is a RV at n if $v \notin \mathsf{def}(n)$, where $m \in succ(n)$.*

The above definition is inherently recursive. As per condition 1(a) in the definition, the initial sets of RVs are established from $\mathsf{ref}(n)$ for $n \in C$. These initial RVs are propagated backward through CFG due to the continuation criteria (2) specified in the definition. New RVs are generated based on conditions 1(b)-1(d), stemming from different dependencies, which are subsequently propagated following the continuation criteria.

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

Definition 16 ($\simeq$). *Consider any valid configurations Γ_1 and Γ_2 of any program P_1 and its slice P_2, respectively, such that $n_i = node(\Gamma_i)$ and $\mu_i = store(\Gamma_i)$ for $i = 1, 2$. Let S_C be the slice set for the slice P_2, and let $V = \mathsf{rv}(n_1)$ be the set of RVs at n_1. The relation $\Gamma_1 \simeq \Gamma_2$ holds if the following conditions are met:*

1. *$obs(n_1) = obs(n_2)$, and*
2. *$\mu_1 \preccurlyeq_V \mu_2$, where the relation $\preccurlyeq_V$ is specified in Def. 12.*

Theorem 1 below states that $\simeq$ is either a weak simulation or a weak bisimulation, depending on the dependence 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 $\simeq$ is a weak simulation relation if S_C is closed under $\xrightarrow{dd}$, $\xrightarrow{wcd}$, and $\xrightarrow{obsd}$. Otherwise, it is a weak bisimulation relation.*

Theorem 2 stated below ensures that the slice P_2 is a correct nontermination (in)sensitive distribution (in)sensitive slice of P_1 if $\simeq$ 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 $\xrightarrow{dd}$, $\xrightarrow{scd}$ and $\xrightarrow{obntd}$.*
2. *P_2 is a nontermination insensitive distribution sensitive slice of P_1 if S_C is closed under $\xrightarrow{dd}$, $\xrightarrow{wcd}$ and $\xrightarrow{obntd}$.*
3. *P_2 is a nontermination insensitive distribution insensitive slice of P_1 if S_C is closed under $\xrightarrow{dd}$, $\xrightarrow{wcd}$ and $\xrightarrow{obsd}$.*

All proofs of theorems are provided in Appendix A.

8 Slicing Algorithm

In this section, we provide 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.

Alg. 1 computes two things: (1) the set of CFG nodes DD that affects the input CFG nodes S due to data dependency, and (2) updates the set of RVs due to the *data dependency criteria* (1b) and the *continuation criteria* (2) in Def. 15. It supports incremental computation of data dependencies on-demand with improved amortized complexity. Since the computation is based on RVs, for any subsets of CFG nodes $S_1 \subset S_2$, calling Alg. 1 first for S_2 and then for S_1 will result in less computation and faster termination for the second call. This is because no new RVs will be identified in the second call. Conversely, if it is called first for S_1 and then for S_2, the computation cost and termination time for the second call will be proportional to that of computing the data dependency set for $S_2 \setminus S_1$ for the first time.

Given a CFG G and a slicing criterion C_p, Alg. 2 calculates a partial slice set S_p in accordance with Eq. 4. The $\mathsf{closure}(\cdot)$ operation computes either the weak or strong control closure specified by the control dependency type $\xrightarrow{cd}$ for the set $S \cup S_p$. It updates RVs according to the *control dependency* criteria (1c) and *continuation criteria* (2) in Def. 15. We consider the use of standard control closure algorithm as discussed in Sec. 3 and omit the details. All updates to the set $\mathsf{rv}(\cdot)$ are considered global by our algorithms.

Alg. 3 computes the slice set $S_C = slice_C(G, \xrightarrow{cd}, \xrightarrow{R})$ for a given CFG G and a slicing criterion C in accordance with Eq. 5. The slice_type takes any value from the set $\{\mathsf{ns}, \mathsf{ds}, \mathsf{ni}\}$ representing nontermination sensitive, nontermination

(in)sensitive distribution sensitive, and nontermination insensitive distribution insensitive slice types. This type determines the type of control dependency $\overset{cd}{\rightarrow}$ and the set N_{obnt} of CFG nodes representing *observe-nontermination* instructions. If $\mathsf{slice\_type} = \mathsf{ni}$, we consider the relation $\overset{R}{\rightarrow}{=}\overset{obsd}{\rightarrow}$, and N_{obnt} includes only the ***observe*** instructions. Otherwise, $\overset{R}{\rightarrow}{=}\overset{obntd}{\rightarrow}$ and N_{obnt} includes all CFG nodes representing ***observe*** instructions and predicate nodes that may contribute to divergence. The $closure(G, \overset{scd}{\rightarrow}, C \cup \{start\})$ operation encompasses all predicate nodes that allow nonterminating executions.

Algorithm 1: compDD

Input : CFG $G = (N, E, start)$, $\mathsf{rv}(n)$ for all $n \in N$, set of nodes $S \subseteq N$
Output: DD - set of CFG nodes affecting S due to data dependency

```
1  DD = ∅
2  while (S ≠ ∅) do
3  |  Remove m from S
4  |  forall (n ∈ pred(m)) do
5  |  |  if (def(n) ∩ rv(m) ≠ ∅) then
6  |  |  |  DD = DD ∪ {n}
7  |  |  |  RV_t = rv(n) ∪ (rv(m) \ def(n))
8  |  |  else
9  |  |  |  RV_t = rv(m)
10 |  |  end
11 |  |  if (RV_t ⊈ rv(n)) then
12 |  |  |  rv(n) = rv(n) ∪ RV_t
13 |  |  |  S = S ∪ {n}
14 |  |  end
15 |  end
16 end
17 return DD
```

Algorithm 2: compPSlice

Input : CFG $G = (N, E, start)$, set of nodes $C_p \subseteq N$, RVs $\mathsf{rv}(n)$ for $n \in N$, and control dependency type $\overset{cd}{\rightarrow}$
Output: partial slice set S_p

```
1  S_p = C_p, S = ∅
2  while S_p ≠ S do
3  |  S = compDD(G, rv(·), S_p)
4  |  S_p = closure(G, →cd, S ∪ S_p, rv(·))
5  end
6  return S_p
```

The partial slice set S_C is computed at Line 9 by the $\mathsf{compPSlice}(\cdot)$ operation, which corresponds to the left part of the right hand side in Eq. 5. For the rightmost part of that equation, $\mathsf{compPSlice}(\cdot)$ is invoked repeatedly at Line 13

Algorithm 3: compSlice

Input : CFG $G = (N, E, start)$, slicing criterion C, slice_type
Output: slice set S_C

1 $\overset{cd}{\rightarrow}= \begin{cases} \overset{scd}{\rightarrow} & \textit{if } \mathsf{slice\_type} == \mathsf{ns} \\ \overset{wcd}{\rightarrow} & \textit{otherwise} \end{cases}$
2 Let $N_{obnt} \subseteq N$ contains all ***observe*** nodes
3 **if** slice_type == ds **then**
4 $\quad scc = \mathsf{closure}(G, \overset{scd}{\rightarrow}, C \cup \{start\})$
5 $\quad N_{obnt} = N_{obnt} \cup scc$
6 **end**
7 $\mathsf{rv}(n) = \begin{cases} \mathsf{ref}(n) & \text{if } n \in C \\ \emptyset & \text{for all } n \notin C \end{cases}$
8 $S_C = \mathsf{compPSlice}(G, C, \mathsf{rv}(\cdot), \overset{cd}{\rightarrow})$
9 **forall** $(n \in N_{obnt})$ **do**
10 $\quad \mathsf{rv\_aux}(n) = \mathsf{ref}(n)$
11 $\quad \mathsf{rv\_aux}(m) = \emptyset$ for all $m \in N$ and $m \neq n$
12 $\quad S_p = \mathsf{compPSlice}(G, \{n\}, \mathsf{rv\_aux}(\cdot), \overset{cd}{\rightarrow})$
13 $\quad$ **if** $(\exists m \in S_C \cap S_p . \mathsf{rv}(m) \cap \mathsf{rv\_aux}(m) \neq \emptyset)$ **then**
14 $\quad\quad S_C = S_C \cup S_p$
15 $\quad\quad \mathsf{rv}(n) = \mathsf{rv}(n) \cup \mathsf{rv\_aux}(n)$ for all $n \in N$
16 $\quad$ **end**
17 **end**
18 **return** S_C

to compute the partial slice set S_p for the slicing criterion $\{n\}$ for each $n \in N_{obnt}$. Sets of RVs $\mathsf{rv\_aux}(\cdot)$ 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 and $\mathsf{rv}(\cdot)$ are expanded by S_p and $\mathsf{rv\_aux}(\cdot)$ if such a dependency exists.

