Slicing Probabilistic Programs

Chung-Kil Hur *
Seoul National University
gil.hur@cse.snu.ac.kr

Aditya V. Nori
Microsoft Research
adityan@microsoft.com

Sriram K. Rajamani
Microsoft Research
sriram@microsoft.com

Selva Samuel
Microsoft Research
t-ssamue@microsoft.com

Abstract

Probabilistic programs use familiar notation of programming languages to specify probabilistic models. Suppose we are interested in estimating the distribution of the return expression r of a probabilistic program P. We are interested in *slicing* the probabilistic program P and obtaining a simpler program $\text{SLI}(P)$ which retains only those parts of P that are relevant to estimating r, and elides those parts of P that are not relevant to estimating r. We desire that the SLI transformation be both correct and efficient. By correct, we mean that P and $\text{SLI}(P)$ have identical estimates on r. By efficient, we mean that estimation over $\text{SLI}(P)$ be as fast as possible.

We show that the usual notion of program slicing, which traverses control and data dependencies backward from the return expression r, is unsatisfactory for probabilistic programs, since it produces incorrect slices on some programs and sub-optimal ones on others. Our key insight is that in addition to the usual notions of control dependence and data dependence that are used to slice non-probabilistic programs, a new kind of dependence called *observe dependence* arises naturally due to observe statements in probabilistic programs.

We propose a new definition of $\text{SLI}(P)$ which is both correct and efficient for probabilistic programs, by including observe dependence in addition to control and data dependences for computing slices. We prove correctness mathematically, and we demonstrate efficiency empirically. We show that by applying the SLI transformation as a pre-pass, we can improve the efficiency of probabilistic inference, not only in our own inference tool R2, but also in other systems for performing inference such as Church and Infer.NET.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords Probabilistic Programming, Program Slicing, Bayesian Reasoning

* Supported by the Engineering Research Center of Excellence Program of Korea Ministry of Science, ICT & Future Planning (MSIP) / National Research Foundation of Korea (NRF) (Grant NRF-2008-0062609).

PLDI '14, June 9–11, 2014, Edinburgh, United Kingdom.

ACM 978-1-4503-2784-8/14/06…$15.00.
http://dx.doi.org/10.1145/2594291.2594303

1. Introduction

Probabilistic programs are "usual" programs (written in languages like C or Java or LISP or ML) with two added constructs: (1) the ability to draw values at random from distributions, and (2) the ability to condition values of variables in a program through observe statements (which allow data from real world observations to be incorporated into a probabilistic program). A variety of probabilistic programming languages and systems have been proposed [2, 10–12, 18, 20, 23, 26]. However, unlike "usual" programs which are written for the purpose of being executed, the purpose of a probabilistic program is to implicitly specify a probability distribution. Probabilistic programs can be used to represent *probabilistic graphical models* [19], which use graphs to denote conditional dependences between random variables. Probabilistic graphical models are widely used in statistics and machine learning, with diverse application areas including information extraction, speech recognition, computer vision, coding theory, biology and reliability analysis.

Probabilistic inference is the problem of computing an explicit representation of the probability distribution implicitly specified by a probabilistic program: depending on the application, we may want to compute the expected value of some function f with respect to the distribution, or the mode of the distribution, or simply a set of samples drawn from the distribution.

Program slicing, as originally defined by Weiser [29], computes a subset of statements in program P that can influence a variable v at a given program point. Slicing is well understood for non-probabilistic programs (*i.e.,* deterministic or non-deterministic programs without probability), and has been used for various purposes such as model checking [13], program understanding [16] and debugging [29].

In this paper, we investigate slicing for probabilistic programs. Specifically, we define a SLI transformation that constructs a slice of a probabilistic program with respect to a fixed set of output variables, and a set of observed variables, which is both correct and efficient.

Usual notions of program dependences do not produce correct slices for probabilistic programs (we illustrate this using examples in Section 2). Our key insight is that in addition to the usual notions of control dependence and data dependence that are used to slice non-probabilistic programs, a new kind of dependence called *observe dependence* arises naturally due to observe statements in probabilistic programs. Observe dependence is a new kind of dependence that has not been studied before in the program slicing literature, even though observe statements are equivalent to non-terminating while loops (which are present in non-probabilistic programs), and variants of program slicing have been proposed to preserve terminating and non-terminating behaviors of non-probabilistic programs before (see for instance, [13]).

Although observe statements can be encoded as non-terminating while loops, a key difference is that the semantics of probabilistic

```
1: bool c1, c2;
2: int count = 0;
3: c1 = Bernoulli(0.5);
4: if (c1) then
5:   count = count + 1;
6: c2 = Bernoulli(0.5);
7: if (c2) then
8:   count = count + 1;
9: return(count);
```

Example 1.

```
1: bool c1, c2;
2: int count = 0;
3: c1 = Bernoulli(0.5);
4: if (c1) then
5:   count = count + 1;
6: c2 = Bernoulli(0.5);
7: if (c2) then
8:   count = count + 1;
9: observe(c1 || c2);
10:return(count);
```

Example 2.

