A Theory of Slicing for Probabilistic Control Flow Graphs

Torben Amtoft[1] and Anindya Banerjee *[2]

[1] Kansas State University, Manhattan, Kansas, USA
[2] IMDEA Software Institute, Madrid, Spain

Abstract. We present a theory for slicing probabilistic imperative programs —containing random assignment and "observe" statements— represented as control flow graphs whose nodes transform probability distributions. We show that such a representation allows direct adaptation of standard machinery such as data and control dependence, postdominators, relevant variables, *etc.* to the probabilistic setting. We separate the specification of slicing from its implementation: first we develop syntactic conditions that a slice must satisfy; next we prove that any such slice is semantically correct; finally we give an algorithm to compute the least slice. A key feature of our syntactic conditions is that they involve two disjoint slices such that the variables of one slice are *probabilistically independent* of the variables of the other. This leads directly to a proof of correctness of probabilistic slicing.

1 Introduction

The task of program slicing [14,12] is to remove the parts of a program that are irrelevant in a given context. This paper addresses slicing of probabilistic imperative programs which, in addition to the usual control structures, contain "random assignment" and "observe" statements. The former assign random values from a given distribution to variables. The latter remove undesirable combinations of values, a feature which can be used to bias the variables according to real world observations. The excellent survey by Gordon *et al.* [6] depicts many applications of probabilistic programs.

Program slicing of deterministic imperative programs is increasingly well understood [10,3,11,1,5]. A basic notion is that if the slice contains a program point which depends on some other program points then these also should be included in the slice; here "depends" typically encompasses data dependence and control dependence. However, Hur *et al.* [7] recently demonstrated that in the presence of random assignments and observations, standard notions of data and control dependence no longer suffice for semantically correct (backward) slicing. They develop a denotational framework in which they prove correct an algorithm for program slicing. In contrast, this paper shows how *classical notions of depen-*

* Research supported by the US National Science Foundation (NSF). Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

dence can be extended to give a semantic foundation for the (backward) slicing of probabilistic programs. The paper's key contributions are:

- A formulation of probabilistic slicing in terms of probabilistic control flow graphs (Sect. 3) that allows direct adaptation of standard machinery such as data and control dependence, postdominators, relevant variables, *etc.* to the probabilistic setting. We also provide a novel operational semantics of probabilistic control flow graphs (Sect. 4): written $(v, D) \Rightarrow (v', D')$, the semantics states that as "control" moves from node v to node v' in the CFG, the probability distribution D transforms to distribution D'.
- Syntactic conditions for correctness (Sect. 5) that in a non-trivial way extend classical work on program slicing [5], and whose key feature is that they involve *two* disjoint slices; in order for the first to be a correct final result of slicing, the other must contain any "observe" nodes sliced away and all nodes on which they depend. We show that the variables of one slice are *probabilistically independent* of the variables of the other, and this leads directly to the correctness of probabilistic slicing (Sect. 6).
- An algorithm, with running time at most cubic in the size of the program, that computes the best possible slice (Sect. 7) in that it is contained in any other (syntactic) slice of the program.

Our approach separates the specification of slicing from algorithms to compute the best possible slice. The former is concerned with defining what is a correct syntactic slice, such that the behavior of the sliced program is equivalent to that of the original. The latter is concerned with how to compute the best possible syntactic slice; this slice is automatically a semantically correct slice —no separate proof is necessary.

A program's behavior is its final probability distribution; we demand equality modulo a constant factor so as to allow the removal of "observe" statements that do not introduce any bias in the final distribution. This will be the case if the variables tested by "observe" statements are independent, in the sense of probability theory, of the variables relevant for the final value.

Full proofs of all results appear in the accompanying technical report [2].

2 Motivating Examples

Probabilistic programs. Whereas in deterministic languages, a variable has only one value at a given time, we consider a language where a variable may have many different values at a given time, each with a certain probability. (Determinism is a special case where one value has probability one, and all others have probability zero.) We assume, to keep our development simple, that each possible value is an integer. A more general development, somewhat orthogonal to the aims of this paper, would allow real numbers and would require us to employ measure theory (as explained in [9]); we conjecture that much will extend naturally (with summations becoming integrals).

Similarly to [6], probabilities are introduced by the construct $x := \texttt{Random}(\psi)$ which assigns to variable x a value with probability given by the random dis-

tribution ψ which in our setting is a mapping from $\mathbb{Z}$ (the set of integers) to $[0,1]$ such that $\sum_{z\in\mathbb{Z}} \psi(z) = 1$. A program phrase transforms a distribution into another distribution, where a distribution assigns a probability to each possible store. This was first formalized by Kozen [8] in a denotational setting. As also in [6], we shall use the construct $\texttt{Observe}(B)$ to "filter out" values which do not satisfy the boolean expression B. That is, the resulting distribution assigns zero probability to all stores not satisfying B, while stores satisfying B keep their probability.

The examples. Slicing amounts to picking a set Q of "program points" (satisfying certain conditions as we shall soon discuss) and then removing nodes not in Q (as we shall formalize in Sect. 4.6). The examples all use a random distribution ψ_4 over $\{0,1,2,3\}$ where $\psi_4(0) = \psi_4(1) = \psi_4(2) = \psi_4(3) = \frac{1}{4}$ whereas $\psi_4(i) = 0$ for $i \notin \{0,1,2,3\}$. The examples all consider whether it is correct to let Q contain exactly $x := \texttt{Random}(\psi_4)$ and $\texttt{Return}(x)$, and thus slice into a program P_x with straightforward semantics: after execution, the probability of each possible store is given by the distribution Δ' defined as $\Delta'(\{x \mapsto i\}) = \frac{1}{4}$ if $i \in \{0,1,2,3\}$; otherwise $\Delta'(\{x \mapsto i\}) = 0$.

Example 1 *Consider the program* $P_1 \stackrel{def}{=}$ *1:* $x := \texttt{Random}(\psi_4)$*; 2:* $y := \texttt{Random}(\psi_4)$*; 3:* $\texttt{Observe}(y \geq 2)$*; 4:* $\texttt{Return}(x)$*. The distribution produced by the first two assignments will assign probability* $\frac{1}{4}\cdot\frac{1}{4} = \frac{1}{16}$ *to each possible store* $\{x \mapsto i,\ y \mapsto j\}$ *with* $i, j \in \{0,1,2,3\}$*. In the final distribution* D_1*, a store* $\{x \mapsto i,\ y \mapsto j\}$ *with* $j < 2$ *is impossible, and for each* $i \in \{0,1,2,3\}$ *there are thus only two possible stores that associate* x *with* i*: the store* $\{x \mapsto i,\ y \mapsto 2\}$*, and the store* $\{x \mapsto i,\ y \mapsto 3\}$*. Restricting to the variable* x *that is ultimately returned,*

$$D_1(\{x \mapsto i\}) = \sum_{j=2}^{3} D_1(\{x \mapsto i,\ y \mapsto j\}) = \frac{1}{16} + \frac{1}{16} = \mathbf{\frac{1}{8}}$$

if $i \in \{0,1,2,3\}$ *(otherwise,* $D_1(\{x \mapsto i\}) = 0$*). We see that the probabilities in* D_1 *do* not *add up to 1 which reflects that the purpose of an* `Observe` *statement is to cause undesired parts of the "local" distribution to "disappear" (which may give certain branches more relative weight than other branches). We also see that* D_1 *equals* Δ' *except for a constant factor:* $D_1 = 0.5 \cdot \Delta'$*. That is,* Δ' *gives the same* relative *distribution over the values of* x *as* D_1 *does. (An alternative way of phrasing this is that "normalizing" the "global" distribution, as done in [6], gives the same result for* P_x *as for* P_1*.) We shall therefore say that* P_x *is a* correct *slice of* P_1*.*

Thus the `Observe` statement is irrelevant to the final relative distribution of x. This is because y and x are *independent* in D_1, as formalized in Def. 3.