Computational complexity The worst-case computational complexity of Alg. 1 is dominated by the cost of the ***while*** loop, which iterates as long as new RVs are discovered. This loop iterates at most $|N| + |E|$ times to transfer the RVs. New RVs are introduced in the sets of RVs when a node is included in the data dependency set DD. Since we cannot include more than $|N|$ nodes in DD, the loop iterates at most $(|N| + |E|) \times |N|$ times. In each iteration of this loop, the worst case cost is $O(|V|)$ where V is the set of program variables. Moreover, according to Def. 2, any CFG node has at most two successors, and thus $|E| \leq 2 \times |N|$. Thus, the worst case complexity of Alg. 1 is $O(|N|^2|V|)$.

Alg. 2 uses Alg. 1 and the closure algorithm. The worst-case complexity of the closure algorithm by the best baseline approaches [16, 14, 15] are $O(|N|^2)$ and $O(|N|^3)$ if $\overset{cd}{\rightarrow}=\overset{wcd}{\rightarrow}$ and $\overset{cd}{\rightarrow}=\overset{scd}{\rightarrow}$ respectively. The ***while*** loop in this algorithm

iterates at most $|N|$ times. Thus, The worst-case costs of Alg. 2 are $O(|N|^3|V|)$ and $O(|N|^4)$ if $\overset{cd}{\rightarrow}=\overset{wcd}{\rightarrow}$ and $\overset{cd}{\rightarrow}=\overset{scd}{\rightarrow}$ respectively.

The dominating cost of Alg. 3 is the cost of the ***forall*** loop at Lines 9-17 that includes the call to Alg. 2 at Line 12, and this loop iterates at most $|N_{obnt}|$ times, where $|N_{obnt}| \leq |N|$ is the maximum number of predicate nodes involved in nonterminating execution and observe nodes in the CFG. Thus, the worst-case cost of this algorithms are $O(|N|^4|V|)$ and $O(|N|^5)$ if $\overset{cd}{\rightarrow}=\overset{wcd}{\rightarrow}$ and $\overset{cd}{\rightarrow}=\overset{scd}{\rightarrow}$ respectively.

However, these algorithms facilitate the incremental computation of data and control dependencies by retaining the set of computed RVs. Thus, the amortized complexity is better than the worst case complexity. For example, it has been shown in [15] that the practical cost of the closure operation for weak control dependence is closer to the $|N|$ or $|N|log|N|$ curve even though its worst-case complexity is $O(|N|^2)$. We believe that our algoritm can be optimized that may improve practical performance and the worst-case complexity by an order of magnitude, by merging the computation of the left and the right part of Eq. 5 under the same iteration. We left this as a future work.

9 Related Work

Although numerous studies have delved into various aspects of slicing deterministic programs [13, 11, 28], relatively few have explored slicing in the context of probabilistic 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 [19] 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].

10 Conclusion

In this study, we have significantly advanced the understanding of static slicing in probabilistic programming by introducing a novel taxonomy and refined semantics. By disentangling observation failure from nontermination, we have identified various slicing variants, namely nontermination sensitive and nontermination insensitive slicing. The latter is further divided into distribution sensitive and distribution insensitive categories. The refined operational semantics provides a clearer distinction between observation violations and nontermination, which is crucial for accurate program analysis. The distinction between nontermination-sensitive and nontermination-insensitive slices is crucial for accurately preserving the relative distribution and behavior of the original program. Understanding these relationships helps ensure that the slicing process maintains the intended probabilistic characteristics, whether or not nontermination is a factor. We have exemplified how a slice can meet multiple criteria, demonstrating the nuanced interplay between distribution sensitivity and nontermination properties. We introduced the concept of observe-nontermination dependence that captures subtle dependencies, improving slicing accuracy.

We have developed a (bi)simulation-based proof framework that ensures the correctness of the computed slices for all variants. Our (bi)simulation-based proof framework verifies the correctness of these slices, and our algorithm offers efficient incremental computation. These contributions enhance the theoretical and practical understanding of probabilistic programming, providing robust tools for various applications.

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. Abu Naser Masud. Simple and efficient computation of minimal weak control closure. In David Pichardie and Mihaela Sighireanu, editors, *Static Analysis - 27th International Symposium, SAS 2020, Virtual Event, November 18-20, 2020, Proceedings*, volume 12389 of *Lecture Notes in Computer Science*, pages 200–222. Springer, 2020.
15. Abu Naser Masud. Efficient computation of minimal weak and strong control closure. *J. Syst. Softw.*, 184:111140, 2022.
16. Abu Naser Masud. Fast and incremental computation of weak control closure. In Gagandeep Singh and Caterina Urban, editors, *Static Analysis - 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5-7, 2022, Proceedings*, volume 13790 of *Lecture Notes in Computer Science*, pages 325–349. Springer, 2022.
17. Abu Naser Masud and Björn Lisper. On the computation of interprocedural weak control closure. In Bernhard Egger and Aaron Smith, editors, *CC '22: 31st ACM SIGPLAN International Conference on Compiler Construction, Seoul, South Korea, April 2 - 3, 2022*, pages 65–76. ACM, 2022.
18. Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
19. Marcelo Navarro and Federico Olmedo. Slicing of probabilistic programs based on specifications. *Science of Computer Programming*, 220:102822, 2022.

20. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999.
21. 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.
22. 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.
23. 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.
24. 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.
25. 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.
26. 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.
27. Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE'81, page 439–449. IEEE Press, 1981.
28. Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 439–449. IEEE Press, 1981.
29. 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.

A Proof of theorems

For the proofs, we assume in general that the program P_1 has nonzero final distribution, since zero final distribution for all states would imply that the program either never terminates or has no valid outcomes, which would make it uninteresting or useless for practical purposes. We develop some auxiliary lemmas to prove the theorems.

Lemma 1. *Let S_C be a slice set that is closed under $\overset{cd}{\rightarrow}$. Then, for any CFG node n, the set $obs(n)$ is at most a singleton.*

Proof. According to Def. 10, $obs(n) = \{n\}$ if $n \in S_C$, $obs(n) = \emptyset$ if no CFG path from n includes a node from S_C. The lemma holds in both cases.

Assume $n \notin S_C$ and suppose, contrary to the lemma, that there exist distinct nodes n_k and m_l such that $n_k, m_l \in obs(n)$. This implies the existence of two distinct CFG paths: $[n = n_1..n_k]$ and $[n = m_1..m_l]$ such that $n_i, m_j \notin S_C$ for each $1 \leq i < k$ and $1 \leq j < l$, and $n_k, m_l \in S_C$. Without loss of generality, we assume that no predicate node exists in either $[n_2..n_{k-1}]$ or $[m_2..m_{l-1}]$, as we can always choose the closest predicate node to n_k or m_l as n.