Figure 1. Two probabilistic programs.

programs is concerned only with the (normalized) probability distribution of outputs over terminating runs (see Section 3 for a formal definition of semantics of probabilistic programs). The semantics does *not* require preservation of non-terminating runs as long as the normalized probability distribution of outputs over terminating runs is preserved during slicing (in order that slicing be semantics preserving). As a result, a more aggressive notion of observe dependence, which does not preserve non-termination but propagates dependence from observed variables (in a specific manner defined more precisely later) turns out to be sufficient. We define the SLI transformation using this new notion of observe dependence combined with the usual notion of control and data dependences. We establish correctness by expressing the SLI transformation as a composition of four sub-transformations and proving that each of these sub-transformations is semantics preserving (see Section 4 and 5). Interestingly, normalization of probabilities from terminating executions is a core issue that we needed to handle carefully in the proofs (unlike correctness proofs of slicing algorithms for non-probabilistic programs).

The notion of observe dependences is related to the concept of active trails [19] in Bayesian networks. We make this connection more precise in Section 2.

We have implemented the slicing algorithm as a source-to-source program transformation in the R2 probabilistic programming system [25]. We show using several benchmarks (in Section 6) that the SLI transformation greatly improves the efficiency of performing probabilistic inference by removing irrelevant statements, while providing provably equivalent answers. Our empirical results show that the sliced programs are not only smaller, but also result in faster convergence during inference. These efficiency gains are not just limited to R2. We apply the SLI transformation as a pre-processing tool and show efficiency improvements on two other tools that perform probabilistic inference—Church and Infer.NET.

2. Overview

In this section, we present some examples to familiarize the reader with probabilistic programs, and also informally explain the main ideas behind slicing probabilistic programs. We describe the formal details in later sections of the paper.

Examples of probabilistic programs. We introduce the syntax and semantics of probabilistic programs using two simple probabilistic programs from Figure 1. The program to the left, Example 1, tosses two fair coins (simulated by draws from a Bernoulli distribution with mean 0.5), and assigns the outcomes of these coin tosses to the Boolean variables `c1` and `c2` respectively. It also counts the number of coin tosses that result in the value `true`, and stores this count in the variable `count`. The semantics of the program is the probability distribution over its return values, which is Pr(`c1=false`, `c2=false`) = Pr(`c1=false`, `c2=true`) = Pr(`c1=true`, `c2=false`) = Pr(`c1=true`, `c2=true`) = 1/4. The program to the right of the figure, Example 2, is slightly different from Example 1—it executes the observe statement `observe(c1||c2)` before returning the value of `count`. The semantics of the `observe` statement blocks runs which do not satisfy the boolean expression `c1||c2` and does not permit those executions to happen. Executions that satisfy `c1||c2` are permitted to happen. The semantics of the program is the distribution of the return values, conditioned by permitted executions, which amounts to Pr(`c1=false,c2=false`) = 0, and Pr(`c1=false,c2=true`) = Pr(`c1=true,c2=false`) = Pr(`c1=true,c2=true`) = 1/3.

```
1:  bool d, i, s, l, g;
2:  d = Bernoulli(0.6);
3:  i = Bernoulli(0.7);

4:  if (!i && !d)
5:    g = Bernoulli(0.3);
6:  else if (!i && d)
7:    g = Bernoulli(0.05);
8:  else if (i && !d)
9:    g = Bernoulli(0.9);
10: else
11:   g = Bernoulli(0.5);

12: if (!i)
13:   s = Bernoulli(0.2);
14: else
15:   s = Bernoulli(0.95);

16: if (!g)
17:   l = Bernoulli(0.1);
18: else
19:   l = Bernoulli(0.4);

20: return s;
```

(a) Example 3.

```
1:  bool d, i, s, l, g;
2:  d = Bernoulli(0.6);
3:  i = Bernoulli(0.7);

4:  if (!i && !d)
5:    g = Bernoulli(0.3);
6:  else if (!i && d)
7:    g = Bernoulli(0.05);
8:  else if (i && !d)
9:    g = Bernoulli(0.9);
10: else
11:   g = Bernoulli(0.5);

12: if (!i)
13:   s = Bernoulli(0.2);
14: else
15:   s = Bernoulli(0.95);

16: if (!g)
17:   l = Bernoulli(0.1);
18: else
19:   l = Bernoulli(0.4);

20: observe(l = true);
21: return s;
```

(b) Example 4.

Figure 2. Examples to illustrate slicing of probabilistic programs.

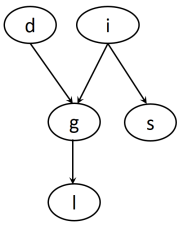


Figure 3. Dependency graph of Example 3.

We note that the statement `observe(x)` is very related to the statement `assume(x)` used in program verification literature [1, 8, 24]. Also, we note that `observe(x)` is equivalent to the while-loop `while(!x) skip` since the semantics of probabilistic programs is concerned about the normalized distribution of outputs over terminating runs of the program, and ignores non-terminating runs. However, we use the terminology `observe(x)` because of its common use in probabilistic programming systems [2, 12].

Slicing. We are interested in *slicing* a probabilistic program P and obtaining a simpler program SLI(P), which retains only those parts of P that are relevant to estimating the distribution over its return expression r, and elides those parts of P that are not relevant to estimating r. We desire that the SLI transformation be both correct

and efficient. By correctness, we mean that P and $\text{SLI}(P)$ have identical estimates for the distribution of r. By efficient, we mean that estimation over $\text{SLI}(P)$ be as fast as possible. We illustrate the intricacies in slicing probabilistic programs using a series of examples below.