Example 2 *Consider the program* $P_2 \stackrel{def}{=}$ *1:* $x := \texttt{Random}(\psi_4)$*; 2:* $y := \texttt{Random}(\psi_4)$*; 3:* $\texttt{Observe}(x + y \geq 5)$*; 4:* $\texttt{Return}(x)$*. Here the final distribution* D_2 *allows only 3 stores:* $\{x \mapsto 2,\ y \mapsto 3\})$*,* $\{x \mapsto 3,\ y \mapsto 2\})$ *and* $\{x \mapsto 3,\ y \mapsto 3\})$*, all with probability* $\frac{1}{16}$*, and hence* $D_2(\{x \mapsto 2\}) = \frac{1}{16}$ *and* $D_2(\{x \mapsto 3\}) = \frac{1}{8}$*. Thus the*

program is biased towards high values of x; in particular we cannot write D_2 in the form $c\Delta'$. Hence it is incorrect *to slice P_2 into P_x.*
In this example, x and y are *not* independent in D_2; this is as expected since the `Observe` statement in P_2 depends on something (the assignment to x) on which the returned variable x also depends.

Example 3 *Consider the program* $P_3 \stackrel{def}{=} x := \texttt{Random}(\psi_4); (\textbf{if } x \geq 2 \;\; z := \texttt{Random}(\psi_4);\; \texttt{Observe}(z \geq 3));\; \texttt{Return}(x)$. *$P_3$ is biased towards returning low values of x, with the final distribution D_3 given by $D_3(\{x \mapsto i\}) = \frac{1}{4}$ when $i \in \{0,1\}$ and $D_3(\{x \mapsto i\}) = D_3(\{x \mapsto i,\; z \mapsto 3\}) = \frac{1}{16}$ when $i \in \{2,3\}$. Hence it is* incorrect *to slice P_3 into P_x.*
The `Observe` statement cannot be removed: it is *control dependent* on the assignment to x, on which the returned x also depends.

The discussion so far suggests the following tentative correctness condition for the set Q picked by slicing:
- Q is "closed under dependency", *i.e.*, if a program point in Q depends on another program point then that program point also belongs to Q;
- Q is part of a "slicing pair": any `Observe` statement that is sliced away belongs to a set Q_0 that is also closed under dependency and is disjoint from Q.

The above condition will be made precise in Def. 9 (Sect. 5) which contains a further requirement, necessary since an `Observe` statement may be encoded as a potentially non-terminating loop, as the next example illustrates.

Example 4 *Consider the program* $P_4 \stackrel{def}{=} x := \texttt{Random}(\psi_4);\; y := \texttt{Random}(\psi_4);\; (\textbf{if } x \geq 2 \;\; (\textbf{while } y \leq 5 \textbf{ do } y := E));\; \texttt{Return}(x)$ *where E is an arithmetic expression. If E is "$y + 1$" then the loop terminates and y's final value is 6. In the resulting distribution D', for $i \in \{0,1,2,3\}$ we have $D'(\{x \mapsto i\}) = D'(\{x \mapsto i,\; y \mapsto 6\}) = \frac{1}{4} = \Delta'(\{x \mapsto i\})$. Thus it is correct to slice P_4 into P_x.*

But if E is "$y - 1$" then the program will not terminate when $x \geq 2$ (and hence the conditional encodes `Observe`$(x < 2)$*). Thus the resulting distribution D_4 is given by $D_4(\{x \mapsto i\}) = \frac{1}{4}$ when $i \in \{0,1\}$ and $D_4(\{x \mapsto i\}) = 0$ when $i \notin \{0,1\}$. Thus it is incorrect to slice P_4 into P_x. Indeed, Def. 9 rules out such a slicing.*

3 Control Flow Graphs

This section precisely defines the kind of CFGs we consider, as well as some key concepts that are mostly standard (see, *e.g.*, [10,3]). However, we also introduce a notion (Def. 1) specific to our approach.

Figure 1 depicts, with the nodes numbered, the CFGs corresponding to the programs P_3 and P_4 from Examples 3 and 4. We see that a node can be labeled with an assignment $x := E$ (x a program variable and E an arithmetic expression), with a random assignment $x := \texttt{Random}(\psi)$ (we shall assume that the probability distribution ψ contains no program variables though it would be straightforward to allow it as in [7]), with `Observe`(B) (B is a boolean expression), or (though not part of these examples) with `Skip`. Also, there are branching nodes with *two* outgoing edges. Finally, there is a unique `End` node

`Return`(x) to which there must be a path from all other nodes but which has no outgoing edges, and a special node `Start` (which may have any label and is numbered 1 in the examples) from which there is a path to all other nodes.

We let $Def(v)$ be the variable occurring on the left hand side if v is a (random) assignment, and let $Use(v)$ be the variables occurring in the right hand side of an assignment, in a boolean expression used in an `Observe` node or in a branching node, or as the sole variable in a `End` node. We demand that all variables be defined before they are used.

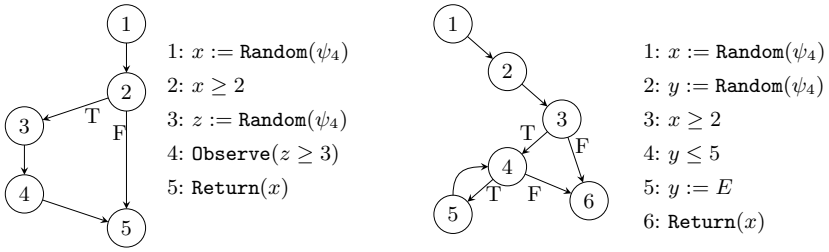

Fig. 1. The CFGs for P_3 (left) and P_4 (right) from Examples 3 and 4.

We say that v_1 ***postdominates*** v if v_1 occurs on all paths from v to `End`; if also $v_1 \neq v$, v_1 is a ***proper postdominator*** of v. And we say that v_1 is the ***first proper postdominator*** of v if whenever v_2 is another proper postdominator of v then all paths from v to v_2 contain v_1. It is easily shown that for any v with $v \neq$ `End`, there is a unique first proper postdominator of v, called $1PPD(v)$. In Fig. 1(right), $1PPD(1) = 2$, while also nodes 3 and 6 are proper postdominators.