Then, node n_k (or m_l) postdominates all nodes between n and n_k (resp. m_l). Given that n has another branch to node m_l (resp. n_k), n is not strictly postdominated by either n_k or m_l. Thus, either $n \xrightarrow{cd} n_k$ or $n \xrightarrow{cd} m_l$ holds. In any case, we arrive at 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 G be the CFG of program P_i for $i = 1, 2$, and let n be any CFG node. If $rv_1(n)$ and $rv_2(n)$ are the sets of RVs of programs P_1 and P_2 at n, then $rv_1(n) = rv_2(n)$.*

Proof. The proof follows from Def. 15 and 11.

Lemma 3. *Let $P_i \vdash \langle n_1, \mu_1, p_1\rangle \xrightarrow{n_1} \langle n_2, \mu_2, p_2\rangle$ be a semantic transition of program P_i for $i = 1, 2$. Then, $\mathsf{rv}(n_1) \supseteq \mathsf{rv}(n_2) \setminus \mathsf{def}(n_1) \cup \mathsf{ref}(n_1)$.*

Proof. According to the continuation criteria in Def. 11, $rv(n') \setminus \mathsf{def}(()n) \subseteq rv(n)$. Moreover, $n_1 \in S_C$ due to the observable transition, and thus $\mathsf{ref}(()n) \subseteq rv(n)$. □

Lemma 4. *Let $P_i \vdash \langle n_1, \mu_1, p_1\rangle \xrightarrow{\tau} \langle n_2, \mu_2, p_2\rangle$ be a semantic transition of program P_i for $i = 1, 2$. Then, $rv(n) = rv(n')$ if n is not a predicate node, and $rv(n) \supseteq rv(n')$ otherwise.*

Proof. As the semantic transition is a silent transition, $n_1 \notin S_C$. In this case, only the continuation criteria in Def. 11 is applicable. As a result, $rv(n) = rv(n')$ if n is not a predicate node. If n is a predicate node, $rv(n) \supseteq rv(n')$ since n has another successor that may contribute different RVs at n. □

Lemma 5. *Let μ_1 and μ_2 be probabilistic stores, V be any subset of program variables, and $V' \subseteq V$. If $\mu_1 \preccurlyeq_V \mu_2$, then $\mu_1 \preccurlyeq_{V'} \mu_2$ holds, i.e.,*

1. *if $\mu_1|_V = \exists q \in [0, 1].q \cdot \mu_2|_V$, then $\mu_1|_{V'} = \exists q \in [0, 1] \cdot q \cdot \mu_2|_{V'}$, and*
2. *if $\mu_1|_V \leq \mu_2|_V$, then $\mu_1|_{V'} \leq \mu_2|_{V'}$.*

Proof. Consider $\mu_1, \mu_2 \colon \Sigma \to [0, 1]$ and $\Sigma : \mathcal{V} \to \mathbb{Z}$. Recall that the set $\sigma|_X$, the restriction of the store $\sigma \in \Sigma$ on a set $X \subseteq \mathcal{V}$, is defined as $\sigma|_X = \cup_{x \in X}\{x \mapsto \sigma(x)\}$. Let $\Sigma_X : X \to \mathbb{Z}$ be a restriction of Σ such that $\sigma|_X \in \Sigma_X$ is the restriction of $\sigma \in \Sigma$ on X. The marginal distribution $\mu_i|_{V'}$ for $i = 1, 2$ is defined as:

$$\mu_i|_{V'}(\sigma_{V'}) = \sum_{\sigma_V \in \Sigma_V, \sigma_V|_{V'} = \sigma_{V'}} \mu_i|_V(\sigma_V)$$

Given the first premise of the lemma, $\mu_1|_V(\sigma_V) = q \cdot \mu_2|_V(\sigma_V)$ holds. Using this, we derive the following equality for the marginal distribution:

$$\begin{aligned}\mu_1|_{V'}(\sigma_{V'}) &= \textstyle\sum_{\sigma_V \in \Sigma_V, \sigma_V|_{V'} = \sigma_{V'}} \mu_1|_V(\sigma_V) \\ &= \textstyle\sum_{\sigma_V \in \Sigma_V, \sigma_V|_{V'} = \sigma_{V'}} q \cdot \mu_2|_V(\sigma_V) \text{[Premise (1) of the lemma]} \\ &= q \cdot \mu_2|_{V'}(\sigma_{V'})\end{aligned}$$

By applying the second premise of the lemma, we obtain the inequality $\mu_1|_{V'}(\sigma_{V'}) \leq \mu_2|_{V'}(\sigma_{V'})$ for all $\sigma_{V'}$. □

Lemma 6. *Let μ_1, μ_2, μ_3 be probabilistic stores, and let V_1, V_2 be subsets of program variables. If $\mu_1 \preccurlyeq_{V_1} \mu_2$, $\mu_2 \preccurlyeq_{V_2} \mu_3$, and $V_1 \subseteq V_2$, then $\mu_1 \preccurlyeq_{V_1} \mu_3$ holds.*

Proof. Assume that $\mu_1 \preccurlyeq_{V_1} \mu_2$, $\mu_2 \preccurlyeq_{V_2} \mu_3$, and $V_1 \subseteq V_2$. By applying Lemma 5, we can derive $\mu_2 \preccurlyeq_{V_1} \mu_3$ from $\mu_2 \preccurlyeq_{V_2} \mu_3$ and $V_1 \subseteq V_2$. Then, the transitivity of the partial-equivalence relation trivially holds.

Lemma 7. *Let $P_i \vdash \langle n_1, \mu_1, p_1\rangle \to \langle n_2, \mu_2, p_2\rangle$ be a semantic transition of program P_i for $i = 1, 2$. If $code_i(n_1)$ is* **skip***, true, or false , then we must have $\mu_1 = \mu_2$.*

Proof. According to the SKIP, COND-T, and COND-F semantic rules in Fig. 5, $\mu_1 = \mu_2$ trivially holds. □

Lemma 8. *Let $P_1 \vdash \langle n, \mu_1, p_1\rangle \xrightarrow{n} \langle n_1', \mu_1', p_1'\rangle$ and $P_2 \vdash \langle n, \mu_2, p_2\rangle \xrightarrow{n} \langle n_2', \mu_2', p_2'\rangle$ be two labeled transitions of P_1 and its slice P_2, and let $V = \mathsf{rv}(n)$ and $V' = \mathsf{rv}(n_1')$. If $\mu_1 \preccurlyeq_V \mu_2$, then $\mu_1' \preccurlyeq_{V'} \mu_2'$ and $n_1' = n_2'$ hold.*

Proof. According to the premise of the lemma, $n \in S_C$. Thus, $code_1(n) = code_2(n)$ due to Def. 11. Let $\Sigma_V : V \to \mathbb{Z}$ be a restriction of Σ to a subset of variables V. We prove $\mu_1' \preccurlyeq_{V'} \mu_2'$ by analyzing the following cases:

- $code_1(n)$ is an assignment statement $x := e$. According to the ASSIGN rule in Fig. 5, we obtain

  $$\mu_1(\sigma) = \mu_1'(\sigma[x \mapsto a])$$

  for all $\sigma \in \Sigma$ where $a = [\![e]\!]\sigma$. Since this equality is point-wise, for any $\sigma_V \in \Sigma_V$, we obtain

  $$\sum_{\sigma \in \Sigma, \sigma|_V = \sigma_V} \mu_1(\sigma) = \sum_{\sigma \in \Sigma, \sigma|_V = \sigma_V} \mu_1'(\sigma[x \mapsto a])$$