Example where the usual definition of slicing works. We start with an example where the usual notion of slicing with dependences works for probabilistic programs as well. We consider the program Example 3 in the left side of Figure 2, with boolean variables variables `d, i, s, l, g`. This is adapted from [19], where the program represents a model for a reference letter (the variable `l`) for a student and this variable depends on the variables `d` (course difficulty), `i` (student intelligence), `g` (course grade), and `s` (SAT score). The dependency graph of this example is shown in Figure 3, where edges denote control or data dependences. Since the program returns the variable `s`, the dependences of `s` include only the variable `i`. Intuitively, the variables `d,g` and `l` and the statements that update these variables (lines 2, 4–11 and 16–19) are irrelevant, and can be sliced away from the program. This intuition is very similar to the usual notion of slicing in ordinary programs, where we collect relevant variables by tracing transitively control and data dependences for output variables, and slice away other variables that are irrelevant. Thus, the slice of this program is given by:

```
1:  bool i, s;
3:  i = Bernoulli(0.7);
12: if (!i)
13:   s = Bernoulli(0.2);
14: else
15:   s = Bernoulli(0.95);
20: return s;
```

Example where the usual definition of slicing is incorrect. Next, consider the program Example 4, in the right side of Figure 2. This program has the same structure as Example 3, and the only difference is the observe statement in line 20, which constrains the value of the variable `l` to be `true`. As before, the value of `s` is returned in line 21. The usual definition of slicing computes the same slice as before (with lines `1,3,12,13,14,15` and 20) as before, but this computation is incorrect, since the sliced program and the original program are not equivalent.

The observe statement introduces new kinds of influences that are not present in usual programs. Specifically, the observation of the value of `l` influences the value of `g`, which indirectly influences `i`, and ultimately influences `s`. In addition, this flow of influence from `i` to `g` also "opens up" a path for influence to flow from `d` to `i`, and ultimately to `s` as well. We explain these flow of influences more using a new notion called *observe dependence* below.

Thus, it turns out that *all* the variables `d, i, g, l` and `s` are relevant in this program, and the only slice that is semantically equivalent to Example 4 is the entire program!

Example where the usual definition of slicing is not efficient. Next, we show another example probabilistic program, where the usual definition of slicing produces a larger slice than what is necessary. Consider the program Example 5 in Figure 4. This is a variant of our previous example, with the variable `l` returned in line 21, and the observe statement in line 12 constraining the value of `g` to `false`. Since the program returns the variable `l`, the transitive dependences include `g, i` and `d`. Intuitively the variable `s` and the computations sliced away from the program, and such a slicing results in the program (b) in Figure 4.

However, due to the observe statement `observe(g = false)` at line 12, it turns out that we can stop traversing dependences at `g`, even though `g` depends on `i` and `d`. This can result in a smaller slice given below, which can be shown to be correct:

```
1:  bool d, i, s, l, g;
2:  d = Bernoulli(0.6);
3:  i = Bernoulli(0.7);

4:  if (!i && !d)
5:    g = Bernoulli(0.3);
6:  else if (!i && d)
7:    g = Bernoulli(0.05);
8:  else if (i && !d)
9:    g = Bernoulli(0.9);
10: else
11:   g = Bernoulli(0.5);

12: observe(g = false);

13: if (!i)
14:   s = Bernoulli(0.2);
15: else
16:   s = Bernoulli(0.95);

17: if (!g)
18:   l = Bernoulli(0.1);
19: else
20:   l = Bernoulli(0.4);

21: return l;
```

(a) Example 5.

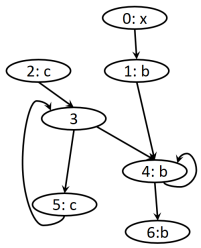
```
1:  bool d, i, l, g;
2:  d = Bernoulli(0.6);
3:  i = Bernoulli(0.7);

4:  if (!i && !d)
5:    g = Bernoulli(0.3);
6:  else if (!i && d)
7:    g = Bernoulli(0.05);
8:  else if (i && !d)
9:    g = Bernoulli(0.9);
10: else
11:   g = Bernoulli(0.5);

12: observe(g = false);

17: if (!g)
18:   l = Bernoulli(0.1);
19: else
20:   l = Bernoulli(0.4);

21: return l;
```

(b) Sliced Example 5 using usual dependencies.

Figure 4. Examples to illustrate slicing of probabilistic programs.

```
    bool x b c;
0:  x = Bernoulli(0.5);
1:  b = x;
2:  c = Bernoulli(0.5);
3:  while (c) do {
4:    b = !b; //toggle b
5:    c = Bernoulli(0.5);
    }
6:  observe (b = false);
7:  return x;
```

(a) Example 6



(b) Dependency graph

Figure 5. Loopy example.

```
1:  bool l, g;
12: g = false;
17: if (!g)
18:   l = Bernoulli(0.1);
19: else
20:   l = Bernoulli(0.4);
21: return l;
```

After constant propagation, this program can be further optimized to produce the simpler slice, equivalent to Example 5:

```
1:  bool l;
18: l = Bernoulli(0.1);
21: return l;
```

In Section 4, we define a transformation OBS which blocks spurious dependencies in observe statements, and produces the more efficient slice shown above.