We say that v_2 is ***data dependent*** on v_1, written $v_1 \overset{dd}{\rightarrow} v_2$, if there exists $x \in Use(v_2) \cap Def(v_1)$, and there exists a non-trivial path π from v_1 to v_2 such that $x \notin Def(v)$ for all nodes v that are interior in π. In Fig. 1(left), $1 \overset{dd}{\rightarrow} 2$. A set of nodes Q is ***closed under data dependence*** if whenever $v_2 \in Q$ and $v_1 \overset{dd}{\rightarrow} v_2$ then also $v_1 \in Q$. We say that x is ***(Q-)relevant*** in v, written $x \in rv_Q(v)$, if there exists $v' \in Q$ such that $x \in Use(v')$, and a path π from v to v' such that $x \notin Def(v_1)$ for all $v_1 \in \pi \setminus \{v'\}$. In Fig. 1(left), $rv_{\{4,5\}}(4) = \{x, z\}$ but $rv_{\{4,5\}}(3) = \{x\}$.

Next, a concept we have discovered useful for the subsequent development:

Definition 1 With v' a postdominator of v, and Q a set of nodes, we say that v ***stays outside*** Q ***until*** v' iff whenever π is a path from v to v' where v' occurs only at the end, π will contain no node in Q except possibly v'.

In Fig. 1(right), node 4 stays outside $\{1, 6\}$ until 6 but does not stay outside $\{1, 5, 6\}$ until 6. It turns out that if v stays outside Q until v' and Q is closed under data dependence then v has the same Q-relevant variables as v'. Moreover, if Q satisfies certain additional properties, the distribution at v' (of the relevant variables) will equal the distribution at v.

4 Semantics

In this section we shall define the meaning of the CFGs introduced in the previous section, in terms of an operational semantics that manipulates distributions which assign probabilities to stores (Sect. 4.1). Sect. 4.2 defines what it means for sets of variables to be independent wrt. a given distribution. To prepare for the full semantics (Sect. 4.5) we define a one-step reduction (Sect. 4.3) from which we construct a reduction which allows multiple steps but only a bounded number of iterations (Sect. 4.4). The semantics also applies to sliced programs and hence (Sect. 4.6) provides the meaning of slicing.

4.1 Stores and Distributions

Let $\mathcal{U}$ be the universe of variables. A store s is a partial mapping from $\mathcal{U}$ to $\mathbb{Z}$. We write $s[x \mapsto z]$ for the store s' that is like s except $s'(x) = z$, and write $dom(s)$ for the domain of s. We write $\mathcal{S}(R)$ for the set of stores with domain R, and also write $\mathcal{F}$ for $\mathcal{S}(\mathcal{U})$. If $s_1 \in \mathcal{S}(R_1)$ and $s_2 \in \mathcal{S}(R_2)$ with $R_1 \cap R_2 = \emptyset$, we may define $s_1 \oplus s_2$ with domain $R_1 \cup R_2$ the natural way. With R a subset of $\mathcal{U}$, we say that s_1 agrees with s_2 on R, written $s_1 \stackrel{R}{=} s_2$, iff $R \subseteq dom(s_1) \cap dom(s_2)$ and for all $x \in R$, $s_1(x) = s_2(x)$. We assume that there is a function $[\![\ ]\!]$ such that $[\![E]\!]s$ is the integer result of evaluating E in store s and $[\![B]\!]s$ is the boolean result of evaluating B in store s (the free variables of E, B must be in $dom(s)$).

A distribution D (we shall later also use the letter Δ) is a mapping from $\mathcal{F}$ to non-negative reals with $\sum D < \infty$ where $\sum D$ is a shorthand for $\sum_{s \in \mathcal{F}} D(s)$. Thanks to our assumption that values are integers, and since $\mathcal{U}$ can be assumed finite, $\mathcal{F}$ is a countable set and thus $\sum D$ is well-defined even without measure theory. If $\sum D \leq 1$, implying $D(s) \leq 1$ for all s, we say that D is a probability distribution. We define $D_1 + D_2$ by stipulating $(D_1 + D_2)(s) = D_1(s) + D_2(s)$, and for $c \geq 0$ we define cD by stipulating $(cD)(s) = cD(s)$. We write $D_1 \leq D_2$ iff $D_1(s) \leq D_2(s)$ for all s, and say that $D = 0$ iff $D(s) = 0$ for all s. We assume there is a designated initial distribution, $D_{\mathcal{I}}$, such that $\sum D_{\mathcal{I}} = 1$ ($D_{\mathcal{I}}$ may be arbitrary as all variables must be defined before they are used).

As suggested by the calculation in Example 1, we have

Definition 2 For partial store s with domain R, let $D(s) = \sum_{s_0 \in \mathcal{F} \ \mid \ s \stackrel{R}{=} s_0} D(s_0)$.

Observe that $D(\emptyset) = \sum D$. Say that D_1 ***agrees with*** D_2 on R, written $D_1 \stackrel{R}{=} D_2$, if $D_1(s) = D_2(s)$ for all $s \in \mathcal{S}(R)$. If $D_1 \stackrel{R'}{=} D_2$ and $R \subseteq R'$ then $D_1 \stackrel{R}{=} D_2$.

4.2 Probabilistic Independence

Some variables of a distribution D may be independent of others. Formally:

Definition 3 (independence) Let R_1 and R_2 be disjoint sets of variables. We say that R_1 and R_2 are ***independent*** in D iff for all $s_1 \in \mathcal{S}(R_1)$ and $s_2 \in \mathcal{S}(R_2)$, we have $D(s_1 \oplus s_2) \sum D = D(s_1)D(s_2)$.

To motivate the definition, first observe that if $\sum D = 1$ it amounts to the well-known definition of probabilistic independence; next observe that if $\sum D > 0$, it is equivalent to the well-known definition for "normalized" probabilities:

$$\frac{D(s_1 \oplus s_2)}{\sum D} = \frac{D(s_1)}{\sum D} \cdot \frac{D(s_2)}{\sum D}$$

Trivially, R_1 and R_2 are independent in D if $D = 0$ or $R_1 = \emptyset$ or $R_2 = \emptyset$.