  Consequently, $\mu_1|_V(\sigma_V) = \mu_1'|_V(\sigma_V)$ for all $\sigma_V \in \Sigma_V$. Similarly, $\mu_2|_V(\sigma_V) = \mu_2'|_V(\sigma_V)$ for all $\sigma_V \in \Sigma_V$.
  According to the premise of the lemma, $\mu_1 \preccurlyeq_V \mu_2$, which leads to $\mu_1' \preccurlyeq_V \mu_2'$. According to Lemma 3, $V' \setminus \{x\} \subseteq V$, and consequently, $\mu_1' \preccurlyeq_{V' \setminus \{x\}} \mu_2'$ due to Lemma 5. Since $n \in S_C$, $x \in V'$ and $\mathsf{ref}(n) \subseteq V$ according to Def. 15. As $\mu_i(\sigma) = \mu_i'(\sigma[x \mapsto a])$ for all $\sigma \in \Sigma$ and $i = 1, 2$ with $a = [\![e]\!]\sigma$, we can derive the following from $\mu_1' \preccurlyeq_{V' \setminus \{x\}} \mu_2'$:

  $$\mu_1'|_{V'}(\sigma_{V' \setminus \{x\}} \cup \{x \mapsto a\}) = \mu_2'|_{V'}(\sigma_{V' \setminus \{x\}} \cup \{x \mapsto a)$$

  Therefore, $\mu_1' \preccurlyeq_{V'} \mu_2'$.

– $code_1(n)$ is a probabilistic assignment $x :\approx d$. Let $Z = V' \setminus \{x\}$. As $n \in S_C$, $Z \subseteq V$ (Lemma 3) and $x \in V'$ according to Def. 15. Assume that $\mu_1|_V(\sigma_V) = q \cdot \mu_2|_V(\sigma_V)$ due to the premise $\mu_1 \preccurlyeq_V \mu_2$ of the lemma, where $q \in [0,1]$ and $\sigma_V \in \Sigma_V$. Since $Z \subseteq V$, from Lemma 5, for all $\sigma_Z \in \Sigma_Z$,

$$\mu_1|_Z(\sigma_Z) = q \cdot \mu_2|_Z(\sigma_Z)$$

According to the RANDOM rule in Fig. 5, we obtain

$$\mu_i'(\sigma) = \textstyle\sum_{\sigma=\sigma'[x\mapsto v]} \mu_X(\sigma(x)) \cdot \mu_i(\sigma')$$

for all $\sigma \in \Sigma$ and $i = 1, 2$, where the distribution function d assigns random values $v = \sigma(x)$ with probabilities given by μ_X. The marginal distribution $\mu_i'|_{V'}$, $i = 1, 2$ for all $\sigma_{V'} \in \Sigma_{V'}$ is as follows:

$$\mu_i'|_{V'}(\sigma_{V'}) = \sum_{\sigma\in\Sigma, \sigma_{V'}=\sigma|_{V'}} \mu_X(\sigma_{V'}(x)) \cdot \mu_i|_Z(\sigma|_Z).$$

Thus,

$$\begin{aligned}
\mu_1'|_{V'}(\sigma_{V'}) &= \textstyle\sum_{\sigma\in\Sigma, \sigma_{V'}=\sigma|_{V'}} \mu_X(\sigma_{V'}(x)) \cdot \mu_1|_Z(\sigma|_Z) \\
&= \textstyle\sum_{\sigma\in\Sigma, \sigma_{V'}=\sigma|_{V'}} \mu_X(\sigma_{V'}(x)) \cdot q \cdot \mu_2|_Z(\sigma|_Z) \\
&= q \cdot \textstyle\sum_{\sigma\in\Sigma, \sigma_{V'}=\sigma|_{V'}} \mu_X(\sigma_{V'}(x)) \cdot \mu_2|_Z(\sigma|_Z) \\
&= q \cdot \mu_2'|_{V'}(\sigma_{V'})
\end{aligned}$$

Thus $\mu_1' \preccurlyeq_{V'} \mu_2'$ holds in this case.

– $code_1(n) =$ ***observe*** b. The relation $\mu_1 \preccurlyeq_V \mu_2$, which holds due to the precondition of the lemma, implies $\mu_1|_V = q \cdot \mu_2|_V$ for some $q \in [0,1]$.
According to the OBSERVE rule in Fig. 5, for all $\sigma \in \Sigma$, either $\mu_i(\sigma) = \mu_i'(\sigma)$ if $[\![b]\!]\sigma = \mathit{true}$, and $\mu_i'(\sigma) = 0$ otherwise for $i = 1, 2$. Let $A \subseteq \Sigma$ be the set of states of node n where μ_i equals μ_i'.
Then, we derive the following calculation for all $\sigma_V = \sigma|_V$ where $\sigma \in \Sigma$:

$$\begin{array}{ll}
\mu_1|_V(\sigma_V) & = q \cdot \mu_2|_V(\sigma_V) \\
\textstyle\sum_{\sigma\in\Sigma, \sigma_V=\sigma|_V} \mu_1(\sigma) & = q \cdot \textstyle\sum_{\sigma\in\Sigma, \sigma_V=\sigma|_V} \mu_2(\sigma) \\
\textstyle\sum_{\sigma\in A, \sigma_V=\sigma|_V} \mu_1(\sigma) & = q \cdot \textstyle\sum_{\sigma\in A, \sigma_V=\sigma|_V} \mu_2(\sigma) \text{ [same relation holds for } A \subseteq \Sigma] \\
\mu_1'|_V(\sigma_V) & = q \cdot \mu_2'|_V(\sigma_V)
\end{array}$$

Thus, $\mu_1' \preccurlyeq_V \mu_2'$ holds. According to Lemma 3, $V' \subseteq V$, and consequently, $\mu_1' \preccurlyeq_{V'} \mu_2'$ holds due to Lemma 5.

– $code_1(n) = b$. The proof is similar to the case for $code_1(n) =$ ***observe*** b due to the fact that the transition of probabilistic states in COND-T, COND-F, and OBSERVE rules in Fig. 5 are similar, and $V' \subseteq V$ (Lemma 3). Thus, $\mu_1' \preccurlyeq_{V'} \mu_2'$ follows from $\mu_1 \preccurlyeq_V \mu_2$ as above.

– $code_1(n) =$ ***skip***. This case is not possible since $n \in S_C$.

Node n has only one successor in all cases except when $code_1(n) = b$. For any $\sigma \in \Sigma$, if $[\![b]\!]\sigma$ evaluates to *true*/*false* in P_1, it also evaluates to the same in P_2, and vice versa. While the distributions $\mu_1(\sigma)$ and $\mu_2(\sigma)$ may differ, they satisfy $\mu_1(\sigma) \preccurlyeq_V \mu_2(\sigma)$. Therefore, $n_1' = n_2'$ in all cases. □

Lemma 9. *Let* $P_1 \vdash \langle n_1, \mu_1, p_1\rangle \xrightarrow{\tau} \langle n_2, \mu_2, p_2\rangle$*, let* $V = rv(n_1)$ *and* $V' = rv(n_2)$*, and let* μ_3 *be any probabilistic store. If* $\mu_1 \preccurlyeq_V \mu_3$*, then* $\mu_2 \preccurlyeq_{V'} \mu_3$ *holds.*

Proof. Since the transition is a silent transition, $n_1 \notin S_C$. We conduct the following case analysis where $V = V'$ holds for all but the last case according to Lemma 4.