Slicing programs with loops. Figure 5 shows Example 6, a probabilistic program with a while-do loop. In this example, the variable `c` is repeatedly sampled inside a loop, `b` is toggled until `c` becomes `false`. On exiting the loop, the program observes the value of `b` and returns `x`. In the right side of Figure 5, we show a simplified

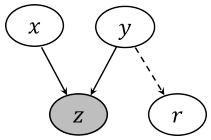


Figure 6. Observe dependences.

dependency graph, where we have labeled each vertex with a line number, so that it is clear what version of the variable (in static single assignment or SSA form [7]) we are referring to. We observe `b` after exiting the loop (which is the node `6:b` in the dependence graph), and return the value of `x`. Intuitively, the slice for this program needs to include the whole program, since the final value of `b` depends on all the iterations of the loop, and dependences from all these variables flow back and influence the probability of `x`, conditioned by the observation on `b`.

Observe dependences. Motivated by all the subtleties illustrated in the above examples, we have designed a slicing transformation SLI for probabilistic programs. In Section 4, we formally define the SLI transformation as a composition of 4 transformations: (1) OBS transformation, (2) SVF transformation, (3) SSA transformation, and (4) actual slicing transformation using an appropriate dependence relation called *influencers*, which is denoted INF.

We give the intuition behind the influencers relation INF (which captures dependences in probabilistic programs), deferring full details and formal proofs to Section 4. Usual notions of control and data dependences are captured by the relation DINF.

The relation INF captures the additional dependences we call as *observe dependences* in addition to the usual control and data dependences. Note that INF $\supseteq$ DINF. The main intuition behind observe dependence is as follows: suppose we have an observe statement $\texttt{observe}(z)$. Furthermore, suppose that we have two variables x and y such that z depends on both x and y using usual notions of control or data dependence (i.e., $x, y \in \text{DINF}(z)$). Let r be the return variable that is returned by the program. Suppose r depends on y, *i.e.,* $y \in \text{INF}(r)$. Figure 6 shows the dependences DINF between x, y, z and the INF dependence between y and r. Then, there is a path for influence to flow from x through z to y, and then from y to r, *i.e.,* $x \in \text{INF}(r)$. Intuitively, once we know what the value of z is (due to observing it), then knowledge about x influences our knowledge about the possible distribution of y and vice versa. These extra flows of influence are due to observe dependences.

Observe dependences are related to the notion of *active trails* in Bayesian networks [19]. In the parlance of Bayesian networks, there is a "v-structure" $x \rightarrow z \leftarrow y$, where the influences from x and y converge into z in a "v" shape. If z is observed, then the flow of influence between x and z gets "activated". On the other hand, if z is not observed, then the flow between x and z is blocked. As we show in Section 4, we combine this notion of observe dependences, together with control and data dependences to produce a correct and efficient slicing operator for probabilistic programs.

Comparison with non-termination preserving slicing. As discussed earlier, `observe(x)` has the same semantics as `while(!x) skip`. Thus, it makes sense to compare observe dependences with other notions of dependences and slicing operators that preserve non-terminating behaviors of non-probabilistic programs.

As a specific example, consider the program P given below:

```
x = Bernoulli(0.5);
while (!x) skip; //equivalent to: observe(x)
y = Bernoulli(0.6);
return y;
```

x	$\in$	Vars	
uop	::=	$\cdots$	C unary operators
bop	::=	$\cdots$	C binary operators
φ, ψ	::=	$\cdots$	logical formula
$\mathcal{E}$	::=		expressions
		$\mid x$	variable
		$\mid c$	constant
		$\mid \mathcal{E}_1$ **bop** $\mathcal{E}_2$	binary operation
		$\mid$ **uop** $\mathcal{E}$	unary operation
$\mathcal{S}$	::=		statements
		$\mid$ `skip`	skip
		$\mid x = \mathcal{E}$	deterministic assignment
		$\mid x \sim \texttt{Dist}(\bar{\theta})$	probabilistic assignment
		$\mid$ `observe` (φ)	observe
		$\mid \mathcal{S}_1; \mathcal{S}_2$	sequential composition
		$\mid$ `if` $\mathcal{E}$ `then` $\mathcal{S}_1$ `else` $\mathcal{S}_2$	conditional composition
		$\mid$ `while` $\mathcal{E}$ `do` $\mathcal{S}$	while−do loop
$\mathcal{P}$	::=	$\mathcal{S}$ `return` $\mathcal{E}$	program

Figure 7. Syntax of PROB.

In this case, our SLI operator (which uses control, data and observe dependences) produces the slice P' given below, which slices away the variable `x`:

```
y = Bernoulli(0.6);
return y;
```

As discussed in [13],

> "to preserve semantics of non-terminating executions, one needs to make sure that the slice includes any program points lying along paths to relevant nodes that could cause non-termination."

Thus, if we consider the assignment to `x` in the first line of program P as non-deterministic, then a non-termination preserving slice will have to include the while loop in the second line in the slice. In contrast, the SLI operator we define in this paper is able to produce the smaller slice P' which slices away the while loop.

Note that P' does *not* preserve non-terminating behaviors of P. However, if we consider the normalized probability distribution of the output over the terminating behaviors, then the distributions produced by P and P' coincide. This example illustrates the crux of the difference between slicing for probabilistic programs and non-termination preserving slicing for non-probabilistic programs.

3. Probabilistic Programs

The probabilistic programming language PROB that we consider is a C-like imperative programming language with two additional statements:

1. The *probabilistic assignment* "$\texttt{x} \sim \texttt{Dist}(\bar{\theta})$" draws a sample from a distribution `Dist` with a vector of parameters $\bar{\theta}$, and assigns it to the variable x. For instance, the statement "$\texttt{x} \sim \texttt{Gaussian}(\mu, \sigma^2)$" draws a value from a Gaussian distribution with mean μ and variance σ^2, and assigns it to the variable x.
2. The *observe statement* "$\texttt{observe}(\varphi)$" conditions a distribution with respect to a predicate or condition φ that is defined over the variables in the program. In particular, every valid execution of the program must satisfy all conditions in observe statements that occur along the execution.