Example 5 *In Example 1,* $\{x\}$ *and* $\{y\}$ *are independent in* D_1*. This is since for* $i \in \{0,1,2,3\}$ *and* $j \in \{2,3\}$ *we have* $D_1(\{x \mapsto i,\ y \mapsto j\}) = \frac{1}{16}$ *so that* $D_1(\{x \mapsto i\}) = \frac{1}{8}$*,* $D_1(\{y \mapsto j\}) = \frac{1}{4}$*, and* $\sum D_1 = \frac{1}{2}$*; we thus have the desired equality* $D_1(\{x \mapsto i,\ y \mapsto j\}) \sum D_1 = \frac{1}{32} = D_1(\{x \mapsto i\}) \cdot D_1(\{y \mapsto j\})$*.*

And the equality holds trivially if $i \notin \{0,1,2,3\}$ *or* $j \notin \{2,3\}$ *since then* $D_1(\{x \mapsto i,\ y \mapsto j\}) = 0$ *and either* $D_1(\{x \mapsto i\}) = 0$ *or* $D_1(\{y \mapsto j\}) = 0$*.*

Example 6 *In Example 2,* $\{x\}$ *and* $\{y\}$ *are* not *independent in* D_2*. This is since* $D_2(\{x \mapsto 3,\ y \mapsto 3\}) \sum D_2 = \frac{3}{256}$ *while* $D_2(\{x \mapsto 3\}) D_2(\{y \mapsto 3\}) = \frac{4}{256}$*.*

4.3 One-step Reduction

If v has label $\texttt{Branch}(B)$, with v_1 (v_2) the successor taken when B is true (false), we define

– $(v, D) \xrightarrow{T} (v_1, D_1)$ where $D_1(s)$ equals $D(s)$ when $[\![B]\!]s$ but is 0 otherwise;
– $(v, D) \xrightarrow{F} (v_2, D_2)$ where $D_2(s)$ is 0 when $[\![B]\!]s$ but equals $D(s)$ otherwise.

Thus $D = D_1 + D_2$. Given v such that v has exactly one successor v', the one step reduction $(v, D) \to (v', D')$ is given by defining $D'(s')$ as follows:

`Skip`	$x := E$	$x := \texttt{Random}(\psi)$	$\texttt{Observe}(B)$
$D(s')$	$\sum_{s \in \mathcal{F} \mid s' = s[x \mapsto [\![E]\!]s]} D(s)$	$\sum_{s \in \mathcal{F} \mid s' \overset{\mathcal{U} \setminus \{x\}}{=} s} \psi(s'(x)) D(s)$	$\begin{cases} D(s') \text{ if } [\![B]\!]s' \\ 0 \text{ otherwise} \end{cases}$

If $(v, D) \to (v', D')$ then $\sum D' \leq \sum D$ with equality if v is not an `Observe` node.

4.4 Multi-step Reduction and Loops

The key semantic relation is of the form $(v, D) \Rightarrow (v', D')$ where v' postdominates v, saying that distribution D transforms to distribution D' as "control" moves from v to v' along paths that may contain multiple branches and even loops but which do *not* contain v' until the end. To define that relation, we need an auxiliary relation of the form $(v, D) \overset{k}{\Rightarrow} (v', D')$ where k is a non-negative integer that bounds the number of times control is allowed to move "away" from the `End` node; if $k = 1$ then we only take into account paths that for each step get closer to the `End` node, but if $k = 2$ we also allow paths with one cycle, etc. ($k = 0$ corresponds to "$\bot$" in denotational semantics.) Our goal is to let D'

be a function of v, D, v' and k, and we do so by a definition that is inductive first in k, and next on the length of the longest acyclic path from v to v' (proving properties of the relation will involve a case analysis on the various clauses).

Definition 4 ($\stackrel{k}{\Rightarrow}$) Given v and v' where v' postdominates v, and given k and D, $(v, D) \stackrel{k}{\Rightarrow} (v', D')$ holds when D' is defined as follows:

1. if $k = 0$ then $D' = 0$;
2. otherwise, if $v' = v$ then $D' = D$;
3. otherwise, if with $v'' = 1PPD(v)$ we have $v' \neq v''$, we recursively first find D'' with $(v, D) \stackrel{k}{\Rightarrow} (v'', D'')$ and next find D' with $(v'', D'') \stackrel{k}{\Rightarrow} (v', D')$;
4. otherwise, if v has exactly one successor, which must be v', we let D' be such that $(v, D) \rightarrow (v', D')$;
5. otherwise, if v has two successors with $(v, D) \stackrel{T}{\rightarrow} (v_1, D_1)$ and $(v, D) \stackrel{F}{\rightarrow} (v_2, D_2)$ (thus v' postdominates v_1 and v_2), we recursively find D'_1 and D'_2 such that $(v_1, D_1) \stackrel{k_1}{\Rightarrow} (v', D'_1)$ and $(v_2, D_2) \stackrel{k_2}{\Rightarrow} (v', D'_2)$, and let $D' = D'_1 + D'_2$. Here k_i $(i = 1, 2)$ is given as follows: if the longest acyclic path from v_i to v' is shorter than the length of the longest acyclic path from v to v' then $k_i = k$, otherwise $k_i = k - 1$.

Example 7 *Consider the CFG for P_4 (Fig. 1(right)) with E chosen as "$y + 1$" and with D_k such that $(1, D_{\mathcal{I}}) \stackrel{k}{\Rightarrow} (6, D_k)$. If $i \in \{0, 1\}$ and $j \in \{0, 1, 2, 3\}$ then for all $k \geq 1$ we have $D_k(\{x \mapsto i,\ y \mapsto j\}) = \frac{1}{16}$ and thus $D_k(\{x \mapsto i\}) = \frac{1}{4}$. But if $i \in \{2, 3\}$ then we have $D_k(\{x \mapsto i,\ y \mapsto 6\}) = 0$ if $k \leq 3$; $D_4(\{x \mapsto i,\ y \mapsto 6\}) = \frac{1}{16}$ (for y initially 3); $D_5(\{x \mapsto i,\ y \mapsto 6\}) = \frac{2}{16}$; $D_6(\{x \mapsto i,\ y \mapsto 6\}) = \frac{3}{16}$; and $D_k(\{x \mapsto i,\ y \mapsto 6\}) = \frac{4}{16}$ if $k \geq 7$.*

Note also that for $k > 0$, $(4, D) \stackrel{k}{\Rightarrow} (4, D')$ only holds if $D' = D$, since the only path from 4 to 4 where 4 does not occur until the end is the empty path. Still, the cycle between nodes 4 and 5 is taken into account. For if $(4, D) \stackrel{k}{\Rightarrow} (6, D')$ then $D' = D'_1 + D_2$ where $D = D_1 + D_2$ (D_2 is D restricted to states where $y > 5$) and D'_1 is such that for some D''_1, $(5, D_1) \stackrel{k-1}{\Rightarrow} (4, D''_1)$ and $(4, D''_1) \stackrel{k-1}{\Rightarrow} (6, D'_1)$.

D' is a monotone function of D and of k:

Lemma 1 (monotonicity) *If $(v, D_1) \stackrel{k_1}{\Rightarrow} (v', D'_1)$ and $(v, D_2) \stackrel{k_2}{\Rightarrow} (v', D'_2)$ with $k_1 \leq k_2$ and $D_1 \leq D_2$ then $D'_1 \leq D'_2$. If $(v, D) \stackrel{k}{\Rightarrow} (v', D')$ then $\sum D' \leq \sum D$.*

Equality between $\sum D'$ and $\sum D$ can fail due to `Observe` nodes (*cf.* Examples 1–3), or due to infinite loops (*cf.* Example 4) which cause k to be eventually zero.