- $code_1(n_1) = x := e$. According to the ASSIGN rule in Fig. 5, we obtain $\mu_1(\sigma) = \mu_2(\sigma[x \mapsto a])$ for all $\sigma \in \Sigma$ where $a = [\![e]\!]\sigma$. For any $\sigma_V \in \Sigma_V$, we obtain
$$\sum_{\sigma\in\Sigma,\sigma|_V=\sigma_V} \mu_1(\sigma) = \sum_{\sigma\in\Sigma,\sigma|_V=\sigma_V} \mu_2(\sigma[x \mapsto a]),$$
and consequently, $\mu_1|_V(\sigma_V) = \mu_2|_V(\sigma_V)$. We conclude $\mu_2|_V = q \cdot \mu_3|_V$ if $\mu_1 \preccurlyeq_V \mu_3$ holds due to $\mu_1|_V = \exists q \in [0,1].q \cdot \mu_3|_V$.
- $code_1(n_1) = x :\approx d$. According to the RANDOM rule in Fig. 5,
$$\mu_2(\sigma) = \textstyle\sum_{\sigma=\sigma'[x\mapsto v]} \mu_X(v) \cdot \mu_1(\sigma')$$
for all $\sigma \in \Sigma$, and the distribution function d assigns random values $v = \sigma(x)$ with probabilities given by μ_X. $x \notin V$ as otherwise we would have $n_1 \in S_C$. The marginal distribution $\mu_2|_V$ for all $\sigma_V \in \Sigma_V$ is as follows:
$$\mu_2|_V(\sigma_V) = \sum_{\sigma\in\Sigma,\sigma_V=\sigma|_V} \mu_X(\sigma_V(x)) \cdot \mu_1(\sigma).$$
If $\mu_1|_V = q \cdot \mu_3|_V$ for some $q \in [0,1]$, we can derive the following:
$$\begin{aligned}\mu_2|_V(\sigma_V) &= \textstyle\sum_{\sigma\in\Sigma,\sigma_V=\sigma|_V} \mu_X(\sigma_V(x)) \cdot \mu_1(\sigma)\\ &= \mu_X(\sigma_V(x)) \cdot \textstyle\sum_{\sigma\in\Sigma,\sigma_V=\sigma|_V} \mu_1(\sigma)\\ &= \mu_X(\sigma_V(x)) \cdot \mu_1|_V(\sigma_V)\\ &= \mu_X(\sigma_V(x)) \cdot q \cdot \mu_3|_V(\sigma_V)\\ &= q' \cdot \mu_3|_V(\sigma_V) \text{ for some } q' \in [0,1]\end{aligned}$$
- $code_1(n_1) =$ ***observe*** b. According to the OBSERVE rule in Fig. 5, for all $\sigma \in \Sigma$, either $\mu_2(\sigma) = \mu_1(\sigma)$ if $[\![b]\!]\sigma = true$, and $\mu_2(\sigma) = 0$ otherwise. Let $A \subseteq \Sigma$ be the set of configurations where μ_2 equals μ_1. Since $\mu_2(\sigma) = \mu_1(\sigma)$ for $\sigma \in A$, we can derive the following:
$$\begin{aligned}\mu_2|_V(\sigma_V) &= \textstyle\sum_{\sigma\in A,\sigma_V=\sigma|_V} \mu_1(\sigma)\\ &= \frac{\sum_{\sigma\in A,\sigma_V=\sigma|_V} \mu_1(\sigma)}{\sum_{\sigma\in\Sigma,\sigma_V=\sigma|_V} \mu_1(\sigma)} \textstyle\sum_{\sigma\in\Sigma,\sigma_V=\sigma|_V} \mu_1(\sigma)\\ &= r \cdot \mu_1|_V(\sigma_V)\\ &= r \cdot q \cdot \mu_3|_V(\sigma_V)\\ &= q' \cdot \mu_3|_V(\sigma_V)\end{aligned}$$
where, $r \in [0,1]$ is essentially the proportion of the mass of $\mu_1|_V$ that is accounted for by the subset A, and $q' = r \cdot q \in [0,1]$.

- $code_1(n_1) =$ ***skip*** . According to the SKIP rule in Fig. 5, $\mu_1 = \mu_2$. Thus, $\mu_2 \preccurlyeq_{V'} \mu_3$ follows from $\mu_1 \preccurlyeq_V \mu_3$.
- $code_1(n_1) = b$. By following the calculation for the case $code_1(n_1) =$ ***observe*** b, since the transition of probabilistic states in COND-T, COND-F, and OBSERVE rules in Fig. 5 are similar, we can prove that $\mu_2 \preccurlyeq_V \mu_3$. According to Lemma 3, $V' \subseteq V$. Thus, $\mu_2 \preccurlyeq_{V'} \mu_3$ holds according to Lemma 5.

□

Lemma 10. *Let S_C be the slice set which is closed under $\xrightarrow{cd}$, and let Γ_1 be any configuration of program P_2 at CFG node n_1. If $obs(n_1) = n_k$ and $n_1 \neq n_k$, then there exists configuration Γ_k of node n_k such that $P_2 \vdash \Gamma_1 \overset{\tau}{\Rightarrow} \Gamma_k$.*

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. 11. 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. 5, 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 labeled transition $P_2 \vdash \Gamma_i \xrightarrow{\tau} \Gamma_{i+1}$ for $1 \leq i < k$. Consequently, $P_2 \vdash \Gamma_1 \overset{\tau}{\Rightarrow} \Gamma_k$ holds. □

Lemma 11. *Let Γ_1 and Γ_2 be valid configurations of programs P_1 and P_2 such that $\Gamma_1 \simeq \Gamma_2$. If there exists a transition $P_1 \vdash \Gamma_1 \xrightarrow{n_1} \Gamma_3$, then a transition $P_2 \vdash \Gamma_2 \overset{n_1}{\Rightarrow} \Gamma_4$ also exists such that $\Gamma_3 \simeq \Gamma_4$.*

Proof. Let $\mu_i = store(\Gamma_i)$ and $n_i = node(\Gamma_i)$ 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 $P_1 \vdash \Gamma_1 \xrightarrow{n_1} \Gamma_3$ yields $n_1 \in S_C$, and consequently, $obs(n_1) = n_1$. The relation $\Gamma_1 \simeq \Gamma_2$ yields $obs(n_1) = obs(n_2) = n_1$.

Let $n_2 \neq n_1$. By Lemma 10, $obs(n_2) = n_1$ 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 $\Gamma^1, \ldots, \Gamma^k$ of P_2 exists such that $n^i = node(\Gamma^i)$ and $\mu^i = store(\Gamma^i)$ for $i = 1, \ldots, k$ where $\Gamma^1 = \Gamma_2$ and $n^k = n_1$. The relation $P_2 \vdash \Gamma^i \xrightarrow{\tau} \Gamma^{i+1}$ 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. 11. Consequently, $\mu^i = \mu^{i+1}$ for all $1 \leq i \leq k$ due to Lemma 7.