• Unnormalized Semantics for Statements
$[\![\mathcal{S}]\!] \in (\Sigma \to [0,1]) \to \Sigma \to [0,1]$

$$[\![\texttt{skip}]\!](f)(\sigma) := f(\sigma)$$
$$[\![x = \mathcal{E}]\!](f)(\sigma) := f(\sigma[x \leftarrow \sigma(\mathcal{E})])$$
$$[\![x \sim \texttt{Dist}(\bar{\theta})]\!](f)(\sigma) := \int_{v \in \text{Val}} \texttt{Dist}(\sigma(\bar{\theta}))(v) \times f(\sigma[x \leftarrow v])\, dv$$
$$[\![\texttt{observe}(\varphi)]\!](f)(\sigma) := \begin{cases} f(\sigma) & \text{if } \sigma(\varphi) = \texttt{true} \\ 0 & \text{otherwise} \end{cases}$$
$$[\![\mathcal{S}_1; \mathcal{S}_2]\!](f)(\sigma) := [\![\mathcal{S}_1]\!]([\![\mathcal{S}_2]\!](f))(\sigma)$$
$$[\![\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2]\!](f)(\sigma) := \begin{cases} [\![\mathcal{S}_1]\!](f)(\sigma) & \text{if } \sigma(\mathcal{E}) = \texttt{true} \\ [\![\mathcal{S}_2]\!](f)(\sigma) & \text{otherwise} \end{cases}$$
$$[\![\texttt{while}\ \mathcal{E}\ \texttt{do}\ \mathcal{S}]\!](f)(\sigma) := \sup_{n \geq 0} [\![\texttt{while}\ \mathcal{E}\ \texttt{do}_n\ \mathcal{S}]\!](f)(\sigma)$$
where
$\texttt{while}\ \mathcal{E}\ \texttt{do}_0\ \mathcal{S} = \texttt{observe(false)}$
$\texttt{while}\ \mathcal{E}\ \texttt{do}_{n+1}\ \mathcal{S} = \texttt{if}\ \mathcal{E}\ \texttt{then}\ (\mathcal{S}; \texttt{while}\ \mathcal{E}\ \texttt{do}_n\ \mathcal{S})\ \texttt{else}\ (\texttt{skip})$

• Normalized Semantics for Programs
$[\![\mathcal{S}\ \texttt{return}\ \mathcal{E}]\!] \in (\mathbb{R} \to [0,1]) \to [0,1]$

$$[\![\mathcal{S}\ \texttt{return}\ \mathcal{E}]\!](f) := \frac{[\![\mathcal{S}]\!](\lambda\sigma.\ f(\sigma(\mathcal{E})))(\bot)}{[\![\mathcal{S}]\!](\lambda\sigma.\ 1)(\bot)}$$
where $\bot$ denotes the empty state

Figure 8. Denotational Semantics of PROB.

The syntax of PROB is formally described in Figure 7. A program consists of statements and a return expression. Variables have base types such as int, bool, float and double. Expressions include variables, constants, binary and unary operations.

Statements include primitive statements (skip, deterministic assignment, probabilistic assignment, observe) and composite statements (sequential composition, conditionals and loops). Features such as arrays, pointers, structures and function calls can be included in the language, and their treatment does not introduce any additional challenges due to probabilistic semantics. Therefore, we omit these features, and focus on a core language.

The semantics of PROB is described in Figure 8. A state σ of a program is a (partial) valuation to all its variables. The set of all states (which can be infinite) is denoted by Σ. We also consider the natural lifting of $\sigma : \text{Vars} \rightharpoonup \text{Val}$ to expressions $\sigma : \text{Exprs} \to \text{Val}$. We make this lifting a total function by assuming default values for uninitialized variables. The definition of the lifting σ for constants, unary and binary operations is standard.

The meaning of a probabilistic statement $\mathcal{S}$ is the probability distribution over all possible output states of $\mathcal{S}$ for any given initial state σ.[1] The semantics is completely specified using the rules in Figure 8. The `skip` statement merely applies the return function f to the input state σ, since the statement does not change the input state. The deterministic assignment statement first transforms the state σ by executing the assignment and then applies f. The meaning of the probabilistic assignment is the expected value obtained by sampling v from the distribution `Dist`, executing the assignment with v as the RHS value, and applying f on the resulting state (the expectation is the integral over all possible values v). The `observe` statement functions like a `skip` statement if the expression φ evaluates to `true` in the initial state σ, and returns the value 0 otherwise. The sequential and conditional statements behave as expected and the while-do loop has a standard fixpoint semantics.

Due to the presence of non-termination and observe statements, the semantics of statements shown in Figure 8 is unnormalized. The normalized semantics for programs is obtained by appropriately performing the normalization operation as shown in the second part of Figure 8.

[1] It is standard to represent a probability distribution on a set X as a function calculating the expected value of f w.r.t. the distribution on X for any given return function $f \in X \to [0,1]$, where $[0,1]$ is the unit interval (*i.e.,* the set of real numbers between 0 and 1, inclusive). Thus, the denotational semantics $[\![\mathcal{S}]\!](f)(\sigma)$ gives the expected value of return function $f \in \Sigma \to [0,1]$ when S is executed with initial state σ.