4.5 Top-Level Semantics

Definition 5 ($\Rightarrow$) Given (v, D), and v' which postdominates v, $(v, D) \Rightarrow (v', D')$ holds when D' is defined as follows: with D_k (for $k \geq 0$) the unique distribution such that $(v, D) \stackrel{k}{\Rightarrow} (v', D_k)$, let $D' = \lim_{k \to \infty} D_k$ (the limit is taken pointwise).

Observe by Lemma 1 that $D_{k_1} \leq D_{k_2}$ when $k_1 \leq k_2$; for each s we thus have $D_0(s) \leq D_1(s) \leq \ldots \leq D_k(s) \leq D_{k+1}(s) \leq \ldots$ and hence it is well-defined to let $D'(s) = \lim_{k \to \infty} D_k(s)$. Also at top-level, monotonicity holds:

Lemma 2 *If* $(v, D) \Rightarrow (v', D')$ *then* $\sum D' \leq \sum D$.
Example 8 *Continuing Example 7, we see that the limit* D' *is given as follows:* $D'(\{x \mapsto i,\ y \mapsto j\}) = \frac{1}{16}$ *if* $i \in \{0,1\}$, $j \in \{0,1,2,3\}$; $D'(\{x \mapsto i,\ y \mapsto j\}) = \frac{1}{4}$ *if* $i \in \{2,3\}, j = 6$; *and* $D'(\{x \mapsto i,\ y \mapsto j\}) = 0$ *otherwise.*

We may define the meaning of a CFG with `End` node `Return`(x) as $\lambda v.D'(\{x \mapsto v\})$ where D' is such that $(\texttt{Start}, D_{\mathcal{I}}) \Rightarrow (\texttt{Return}(x), D')$.

4.6 Semantics of Slicing

A *slice set* is a set of nodes (which must satisfy certain conditions, *cf.* Def. 9). Slicing amounts to ignoring nodes not in the slice set:
Definition 6 ($\Longrightarrow$) Given a slice set Q, we define the semantics of the CFG that results from slicing wrt. Q as follows:

Let $(v, \Delta) \overset{k}{\Longrightarrow} (v', \Delta')$ be defined as $(v, D) \overset{k}{\Rightarrow} (v', D')$ (Def. 4), except that whenever $v \notin Q \cup \{\texttt{End}\}$ then v is labeled `Skip` and the successor of v becomes $1PPD(v)$. And let $(v, \Delta) \Longrightarrow (v', \Delta')$ be defined by letting $\Delta' = \lim_{k \to \infty} \Delta_k$ where for each $k \geq 0$, Δ_k is the unique distribution such that $(v, \Delta) \overset{k}{\Longrightarrow} (v', \Delta_k)$.

5 Conditions for Slicing

With Q the slice set, we now develop conditions for Q that ensure semantic correctness. It is standard to require Q to be closed under data dependence, and additionally also under some kind of "control dependence". In this section we first elaborate on the latter condition after which we study the extra conditions needed in our probabilistic setting. Eventually, Def. 9 gives conditions that involve not only Q but also another slice set Q_0 containing all `Observe` nodes to be sliced away. As stated in Proposition 1, these conditions are sufficient to establish probabilistic independence of Q and Q_0. This in turn is crucial for establishing the correctness of slicing, as stated in Theorem 1 (Sect. 6).

Weak Slice Sets. Danicic *et al.* [5] show that various kinds of control dependence can all be elegantly expressed within a general framework whose core is the following notion:
Definition 7 (next observable) With Q a set of nodes, v' is a ***next observable*** in Q of v iff $v' \in Q \cup \{\texttt{End}\}$, and v' occurs on all paths from v to a node in $Q \cup \{\texttt{End}\}$.
A node v can have at most one next observable in Q. It thus makes sense to write $v' = next_Q(v)$ if v' is a next observable in Q of v. We say that Q ***provides next observables*** iff $next_Q(v)$ exists for all nodes v. If $v' = next_Q(v)$ then v' is a postdominator of v, and if $v \in Q \cup \{\texttt{End}\}$ then $next_Q(v) = v$.

In the CFG for P_3 (Fig 1), letting $Q = \{1, 3, 5\}$, node 5 is a next observable in Q of 4: all paths from 4 to a node in Q will contain 5. But no node is a next observable in Q of 2: node 3 is not since there is a path from 2 to 5 not containing 3, and node 5 is not since there is a path from 2 to 3 not containing

5. Therefore Q cannot be the slice set: node 1 can have only one successor in the sliced program but we have no reason to choose either of the nodes 3 and 5 over the other as that successor. This motivates the following definition:

Definition 8 (weak slice set) We say that Q is a ***weak slice set*** iff it provides next observables, and is closed under data dependence.

While the importance of "provides next observable" was recognized already in [11,1], Danicic *et al.* were the first to realize that it is *the* key property (together with data dependence) to ensure semantically correct slicing. They call the property "weakly committing" (thus our use of "weak") and our definition differs slightly from theirs in that we always consider `End` an "observable".

It is easy to see that the empty set, as well as the set of all nodes, is a weak slice set. Moreover, if Q_1, Q_2 are weak slice sets, so is $Q_1 \cup Q_2$.

Adapting to the Probabilistic Setting. The key challenge in slicing probabilistic programs is, as already motivated through Examples 1–4, to handle `Observe` nodes. In Sect. 2 we hinted at some tentative conditions a slice set Q must satisfy; we can now phrase them more precisely:

1. Q must be a weak slice set that contains `End`, and
2. there exists another weak slice set Q_0 such that (a) Q and Q_0 are disjoint and (b) all `Observe` nodes belong to either Q or Q_0.

For programs P_1, P_2, the control flow is linear and hence all nodes have a next observable (so a node set that is closed under data dependence is a weak slice set). For P_1 we may choose $Q = \{1, 4\}$ and $Q_0 = \{2, 3\}$ as they are disjoint, and both closed under data dependence. Hence we may use $\{1, 4\}$ as a slice set; from Def. 6 we see that the resulting slice is 1: $x :=$ `Random`(ψ_4); 2: `Skip`; 3: `Skip`; 4: `Return`(x), which is obviously equivalent to P_x as defined in Sect. 2.

Next consider the program P_2 where Q should contain 4 and hence (by data dependence) also contain 1. Now assume, in order to remove the `Observe` node (and produce P_x), that Q does not contain 3. Then Q_0 must contain 3, and (as Q_0 is closed under data dependence) also 1. But then Q and Q_0 are not disjoint, which contradicts our requirements. Thus Q does contain 3, and hence also 2. That is, $Q = \{1, 2, 3, 4\}$. We see that the only possible slicing is the trivial one.

Any slice for P_3 will also be trivial. From $5 \in Q$ we infer (by data dependence) that $1 \in Q$. Assume, to get a contradiction, that $4 \notin Q$. Then $4 \in Q_0$, and for node 2 to have a next observable in Q_0 we must also have $2 \in Q_0$ which by data dependence implies $1 \in Q_0$ which as $1 \in Q$ contradicts Q and Q_0 being disjoint. Thus $4 \in Q$ which implies $3 \in Q$ (by data dependence) and $2 \in Q$ (as otherwise 2 has no next observable in Q).