As $\Gamma_1 \simeq \Gamma_2$, we have $\mu_1 \preccurlyeq_V \mu_2$ where V is the set of RVs at n_1 in P_1. Since $\mu_2 = \mu^1$ and $\mu^1 = \mu^k$, we obtain $\mu_1 \preccurlyeq_V \mu^k$. If $n_2 = n_1$, then $\Gamma_2 = \Gamma^k$ and $\mu_1 \preccurlyeq_V \mu^k$ trivially hold due to the precondition of the lemma. Hence, either exists $q \in [0, 1]$ such that $\mu_1|_V = q.\mu^k|_V$ 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, if the transition $P_1 \vdash \Gamma_1 \xrightarrow{n_1} \Gamma_3$ holds, the probability of P_2 taking the same path as P_1 from n_1 is higher even if n_1 is a predicate node or represents an ***observe*** statement. Thus, a configuration Γ_4 of

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

As (i) $obs(n_3) = obs(n_3)$ holds trivially, and (ii) $\mu_3 \preccurlyeq_{rv(n_3)} \mu_4$ according to Lemma 8, the relation $\Gamma_3 \simeq \Gamma_4$ holds according to the definitions of $\simeq$. □

Lemma 12. *Assume that S_C is closed under $\xrightarrow{dd}$, $\xrightarrow{cd}$, and either $\xrightarrow{obntd}$ or $\xrightarrow{obsd}$. Let Γ_1 and Γ_2 be valid configurations of programs P_1 and P_2 such that $\Gamma_1 \simeq \Gamma_2$. 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 $\Gamma_3 \simeq \Gamma_4$.*

Proof. Consider that the transition $P_1 \vdash \Gamma_1 \overset{n}{\Rightarrow} \Gamma_3$ exists. Thus, a sequence of silent transitions $P_1 \vdash \Gamma^i \xrightarrow{\tau} \Gamma^{i+1}$ for $1 \leq i < k$, and the transition $P_1 \vdash \Gamma^k \xrightarrow{n} \Gamma_3$ exist such that $\Gamma^1 = \Gamma_1$. Let $\mu_i = store(\Gamma_i)$, $n_i = node(\Gamma_i)$, $\mu^j = store(\Gamma^j)$, and $n^j = node(\Gamma^j)$ for $1 \leq i \leq 4$ and $1 \leq j \leq k$. First, we establish the relation $\Gamma^k \simeq \Gamma_2$.

The silent transition $P_1 \vdash \Gamma^i \xrightarrow{\tau} \Gamma^{i+1}$ implies $n^i \notin S_C$. The transition $P_1 \vdash \Gamma^k \xrightarrow{n} \Gamma^3$ indicates that $n^k = n \in S_C$. Consequently, $obs(n_1) = obs(n^k) = n$. The relation $\Gamma_1 \simeq \Gamma_2$ implies $obs(n_1) = obs(n_2)$, and hence $obs(n^k) = obs(n_2)$. This proves the first precondition for the relation $\Gamma^k \simeq \Gamma_2$.

$\Gamma_1 \simeq \Gamma_2$ implies that $\mu^1 \preccurlyeq_V \mu_2$ where $V = rv(n_1)$. By repeatedly applying Lemma 9 for the transition $P_1 \vdash \Gamma^i \xrightarrow{\tau} \Gamma^{i+1}$ for $1 \leq i < k$ sequentially, we conclude $\mu^k \preccurlyeq_{V'} \mu_2$, where $V' = rv(n^k)$. Consequently, $\Gamma^k \simeq \Gamma_2$ holds. As $P_1 \vdash \Gamma^k \xrightarrow{n} \Gamma_3$ exists, the lemma is proven by lemma 11. □

Lemma 13. *Assume that S_C is closed under $\xrightarrow{cd}$. Let Γ_1 be any configuration of program P_1 at node n_1. If there exists a node n_k such that $obs(n_1) = n_k$ and $n_1 \neq n_k$, then there exists a configuration Γ_k at node n_k such that $P_1 \vdash \Gamma_1 \overset{\tau}{\Rightarrow} \Gamma_k$ and $store(\Gamma_k) \preccurlyeq_V store(\Gamma_1)$, where $V = rv(n_k)$.*

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 $1 \leq i < k$, and $n_k \in S_C$. Let $\Gamma_i = \langle n_i, \mu_i, p_i \rangle$ be any configuration of node n_i. Let $V_i = rv(n_i)$ for $1 \leq i \leq k$, and let $V_k = V$.

First, we prove that $P_1 \vdash \Gamma_1 \overset{\tau}{\Rightarrow} \Gamma_k$ exists. If n_i is a skip, assignment, or random assignment instruction, then there always exists a transition $P_1 \vdash \Gamma_i \to \Gamma_{i+1}$. Let $code_1(n_i) = \textbf{\textit{observe}}\ b$. There must exist a state $\sigma \in \Sigma$ such that $[\![b]\!]\sigma = true$, thereby ensuring a transition $P_1 \vdash \Gamma_i \to \Gamma_{i+1}$ exists, as otherwise, it contradicts our general assumption that the program has nonzero final distribution and practically useful. If $code_1(n_i) = b$, then a transition $P_1 \vdash \Gamma_i \to \Gamma_{i+1}$ exists, where either $n_{i+1} = \mathsf{succ_T}(n_i)$ or $n_{i+1} = \mathsf{succ_F}(n_i)$. Since $n_i \notin S_C$, we have the transition $P_1 \vdash \Gamma_i \xrightarrow{\tau} \Gamma_{i+1}$, and thus, $P_1 \vdash \Gamma_1 \overset{\tau}{\Rightarrow} \Gamma_k$ exists.

The relation $\mu_1 \preccurlyeq_{V_1} \mu_1$ holds trivially. By applying Lemma 9 sequentially to the transitions $P_1 \vdash \Gamma_i \xrightarrow{\tau} \Gamma_{i+1}$ for $i = 1, \ldots, k-1$, we infer that $\mu_i \preccurlyeq_{V_i} \mu_1$, and consequently, $\mu_k \preccurlyeq_{V_k} \mu_1$ holds. □

Lemma 14. *Assume that S_C is closed under $\xrightarrow{cd}$. Let Γ_1 and Γ_2 be valid configurations of programs P_1 and P_2 respectively, such that $\Gamma_1 \simeq \Gamma_2$. If there exists*

a transition $P_2 \vdash \Gamma_2 \xrightarrow{n_2} \Gamma_4$*, then there also exists a transition* $P_1 \vdash \Gamma_1 \overset{n_2}{\Rightarrow} \Gamma_3$ *such that* $\Gamma_3 \simeq \Gamma_4$*.*

Proof. Let $\mu_i = store(\Gamma_i)$ and $n_i = node(\Gamma_i)$ for $1 \leq i \leq 4$. Given the transition $P_2 \vdash \Gamma_2 \xrightarrow{n_2} \Gamma_4$, we have $n_2 \in S_C$, and thus $obs(n_2) = n_2$. Since $\Gamma_1 \simeq \Gamma_2$, it follows that $obs(n_1) = obs(n_2) = n_2$.

Assume $n_1 \neq n_2$. By Lemma 13, there exists a configuration Γ at n_2 such that the labeled transition $P_1 \vdash \Gamma_1 \overset{\tau}{\Rightarrow} \Gamma$ holds, and $\mu \preccurlyeq_V \mu_1$, where $V = rv(node(\Gamma))$ and $\mu = store(\Gamma)$. Since $obs(n_1) = n_2$, $n_1 \notin S_C$, and $V \subseteq V' = rv(n_1)$ due to the continuation criteria in Def. 15). We have $\mu_1 \preccurlyeq_{V'} \mu_2$ due to $\Gamma_1 \simeq \Gamma_2$. Thus, $\mu \preccurlyeq_V \mu_2$ holds (Lemma 6). If $n_2 = n_1$, then $\Gamma_1 = \Gamma$ and $\mu \preccurlyeq_V \mu_2$ holds due to the precondition of the lemma.