• $\text{OVAR}(\mathcal{S}) \in \mathbb{P}(\text{Vars})$

$$\begin{aligned} \text{OVAR}(\texttt{observe}(x)) &:= \{x\} \\ \text{OVAR}(\mathcal{S}_1; \mathcal{S}_2) &:= \text{OVAR}(\mathcal{S}_1) \cup \text{OVAR}(\mathcal{S}_2) \\ \text{OVAR}(\texttt{if}\ x\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2) &:= \text{OVAR}(\mathcal{S}_1) \cup \text{OVAR}(\mathcal{S}_2) \\ \text{OVAR}(\texttt{while}\ x\ \texttt{do}\ \mathcal{S}) &:= \{x\} \cup \text{OVAR}(\mathcal{S}) \\ \text{OVAR}(\mathcal{S}) &:= \emptyset \qquad \text{otherwise} \end{aligned}$$

• $\text{DEP}(\mathcal{S}) \in \mathbb{P}(\text{Vars}) \to \mathbb{P}(\text{Vars} \times \text{Vars})$

$$\begin{aligned} \text{DEP}(\texttt{skip})(C) &:= \emptyset \\ \text{DEP}(x = \mathcal{E})(C) &:= \{\, (y,x) \mid y \in C \cup \text{FV}(\mathcal{E}) \,\} \\ \text{DEP}(x \sim \texttt{Dist}(\bar{\theta}))(C) &:= \{\, (y,x) \mid y \in C \cup \text{FV}(\bar{\theta}) \,\} \\ \text{DEP}(\texttt{observe}(x))(C) &:= \{\, (y,x) \mid y \in C \,\} \\ \text{DEP}(\mathcal{S}_1; \mathcal{S}_2)(C) &:= \text{DEP}(\mathcal{S}_1)(C) \cup \text{DEP}(\mathcal{S}_2)(C) \\ \text{DEP}(\texttt{if}\ x\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2)(C) &:= \\ &\quad \text{DEP}(\mathcal{S}_1)(C \cup \{x\}) \cup \text{DEP}(\mathcal{S}_2)(C \cup \{x\}) \\ \text{DEP}(\texttt{while}\ x\ \texttt{do}\ \mathcal{S})(C) &:= \{\, (y,x) \mid y \in C \,\} \cup \text{DEP}(\mathcal{S})(C \cup \{x\}) \end{aligned}$$

Figure 9. Observed Variables and Dependency Graph Calculation.

• Direct Influencer

$$\frac{x \in R}{x \in \text{DINF}(\mathcal{G})(R)} \qquad \frac{(x,y) \in \mathcal{G} \quad y \in \text{DINF}(\mathcal{G})(R)}{x \in \text{DINF}(\mathcal{G})(R)}$$

• Influencer

$$\frac{x \in \text{DINF}(\mathcal{G})(R)}{x \in \text{INF}(\mathcal{O}, \mathcal{G})(R)} \qquad \frac{\begin{array}{c} x, y \in \text{DINF}(\mathcal{G})(\{z\}) \\ z \in \mathcal{O} \quad y \in \text{INF}(\mathcal{O}, \mathcal{G})(R) \end{array}}{x \in \text{INF}(\mathcal{O}, \mathcal{G})(R)}$$

Figure 10. Influencer Calculation.

• $\text{SLI}(\mathcal{S}) \in \mathbb{P}(\text{Vars}) \to \text{Statement}$

$$\text{SLI}(\texttt{skip})(X) := \texttt{skip}$$
$$\text{SLI}(x = \mathcal{E})(X) := \begin{cases} x = \mathcal{E} & \text{if } x \in X \\ \texttt{skip} & \text{otherwise} \end{cases}$$
$$\text{SLI}(x \sim \texttt{Dist}(\bar{\theta}))(X) := \begin{cases} x \sim \texttt{Dist}(\bar{\theta}) & \text{if } x \in X \\ \texttt{skip} & \text{otherwise} \end{cases}$$
$$\text{SLI}(\texttt{observe}(x))(X) := \begin{cases} \texttt{observe}(x) & \text{if } x \in X \\ \texttt{skip} & \text{otherwise} \end{cases}$$
$$\text{SLI}(\mathcal{S}_1; \mathcal{S}_2)(X) := \text{SLI}(\mathcal{S}_1)(X); \text{SLI}(\mathcal{S}_2)(X)$$
$$\text{SLI}(\texttt{if}\ x\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2)(X) :=$$
$$\begin{cases} \texttt{skip} & \text{if } \text{SLI}(\mathcal{S}_1)(X) = \text{SLI}(\mathcal{S}_2)(X) = \texttt{skip} \\ \texttt{if}\ x\ \texttt{then}\ \text{SLI}(\mathcal{S}_1)(X)\ \texttt{else}\ \text{SLI}(\mathcal{S}_2)(X) & \text{otherwise} \end{cases}$$
$$\text{SLI}(\texttt{while}\ x\ \texttt{do}\ \mathcal{S})(X) := \begin{cases} \texttt{while}\ x\ \texttt{do}\ \text{SLI}(\mathcal{S})(X) & \text{if } x \in X \\ \texttt{skip} & \text{otherwise} \end{cases}$$

• $\text{SLI}(\mathcal{S}\ \texttt{return}\ \mathcal{E}) := \text{SLI}(\mathcal{S})(\text{INF}(\mathcal{O}, \mathcal{G})(R))\ \texttt{return}\ \mathcal{E}$
where
$\mathcal{O} = \text{OVAR}(\mathcal{S}), \mathcal{G} = \text{DEP}(\mathcal{S})(\emptyset), R = \text{FV}(\mathcal{E})$

Figure 11. Slicing Transformation.