For P_4, it is plausible that $Q = \{1, 6\}$ and $Q_0 = \emptyset$, since for all $v \neq 1$ we would then have $6 = next_Q(v)$. From Def. 6 we see that after removing unreachable nodes, the resulting slice is: 1: $x :=$ `Random`(ψ_4); 2: `Skip`; 3: `Skip`; 6: `Return`(x). Yet, in Example 4 we saw that this is in general *not* a correct slice of P_4. This reveals a problem with our tentative correctness conditions; they do not take into account that `Observe` nodes may be "encoded" as infinite loops. To repair that, we demand that just like all `Observe` nodes must belong to either Q or Q_0, also all cycles must touch either Q or Q_0. With this requirement, it is no longer

possible that Q contains only nodes 1 and 6. For if so, then Q_0 would have to contain node 4 or node 5 since these two nodes form a cycle. But then, in order for node 3 to have a next observable in Q_0, it must be the case that Q_0 contains node 3, and hence (by data dependence) also node 1 which contradicts Q and Q_0 being disjoint. Thus we have motivated the following definition.

Definition 9 (slicing pair) Let Q, Q_0 be sets of nodes. (Q, Q_0) is a ***slicing pair*** iff (a) Q, Q_0 are both weak slice sets with $\texttt{End} \in Q$; (b) Q, Q_0 are disjoint; (c) all `Observe` nodes are in $Q \cup Q_0$; and (d) all cycles contain at least one element of $Q \cup Q_0$.

If (Q, Q_0) is a slicing pair then $rv_Q(v) \cap rv_{Q_0}(v) = \emptyset$ for all nodes v. For, if $x \in rv_Q(v) \cap rv_{Q_0}(v)$ then a definition of x, known to exist, would be in $Q \cap Q_0$ which is empty. Moreover, the Q-relevant variables are probabilistically independent (as defined in Def. 3) of the Q_0-relevant variables, as stated by the following preservation result which is one of the main contributions of this paper in that it gives a syntactic condition for probabilistic independence:

Proposition 1 (Independence) *Let (Q, Q_0) be a slicing pair. Assume that $(v, D) \overset{k}{\Rightarrow} (v', D')$ where v' postdominates v. If $rv_Q(v)$ and $rv_{Q_0}(v)$ are independent in D then $rv_Q(v')$ and $rv_{Q_0}(v')$ are independent in D'.*

6 Slicing and its Correctness

We can now precisely phrase the desired correctness result:

Theorem 1 *Given a CFG with* `End` *of the form* `Return`(x)*, and let (Q, Q_0) be a slicing pair. Let D' and Δ' be the unique distributions such that $(\texttt{Start}, D_{\mathcal{I}}) \Rightarrow (\texttt{End}, D')$ and $(\texttt{Start}, D_{\mathcal{I}}) \Longrightarrow (\texttt{End}, \Delta')$ where the latter denotes slicing wrt. Q (cf. Sect. 4.6). Then there exists a real number c with $0 \leq c \leq 1$ such that for all v, $D'(\{x \mapsto v\}) = c\Delta'(\{x \mapsto v\})$.*

This follows from a more general proposition (allowing an inductive proof) stated below, with $v = \texttt{Start}$ so that (from the requirement that a variable must be defined before it is used) R and R_0 are both empty (and thus independent), with $v' = \texttt{End} = \texttt{Return}(x)$ so that $R' = \{x\}$, and with $\Delta = D = D_{\mathcal{I}}$.

Proposition 2 *Let (Q, Q_0) be a slicing pair. Let v' postdominate v, with $R = rv_Q(v)$, $R' = rv_Q(v')$, and $R_0 = rv_{Q_0}(v)$. Assume that D is such that R and R_0 are independent in D, and that Δ is such that $D \overset{R}{=} \Delta$. Let D' and Δ' be the unique distributions such that $(v, D) \Rightarrow (v', D')$ and $(v, \Delta) \Longrightarrow (v', \Delta')$. Then there exists a real number c with $0 \leq c \leq 1$ such that $D' \overset{R'}{=} c\Delta'$.*

Moreover, $c = 1$ if v stays outside Q_0 until v' (since then the conditions for a slicing pair guarantee that no observe nodes, or infinite loops, are sliced away).

That is, for the relevant variables, the final distribution is the same in the sliced program as in the original program, except for a constant factor c such that $\sum D' = c \sum \Delta'$.

To prove Proposition 2, we need a similar result that involves k and where the proof of sequential composition (case 3 in Definition 4) employs Proposition 1 to ensure that the independence property still holds after the first reduction, so that we can apply the induction hypothesis to the second reduction.

7 Computing the (Least) Slice

There always exists at least one slicing pair, with Q the set of all nodes and with Q_0 the empty set; in that case, the sliced program is the same as the original. Our goal, however, is to find a slicing pair (Q, Q_0) where Q is as small as possible while including the `End` node. This section describes an algorithm `BSP` for doing so. The running time of our algorithms is measured in terms of n, the number of nodes in the graph. Note that the number of edges is at most $2n$ and thus in $O(n)$. Our algorithms use a boolean table $\texttt{DD}^*$ such that $\texttt{DD}^*(v, v')$ is true iff $v \stackrel{dd}{\rightarrow}^* v'$ where $\stackrel{dd}{\rightarrow}^*$ is the reflexive and transitive closure of $\stackrel{dd}{\rightarrow}$. Given $\texttt{DD}^*$, it is easy to ensure that sets are closed under data dependence, and we shall do that in an incremental way: there exists an algorithm $\texttt{DD}^{\text{close}}$ which given a node set Q_0 that is closed under data dependence, and a node set Q_1, returns the least set Q containing Q_0 and Q_1 that is closed under data dependence.

Computing the Least Weak Slice. In Fig. 2(upper right) we define a function `LWS` which constructs the least weak slice set that contains a given set; it works by successively adding nodes to the set until it is closed under data dependence, and provides next observables.

To check the latter, we employ a function `PN`? as defined in Fig. 2(left): given Q, it does a backward breadth-first search from $Q \cup \{\texttt{End}\}$ to find the first node (if any) from which two nodes in that set are reachable without going through Q; such a node must be included in any superset providing next observables.