According to Lemma 8, there exists a transition $P_1 \vdash \Gamma \xrightarrow{n_2} \Gamma_3$ such that $n_3 = n_4$ and $\mu_3 \preccurlyeq_{V''} \mu_4$ where $V'' = rv(n_3)$. Therefore, the transition $P_1 \vdash \Gamma \overset{n_2}{\Rightarrow} \Gamma_3$ exists. Furthermore, since $n_3 = n_4$, $obs(n_3) = obs(n_4)$ trivially holds, and consequently, $\Gamma_3 \simeq \Gamma_4$ holds according to the definitions of $\simeq$. □

Lemma 15. *Assume that* S_C *is closed under either* $\xrightarrow{scd}$ *or both* $\xrightarrow{wcd}$ *and* $\xrightarrow{obntd}$*. Let* Γ_1 *and* Γ_2 *be valid configurations of programs* P_1 *and* P_2 *such that* $\Gamma_1 \simeq \Gamma_2$*. 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* $\Gamma_3 \simeq \Gamma_4$*.*

Proof. Let $\mu_i = store(\Gamma_i)$ and $n_i = node(\Gamma_i)$ for $1 \leq i \leq 4$. If $n_2 = n$, the lemma follows directly from Lemma 14. Assume $n_2 \neq n$ and that $P_2 \vdash \Gamma_2 \overset{n}{\Rightarrow} \Gamma_4$. Therefore, there exists a sequence of silent transitions $P_2 \vdash \Gamma^i \xrightarrow{\tau} \Gamma^{i+1}$ for $1 \leq i < k$ and a transition $P_2 \vdash \Gamma^k \xrightarrow{n} \Gamma_4$ such that $\Gamma^1 = \Gamma_2$. Let $\mu^i = store(\Gamma^i)$ and $n^i = node(\Gamma^i)$ for $1 \leq i < k$.

First, we establish the relation $\Gamma_1 \simeq \Gamma^k$. 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 $\Gamma_1 \simeq \Gamma_2$ implies $obs(n_1) = obs(n_2)$, and hence $obs(n_1) = obs(n^k)$. This proves the first precondition for the relation $\Gamma_1 \simeq \Gamma^k$.

Next, we establish the relation $\mu_1 \preccurlyeq_V \mu^k$, where $V = rv(n_1)$. We have one of the following scenarios:

1. S_C is closed under $\xrightarrow{scd}$. Since $obs(n_2) = n = n^k$, all CFG paths from n_2 must converge at n^k before going through S_C. There is no diverging path from n^i for any $1 \leq i < k$, as otherwise n^i would be included in S_C due to $\xrightarrow{scd}$.
2. S_C is closed under $\xrightarrow{wcd}$ and $\xrightarrow{obntd}$. There may have a diverging branch from node n^i for any $1 \leq i < k$. However, either this branch does not include a node from S_C or it does after reaching n^i. As otherwise, n^i would be included in S_C due to $\xrightarrow{wcd}$. Alternatively, no diverging branch from any node n^i exists that affect the slicing criterion. Otherwise, n^i would be included in S_C due to $\xrightarrow{obntd}$.

In any case, $n^i \notin S_C$, and $code_2(n^i)$ is ***skip***, *true*, or *false* according to Def. 11. By successive application of Lemma 7, we obtain $\mu_2 = \mu^k$. Therefore, $\mu_1 \preccurlyeq_V \mu^k$ follows from the relation $\mu_1 \preccurlyeq_V \mu_2$, and consequently, $\Gamma_1 \simeq \Gamma^k$ holds.

Since $P_2 \vdash \Gamma^k \xrightarrow{n} \Gamma_4$ holds, the lemma is proven by Lemma 14.

□

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

1. Assume that there exists a labeled transition $P_1 \vdash \Gamma_1 \overset{n}{\Rightarrow} \Gamma_1'$. By lemma 12, $P_2 \vdash \Gamma_2 \overset{n}{\Rightarrow} \Gamma_2'$ also exists such that $\Gamma_1' \simeq \Gamma_2'$. This proves that $\simeq$ 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 either $\xrightarrow{scd}$, or both $\xrightarrow{wcd}$ and $\xrightarrow{obntd}$. For the other direction, let us assume that the labeled transition $P_2 \vdash \Gamma_2 \overset{n}{\Rightarrow} \Gamma_4$ exists. By Lemma 15, a transition $P_1 \vdash \Gamma_1 \overset{n}{\Rightarrow} \Gamma_3$ also exists such that $\Gamma_3 \simeq \Gamma_4$.

□

Proof (Theorem 2). Let $\Gamma_0 = \langle start, \mu_0, 0\rangle$ and $\Gamma_0' = \langle start, \mu_0', 0\rangle$ be the initial configurations of P_1 and P_2, and let V be the set (and subset) of program variables in P_2 (resp. P_1). We can safely assume the following: (1) $obs(start) = obs(start)$ trivially holds, (2) both programs start execution from equivalent probabilistic stores, and thus $\mu_0|_V = \mu_0'|_V$ holds, (3) $\mathsf{rv}(node(\Gamma_0)) = \mathsf{rv}(node(\Gamma_0')) \subseteq V$, and consequently, (4) $\mu_0 \preccurlyeq_{\mathsf{rv}(start)} \mu_0'$ according to Lemma 5. Thus, $\Gamma_0 \simeq \Gamma_0'$ holds (Def. 16). According to Theorem 1, $\simeq$ is either a *weak simulation* or a *weak bisimulation* relation.

If $\simeq$ is a *weak simulation* relation, for any finite sequence of configurations $\Gamma_0, \ldots, \Gamma_k$ of P_1, a corresponding sequence $\Gamma_0', \ldots, \Gamma_k'$ of P_2 exists such that Γ_k, Γ_k' are final configurations, where $P_1 \vdash \Gamma_i \overset{n_i}{\Rightarrow} \Gamma_{i+1}$ and $P_2 \vdash \Gamma_i' \overset{n_i}{\Rightarrow} \Gamma_{i+1}'$ hold for all $0 \leq i < k-1$, and $P_1 \vdash \Gamma_{k-1} \overset{\tau}{\Rightarrow} \Gamma_k$ and $P_2 \vdash \Gamma_{k-1}' \overset{\tau}{\Rightarrow} \Gamma_k'$ hold. We can deduce from Theorem 1 that $\Gamma_{k-1} \simeq \Gamma_{k-1}$, which implies $store(\Gamma_{k-1}) \preccurlyeq_{V'} store(\Gamma_{k-1}')$ for $V' = \mathsf{rv}(node(\Gamma_{k-1}))$. As $P_1 \vdash \Gamma_{k-1} \overset{\tau}{\Rightarrow} \Gamma_k$, we successively apply Lemma 9 and obtain $store(\Gamma_k) \preccurlyeq_{V''} store(\Gamma_{k-1}')$ for $V'' = \mathsf{rv}(node(\Gamma_k))$. As $P_2 \vdash \Gamma_{k-1}' \overset{\tau}{\Rightarrow} \Gamma_k'$, we have a sequence of silent transitions $P_2 \vdash \Gamma_k^i \xrightarrow{\tau} \Gamma_k^{i+1}$ for $1 \leq i \leq l$ such that $\Gamma_k^1 = \Gamma_{k-1}'$, $\Gamma_k^l = \Gamma_k'$. Let $m_i = node(\Gamma_k^i)$ and m_k is the final return node. $m_i \notin S_C$ for all i due to the silent transitions, $\mathsf{code}_2(m_i)$ is ***skip***, *true*, or *false* (Def. 11), and thus $store(\Gamma_k^i) = store(\Gamma_k^{i+1})$ according to Lemma 7. Thus, we conclude that $store(\Gamma_k) \preccurlyeq_{V''} store(\Gamma_k')$. If $\simeq$ is a *weak bisimulation* relation, the reverse is also true. In particular, for every $P_2 \vdash \Gamma_i' \overset{n_i}{\Rightarrow} \Gamma_{i+1}'$, there exists $P_1 \vdash \Gamma_i \overset{n_i}{\Rightarrow} \Gamma_{i+1}$ for $1 \leq i < k$ such that $\Gamma_i \simeq \Gamma_i'$ holds. By similar arguments as above, we can conclude that $store(\Gamma_k) \preccurlyeq_{V''} store(\Gamma_k')$ holds. Since $n_k \in C$, and $V'' = \mathsf{rv}(n_k) = \mathsf{ref}(n_k)$ according to Def. 15, this implies that there exists $q \in [0,1]$ such that $([\![P_1]\!]\mu_0)|_{V''} = q \cdot ([\![P_2]\!]\mu_0')|_{V''}$.