4. The Slice Transformation

We describe a transformation SLI such that given a program P, the program $\text{SLI}(P)$ is semantically equivalent to P, and contains only statements that are "relevant" to the variables present in the return expression of P.

To simplify our presentation, we assume that the given program P is in SSA form [7] (*i.e.,* each variable is assigned only once in the program), and that all predicates in observe statements as

$$
\begin{aligned}
\textsc{Obs}(\texttt{observe}(\mathcal{E})) &:= \texttt{observe}(\mathcal{E}); \textsc{ObserveSet}(\mathcal{E}) \\
\textsc{Obs}(\texttt{while}\ \mathcal{E}\ \texttt{do}\ \mathcal{S}) &:= (\texttt{while}\ \mathcal{E}\ \texttt{do}\ \textsc{Obs}(\mathcal{S})); \textsc{WhileSet}(\mathcal{E}) \\
\textsc{Obs}(\mathcal{S}_1; \mathcal{S}_2) &:= \textsc{Obs}(\mathcal{S}_1); \textsc{Obs}(\mathcal{S}_2) \\
\textsc{Obs}(\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2) &:= \texttt{if}\ \mathcal{E}\ \texttt{then}\ \textsc{Obs}(\mathcal{S}_1)\ \texttt{else}\ \textsc{Obs}(\mathcal{S}_2) \\
\textsc{Obs}(\mathcal{S}) &:= \mathcal{S},\ \text{otherwise}
\end{aligned}
$$

$$
\textsc{ObserveSet}(\mathcal{E}) := \begin{cases} x = \mathcal{E}' & \text{if } \mathcal{E} \text{ is } (x = \mathcal{E}') \text{ or } (\mathcal{E}' = x) \\ & \quad \text{for } \mathcal{E}' \text{ with no variables} \\ \texttt{skip} & \text{otherwise} \end{cases}
$$

$$
\textsc{WhileSet}(\mathcal{E}) := \begin{cases} x = \mathcal{E}' & \text{if } \mathcal{E} \text{ is } (x \neq \mathcal{E}') \text{ or } (\mathcal{E}' \neq x) \\ & \quad \text{for } \mathcal{E}' \text{ with no variables} \\ \texttt{skip} & \text{otherwise} \end{cases}
$$

$$
\bullet\ \textsc{Obs}(\mathcal{S}\ \texttt{return}\ \mathcal{E}) := \textsc{Obs}(\mathcal{S})\ \texttt{return}\ \mathcal{E}
$$

Figure 12. Observation Transformation.

$$
\begin{aligned}
\textsc{Svf}(\texttt{observe}(\mathcal{E})) &:= \textbf{let}\ x' \in \mathit{freshvar}()\ \textbf{in} \\
&\quad\ x' = \mathcal{E}; \texttt{observe}(x') \\
\textsc{Svf}(\texttt{while}\ \mathcal{E}\ \texttt{do}\ \mathcal{S}) &:= \textbf{let}\ x' \in \mathit{freshvar}()\ \textbf{in} \\
&\quad\ x' = \mathcal{E}; \texttt{while}\ x'\ \texttt{do}\ (\mathcal{S}; x' = \mathcal{E}) \\
\textsc{Svf}(\mathcal{S}_1; \mathcal{S}_2) &:= \textsc{Svf}(\mathcal{S}_1); \textsc{Svf}(\mathcal{S}_2) \\
\textsc{Svf}(\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2) &:= \textbf{let}\ x' \in \mathit{freshvar}()\ \textbf{in} \\
&\quad\ x' = \mathcal{E}; \texttt{if}\ x'\ \texttt{then}\ \textsc{Svf}(\mathcal{S}_1)\ \texttt{else}\ \textsc{Svf}(\mathcal{S}_2) \\
\textsc{Svf}(\mathcal{S}) &:= \mathcal{S},\ \text{otherwise}
\end{aligned}
$$

$$
\bullet\ \textsc{Svf}(\mathcal{S}\ \texttt{return}\ \mathcal{E}) := \textsc{Svf}(\mathcal{S})\ \texttt{return}\ \mathcal{E}
$$

Figure 13. Single Variable Form Transformation.

well as while loops are single boolean variables (*i.e.,* each observe statement is of the form $\texttt{observe}(x)$ and each while loop is of the form $\texttt{while}\ (y)\ \texttt{do}\ \mathcal{S}$, where x and y are boolean variables). We ensure that these conditions hold on P by performing a pre-pass which performs the following steps: (1) convert the program to SSA form, (2) introduce a fresh boolean variable that holds the value of the predicate or conditional expression for every observe and while statement. We discuss the pre-pass in more detail in Section 4.2.

4.1 Main Transformation

In order to slice a program P, we first build its dependency graph, and calculate all variables that influence the variables returned by P. Recall from Section 2 that the notion of influences we need include control and data dependences from traditional slicing literature, as well as the new observe dependences (see Figure 6 in Section 2). In this section, we formally define the transformation SLI.