Lemma 3 *Given Q, the function* `PN`? *runs in time $O(n)$ and returns C such that (a) if C is empty then Q provides next observables, and (b) if C is non-empty then $C \cap Q = \emptyset$ and all supersets of Q that provide next observables contain C.*

Lemma 4 *Given Q_0, the function* `LWS` *returns Q such that Q is a weak slice set and $Q_0 \subseteq Q$. Moreover, if Q' is a weak slice set with $Q_0 \subseteq Q'$, then $Q \subseteq Q'$. Finally, given* $\texttt{DD}^*$, `LWS` *runs in time $O(n^2)$.*

Computing the Best Slicing Pair. We now develop an algorithm `BSP` which given a CFG returns a slicing pair (Q, Q_0) with Q as small as possible. From Def. 9 we know that each `Observe` node has to be put either in Q or in Q_0, and also that at least one node from each cycle has to be put either in Q or in Q_0. Especially the latter requirement may suggest that our algorithm will have to explore a huge search space, but fortunately it is sufficient to consider only the node(s) with minimal *height*. Here node v's height, denoted $H(v)$, is the length of the shortest path(s) from v to `End`. This motivates

Definition 10 A node v is ***essential*** if either (a) v is an `Observe` node, or (b) v belongs to a cycle π where $H(v) \leq H(v_1)$ for all $v_1 \in \pi$.

For disjoint weak slice sets (Q, Q_0) with $\texttt{End} \in Q$ to be a slicing pair, it is sufficient and necessary that each essential node be placed either in Q or in Q_0.

Lemma 5 (Sufficient) *Let Q and Q_0 be disjoint weak slice sets with* $\texttt{End} \in Q$*, and assume that all essential nodes are in $Q \cup Q_0$. Then (Q, Q_0) is a slicing pair.*

Lemma 6 (Necessary) *Let (Q, Q_0) be a slicing pair. If v is essential then $v \in Q \cup Q_0$.*

```
PN?(Q)
    F ← Q ∪ {End};
    foreach v ∈ V
        if v ∈ F
            N[v] ← v
        else
            N[v] ← ⊥;
    C ← ∅;
    while F ≠ ∅ ∧ C = ∅
        F' ← ∅;
        foreach edge from v ∉ Q
                        to v' ∈ F
            if N(v) = ⊥
                N(v) ← N(v');
                F' ← F' ∪ {v}
            else if N(v) ≠ N(v')
                C ← C ∪ {v};
        F ← F'
    return C
```

```
LWS(Q0)
    Q ← DD^close(∅, Q0); C ← PN?(Q);
    while C ≠ ∅
        Q ← DD^close(Q, C);
        C ← PN?(Q)
    return Q

BSP()
    W ← the set of essential nodes;
    foreach v ∈ W ∪ {End}
        Q_v ← LWS({v});
    Q ← ∅; F ← Q_End;
    while F ≠ ∅
        Q ← Q ∪ F; F ← ∅;
        foreach v ∈ W
            if Q_v ∩ Q ≠ ∅
                W ← W \ {v}; F ← F ∪ Q_v;
    Q0 ← ⋃_{v∈W} Q_v;
    return (Q, Q0)
```

Fig. 2. Algorithms for: checking if a set provides next observables (PN?); finding the least weak slice set containing a given set (LWS); finding the best slicing pair (BSP).

Fig. 2(lower right) presents the algorithm BSP that computes the best slicing pair (Q, Q_0). The idea is to built Q incrementally, initially containing only End, and then add v whenever v is essential but cannot be placed in Q_0 as then Q and Q_0 would overlap. That BSP produces the *best slicing pair* is captured by

Proposition 3 *The algorithm* BSP *returns (on a given CFG) Q and Q_0 such that (Q, Q_0) is a slicing pair, and if (Q', Q'_0) is a slicing pair then $Q \subseteq Q'$.*

After analyzing the running times, we get:

Theorem 2 *For a given CFG, there is an algorithm that in time $O(n^3)$ computes a slicing pair (Q, Q_0) where $Q \subseteq Q'$ for any other slicing pair (Q', Q'_0).*

Also the algorithm given in [5] for computing (their version of) weak slices runs in cubic time. We do not expect that there exists an algorithm wih lower asymptotic complexity, since we need to compute data dependencies which is known to involve computing a transitive closure.

Illustrating the algorithms. First consider the program P_1 from Example 1 where the non-trivial true entries of $\mathtt{DD}^*$ are $(1, 4)$ (since $1 \stackrel{dd}{\rightarrow} 4$) and $(2, 3)$, and where 3 is the only essential node. BSP thus computes LWS($\{4\}$) and LWS($\{3\}$). When running LWS on $\{4\}$, initially $Q = \{1, 4\}$ which is also the final value of Q since PN?($\{1, 4\}$) returns $\emptyset$ (after a sequence of iterations where F is first $\{1, 4\}$ and next $\{3\}$ and next $\{2\}$ and finally $\emptyset$). Thus $Q_4 = \{1, 4\}$ and similarly we get $Q_3 = \{2, 3\}$. When the members of $W = \{3\}$ are first examined in the BSP algorithm, we have $Q = Q_4$ and thus $Q_3 \cap Q = \emptyset$. Hence the while loop terminates with $Q = \{1, 4\}$ and subsequently we get $Q_0 = Q_3 = \{2, 3\}$.

Next consider the CFG for P_4 (Fig. 1) with E containing only y free. Here $\stackrel{dd}{\rightarrow}$ is given as follows: $1 \stackrel{dd}{\rightarrow} 3$, $1 \stackrel{dd}{\rightarrow} 6$, $2 \stackrel{dd}{\rightarrow} 4$, $2 \stackrel{dd}{\rightarrow} 5$, $5 \stackrel{dd}{\rightarrow} 4$, and $5 \stackrel{dd}{\rightarrow} 5$. Since $H(4) = 1$ and $H(5) = 2$, node 4 is the only essential node. `BSP` thus has to compute `LWS`($\{6\}$) and `LWS`($\{4\}$):

- `LWS`($\{6\}$): initially, $Q = \{1, 6\}$ which is also the final value of Q since `PN`?($\{1, 6\}$) returns $\emptyset$ (after a sequence of iterations where F is first $\{1, 6\}$ and next $\{3, 4\}$ and next $\{2, 5\}$ and finally $\emptyset$). Thus $Q_6 = \{1, 6\}$.
- `LWS`($\{4\}$): initially, $Q = \{2, 4, 5\}$. In `PN`? we initially thus have $F = \{2, 4, 5, 6\}$ which causes the first iteration of the while loop to put 3 in C so that $\{3\}$ is eventually returned. Since $1 \stackrel{dd}{\rightarrow} 3$ holds, the next iteration of `LWS` will have $Q = \{1, 2, 3, 4, 5\}$ on which `PN`? will return $\emptyset$. Thus $Q_4 = \{1, 2, 3, 4, 5\}$.

When the members of $W = \{4\}$ are first examined in the `BSP` algorithm, we have $Q = Q_6$ and thus $Q_4 \cap Q = \{1\} \neq \emptyset$. Hence Q will equal $Q_6 \cup Q_4 = \{1, 2, 3, 4, 5, 6\}$ and as W is now empty, the loop will terminate and we get $Q_0 = \emptyset$.