Next, we have the following cases:

1. Assume that S_C is closed under $\overset{dd}{\rightarrow}$, $\overset{scd}{\rightarrow}$ and $\xrightarrow{obntd}$. Thus, $\simeq$ is a weak bisimulation relation (Theorem 1). Any node that is part of a nonterminating execution is captured in S_C due to the relation $\overset{scd}{\rightarrow}$. Thus, for any infinite sequence of configurations $\Gamma_0, \Gamma_1, \ldots$ of P_1, there exists an infinite sequence of configurations $\Gamma'_0, \Gamma'_1, \ldots$ of P_2, and vice versa, such that $P_1 \vdash \Gamma_i \overset{n_i}{\Rightarrow} \Gamma_{i+1}$, $P_2 \vdash \Gamma'_i \overset{n_i}{\Rightarrow} \Gamma'_{i+1}$, and $\Gamma_i \simeq \Gamma'_i$ for all $i \geq 0$. Assume that the execution diverges from node n_i. The transition $P_1 \vdash \Gamma_i \overset{n_i}{\Rightarrow} \Gamma_{i+1}$ implies $P_1 \vdash \Gamma_i \overset{\tau}{\Rightarrow} \Gamma$ and $P_1 \vdash \Gamma \overset{n_i}{\rightarrow} \Gamma_{i+1}$. Similarly, we have $P_2 \vdash \Gamma'_i \overset{\tau}{\Rightarrow} \Gamma'$ and $P_1 \vdash \Gamma' \overset{n_i}{\rightarrow} \Gamma'_{i+1}$. Thus, $store(\Gamma_i) \preccurlyeq_S store(\Gamma'_i)$ and $store(\Gamma'_i) \preccurlyeq_S store(\Gamma_i)$ hold, where $S = \mathsf{rv}(node(\Gamma_i))$ according to Lemma 8. This implies that $store(\Gamma'_i)|_S = store(\Gamma_i)|_S$. Thus, the nonterminating execution in P_1 and P_2 accumulates the same distribution mass. Therefore, $\mathsf{Pr}[P_1(\mu_0) \in \Uparrow] = \mathsf{Pr}[P_2(\mu'_0) \in \Uparrow]$. Also, $(\llbracket P_1 \rrbracket \mu_0)|_{V''} = q \cdot (\llbracket P_2 \rrbracket \mu'_0)|_{V''}$ for $V'' = \mathsf{rv}(n_k)$ as shown above. Thus, P_2 is a nontermination sensitive slice of P_1 according to Eq. 1.
2. Assume that S_C is closed under $\overset{dd}{\rightarrow}$, $\overset{wcd}{\rightarrow}$ and $\xrightarrow{obntd}$. Then, according to Theorem 1, $\simeq$ is a weak bisimulation relation, and $(\llbracket P_1 \rrbracket \mu_0)|_{V''} = q \cdot (\llbracket P_2 \rrbracket \mu'_0)|_{V''}$ holds for $V'' = \mathsf{rv}(n_k)$ and $q \in [0, 1]$. By using the same argument as (1) above, we can show that for all diverging executions at node n_i in both P_1 and P_2, $store(\Gamma'_i)|_S = store(\Gamma_i)|_S$ where $S = \mathsf{rv}(node(\Gamma_i))$, and same distribution mass for nontermination is accumulated in this case. However, since S_C is closed under $\overset{wcd}{\rightarrow}$ and $\xrightarrow{obntd}$, not all node n_i involved in diverging execution is included in S_C. While P_1 accumulates part of the distribution mass for nonterminating execution from n_i, no nonterminating execution exists from n_i in P_2 since $codegn_i 2$ is *true* or *false* that leads an execution from n_i to $obs(n_i)$. As a result, $\mathsf{Pr}[P_1(\mu_0) \in \Uparrow] \geq \mathsf{Pr}[P_2(\mu'_0) \in \Uparrow]$. Thus, P_2 is a nontermination insensitive distribution sensitive slice of P_1 according to Eq. 3.
3. Assume that S_C is closed under $\overset{dd}{\rightarrow}$, $\overset{wcd}{\rightarrow}$ and $\xrightarrow{obsd}$. Then, $\simeq$ is a weak simulation relation (Theorem 1). Consider any node n_i such that $\mathsf{code}_1(n_i) = b$ that may involve in nonterminating execution in P_1. We have the following cases:
   - $n_i \notin S_C$. Therefore, $\mathsf{code}_2(n_i)$ is set to *true* or *false* , which leads all executions of P_2 going through n_i to the closest observable node $obs(n_i)$ (see Def. 11). Thus, n_i cannot cause any nonterminating execution in P_2. On the other hand, P_1 may have nonterminating execution due to n_i not being captured in S_C, which is only closed under $\overset{wcd}{\rightarrow}$ and $\xrightarrow{obsd}$.
   - $n_i \in S_C$. There may have diverging executions through n_i in both P_1 and P_2. However, we argue that it is impossible to have a diverging execution from n_i in P_2 but not in P_1. If so, assume that there exist sequences of configurations $\Gamma_0, \ldots, \Gamma_k, \ldots$ and $\Gamma'_0, \ldots, \Gamma'_k, \ldots$ such that $P_1 \vdash \Gamma_i \overset{n_i}{\Rightarrow} \Gamma_{i+1}$, $P_2 \vdash \Gamma'_i \overset{n_i}{\Rightarrow} \Gamma'_{i+1}$ for all $1 \leq i < k$, but $P_1 \vdash \Gamma_k \overset{n}{\Rightarrow} \Gamma_{k+1}$, $P_2 \vdash \Gamma'_k \overset{n'}{\Rightarrow} \Gamma'_{k+1}$, and $n \neq n'$. This implies that there exists $\sigma \in \Sigma$ for P_1 that has no corresponding store $\sigma' \in \Sigma_V$ for P_2 at n_k such that $\llbracket b \rrbracket \sigma = \llbracket b \rrbracket \sigma'$. As S_C is closed under $\overset{dd}{\rightarrow}$, this is impossible.

Thus, P_1 may have more nonterminating executions than P_2, and therefore $\mathsf{Pr}[P_1(\mu_0) \in \Uparrow] \geq \mathsf{Pr}[P_2(\mu_0') \in \Uparrow]$. Moreover, since $\simeq$ is a weak simulation relation, for every final configuration Γ_k of P_1, a corresponding final configuration Γ_k' of P_2 exists such that $\mathit{store}(\Gamma_k) \preccurlyeq_{V''} \mathit{store}(\Gamma_k')$ holds as proved above. However, the reverse may not be true always since P_2 may have a final configuration Γ_k' and a corresponding execution in P_1 diverges. Thus, there may have a store $\sigma_V \in \Sigma_V$ such that $\mathit{store}(\Gamma_k')(\sigma_V) > 0$, but a corresponding store $\sigma \in \Sigma$ exists such that $\mathit{store}(\Gamma_k)(\sigma) = 0$. Thus, $([\![P_1]\!]\mu_0)|_{V''} \leq ([\![P_2]\!]\mu_0')|_{V''}$ holds, but $([\![P_1]\!]\mu_0)|_{V''} = q \cdot ([\![P_2]\!]\mu_0')|_{V''}$ may not hold for $V'' = \mathsf{rv}(n_k)$ and $q \in [0, 1]$. As Eq. 2 holds but Eq. 3 does not, the slice P_2 is a nontermination insensitive distribution insensitive slice of P_1.

□