8 Extensions and Future Work

Sect. 7 presented an algorithm for computing the least syntactic slice. But there may exist even smaller slices that are still semantically correct: recall Example 4 where the only correct syntactic slice is the program itself (as shown in Sect. 7) but where a much smaller slice may be semantically correct for certain instantiations of the generic "E". Similarly, a node labeled `Observe`(B) can be safely discarded if B always evaluates to *true*. For example, the CFG with textual representation $1 : x := \texttt{Random}(\psi_4)$; $2 : y := 7$; $3 :$ **if** $x \geq 2$ ($4 : \texttt{Observe}(y = 7)$); $5 : \texttt{Return}(x)$ is semantically equivalent to the CFG containing only nodes 1 and 5. But the former has no smaller syntactic slice, since if (Q, Q_0) is a slicing pair with $5 \in Q$ (and thus $1 \in Q$) then $Q = \{1, 2, 3, 4, 5\}$ as we now show. If $4 \in Q_0$ then $3 \in Q_0$ (as Q_0 provides next observables) and thus $1 \in Q_0$ which contradicts $Q \cap Q_0 = \emptyset$. Hence (as 4 must belong to $Q \cup Q_0$) $4 \in Q$; but then $2 \in Q$ (by data dependence) and $3 \in Q$ (as Q provides next observables).

Simple analyses like constant propagation may improve the precision of slicing even in a deterministic setting, but the probabilistic setting gives an extra opportunity: after an `Observe`(B) node, we know that B holds. As richly exploited in [7], a simple syntactic transformation often suffices to get the benefits of that information, as we illustrate on the program from [7, Fig. 4] whose CFG (in slightly modified form) is depicted in Fig. 3. In our setting, if (Q, Q_0) with $18 \in Q$ is the best slicing pair, then Q will contain everything except nodes 12, 13, 14, as can be seen as follows: $16, 17 \in Q$ by data dependence; $15 \in Q$ as Q provides next observables; $6, 7, 8, 9 \in Q$ by data dependence; $3, 4, 5 \in Q$ as Q provides next observables; $1, 2 \in Q$ by data dependence; also $10 \in Q$ as otherwise $10 \in Q_0$ and thus also $9 \in Q_0$ which contradicts $Q \cap Q_0 = \emptyset$.

Alternatively, suppose we insert a node 11 labeled $g := 0$ between nodes 10 and 12. This clearly preserves the semantics, but allows a much smaller slice: choose $Q = \{11, 15, 16, 17, 18\}$ and $Q_0 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. This is much like what is arrived at (through a more complex process) in [7, Fig. 15].

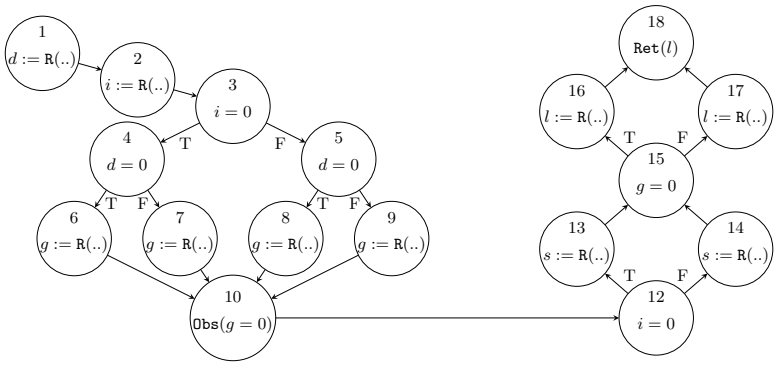

Fig. 3. The program from [7, Fig. 4] (modified).

Future work involves exploring a larger range of examples, and investigating useful techniques for computing slices that are smaller than the least syntactic slice yet semantically correct. (Of course it is in general undecidable to compute the least semantically correct slice.) We have recently augmented our theory such that we can ignore loops that are known (by some means) to always terminate. That is, for a slicing pair (Q, Q_0), a cycle which cannot go on forever (or does it with probability zero) does not need to contain a node from $Q \cup Q_0$.

9 Conclusion and Related Work

We have developed a theory for the slicing of probabilistic imperative programs. We have used and extended techniques from the literature [10,3,11,1] on the slicing of deterministic imperative programs. These frameworks, some of which have been partly verified by mechanical proof assistants [13,4], were recently coalesced by Danicic *et al.* [5] who provided solid semantic foundations to the slicing of a large class of deterministic programs. Our extension of that work is non-trivial in that we need to capture probabilistic independence as done in Proposition 1 which requires *two* slice sets rather than just one.

We were directly inspired by Hur *et al.* [7] who point out the challenges involved in the slicing of probabilistic programs, and present an algorithm which constructs a semantically correct slice. The paper does not state whether it is in some sense the least possible slice; neither does it address the complexity of the algorithm. While Hur *et al.*'s approach differs from ours, for example it is based on a denotational semantics for a structured language (we expect the two semantics to coincide for CFGs of structured programs), it is not surprising that their correctness proof also has probabilistic independence (termed "decomposition") as a key notion. Our theory separates specification and implementation which we believe provides for a cleaner approach. But as mentioned in Sect. 8, they incorporate powerful optimizations that we do not (yet) allow.

Acknowledgements. Thanks to Gordon Stewart for comments on earlier drafts.

References

1. T. Amtoft. Slicing for modern program structures: A theory for eliminating irrelevant loops. *Information Processing Letters*, 106(2):45–51, Apr. 2008.
2. T. Amtoft and A. Banerjee. A theory of slicing for probabilistic control flow graphs. Technical Report CIS TR 2015-1, Kansas State University, July 2015. http://people.cis.ksu.edu/∼tamtoft/Papers/Amt+Ban:ProbSlice-2015/long.pdf.
3. T. Ball and S. Horwitz. Slicing programs with arbitrary control flow. In P. Fritzon, editor, *Proceedings of the First International Workshop on Automated and Algorithmic Debugging (AADEBUG '93)*, volume 749 of *LNCS*, pages 206–222, London, UK, 1993. Springer-Verlag.
4. S. Blazy, A. Maroneze, and D. Pichardie. Verified validation of program slicing. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, pages 109–117, New York, NY, USA, 2015. ACM.
5. S. Danicic, R. W. Barraclough, M. Harman, J. D. Howroyd, Á. Kiss, and M. R. Laurence. A unifying theory of control dependence and its application to arbitrary program structures. *Theoretical Computer Science*, 412(49):6809–6842, Nov. 2011.
6. A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In M. B. Dwyer and J. Herbsleb, editors, *ICSE, Future of Software Engineering track*, FOSE 2014, pages 167–181, New York, NY, USA, 2014. ACM.
7. C.-K. Hur, A. V. Nori, S. K. Rajamani, and S. Samuel. Slicing probabilistic programs. In K. Pingali, editor, *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 133–144, New York, NY, USA, 2014. ACM.
8. D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22:328–350, 1981.
9. P. Panangaden. *Labelled Markov Processes*. Imperial College Press, 2009.
10. A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, Sept. 1990.
11. V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, and M. B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 29(5), Aug. 2007. A special issue with extended versions of selected papers from the 14th European Symposium on Programming (ESOP'05).
12. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.
13. D. Wasserrab. *From Formal Semantics to Verified Slicing*. PhD thesis, Karlsruher Institut für Technologie, 2010.
14. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.