Dependency graph. We first calculate the set of observed variables and dependency graph by making passes over the program, as shown in Figure 9. The calculation of the set of observed variables OVAR is done by structural induction over the statements of the program and accumulating the arguments of observe statements as well as conditional expressions of while-do statements. The dependency graph DEP is a binary relation over the variables. It is also calculated by collecting both the control and data dependences during structural induction over the statements of the program. The function DEP takes the control dependences for executing the current statement as an argument. This argument (C in Figure 9) accumulates the variables which are conditions of if-then-else and while-do statements. During processing of deterministic and probabilistic assignments, in addition to data flow dependences from the right-hand side of the assignment, the control dependences are also added from C. The observe statement merely accumulates control dependences.

Influencers. The core step of the slicing algorithm is the calculation of the variables in the program that influence the return values

$$
\begin{aligned}
&\bullet\ \textsc{SSA}(\mathcal{S}) \in \mathbb{P}(\text{Vars}) \times \text{Ren} \to \mathbb{P}(\text{Vars}) \times \text{Ren} \times \text{Statement} \\
&\quad \text{for Ren} = \text{Vars} \rightharpoonup \text{Vars} \\
&\quad \textsc{SSA}(\texttt{skip})(X, \rho)\ :=\ (X, \rho, \texttt{skip}) \\
&\quad \textsc{SSA}(\texttt{observe}(\mathcal{E}))(X, \rho)\ :=\ (X, \rho, \texttt{observe}(\rho(\mathcal{E}))) \\
&\quad \textsc{SSA}(x = \mathcal{E})(X, \rho)\ := \\
&\qquad \textbf{let}\ x' \notin X\ \textbf{in} \\
&\qquad (X \cup \{x'\}, \rho[x \mapsto x'], x' = \rho(\mathcal{E})) \\
&\quad \textsc{SSA}(x \sim \texttt{Dist}(\bar{\theta}))(X, \rho)\ := \\
&\qquad \textbf{let}\ x' \notin X\ \textbf{in} \\
&\qquad (X \cup \{x'\}, \rho[x \mapsto x'], x' \sim \texttt{Dist}(\rho(\bar{\theta}))) \\
&\quad \textsc{SSA}(\mathcal{S}_1; \mathcal{S}_2)(X, \rho)\ := \\
&\qquad \textbf{let}\ (X', \rho', \mathcal{S}_1') = \textsc{SSA}(\mathcal{S}_1)(X, \rho)\ \textbf{and} \\
&\qquad \textbf{let}\ (X'', \rho'', \mathcal{S}_2') = \textsc{SSA}(\mathcal{S}_2)(X', \rho')\ \textbf{in} \\
&\qquad (X'', \rho'', \mathcal{S}_1'; \mathcal{S}_2') \\
&\quad \textsc{SSA}(\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2)(X, \rho)\ := \\
&\qquad \textbf{let}\ (X', \rho', \mathcal{S}_1') = \textsc{SSA}(\mathcal{S}_1)(X, \rho)\ \textbf{and} \\
&\qquad \textbf{let}\ (X'', \rho'', \mathcal{S}_2') = \textsc{SSA}(\mathcal{S}_2)(X', \rho)\ \textbf{and} \\
&\qquad \textbf{let}\ \mathcal{S}_2'' = \textsc{Merge}(\rho', \rho'')\ \textbf{in} \\
&\qquad (X'', \rho', \texttt{if}\ \rho(\mathcal{E})\ \texttt{then}\ \mathcal{S}_1'\ \texttt{else}\ (\mathcal{S}_2'; \mathcal{S}_2'')) \\
&\quad \textsc{SSA}(\texttt{while}\ \mathcal{E}\ \texttt{do}\ \mathcal{S})(X, \rho)\ := \\
&\qquad \textbf{let}\ (X', \rho', \mathcal{S}') = \textsc{SSA}(\mathcal{S})(X, \rho)\ \textbf{and} \\
&\qquad \textbf{let}\ \mathcal{S}'' = \textsc{Merge}(\rho, \rho')\ \textbf{in} \\
&\qquad (X', \rho, \texttt{while}\ \rho(\mathcal{E})\ \texttt{do}\ (\mathcal{S}'; \mathcal{S}''))
\end{aligned}
$$

$$
\begin{aligned}
&\textsc{Merge}(\rho, \rho')\ :=\ \textsc{Merge}_{\text{rec}}(\rho, \rho', \text{dom}(\rho)) \\
&\textsc{Merge}_{\text{rec}}(\rho, \rho', \emptyset)\ :=\ \texttt{skip} \\
&\textsc{Merge}_{\text{rec}}(\rho, \rho', \{x\} \uplus X)\ := \\
&\quad \begin{cases} (\rho(x) = \rho'(x); \textsc{Merge}_{\text{rec}}(\rho, \rho', X)) & \text{if } \rho(x) \neq \rho'(x) \\ \textsc{Merge}_{\text{rec}}(\rho, \rho', X) & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&\bullet\ \textsc{SSA}(\mathcal{S}\ \texttt{return}\ \mathcal{E})\ := \\
&\quad \textbf{let}\ X = \textsc{Fv}(\mathcal{S}) \cup \textsc{Fv}(\mathcal{E})\ \textbf{and} \\
&\quad \textbf{let}\ (\_, \rho', \mathcal{S}') = \textsc{SSA}(\mathcal{S})(X, \text{ID}_X)\ \textbf{in} \\
&\quad \mathcal{S}'\ \texttt{return}\ \rho'(\mathcal{E})
\end{aligned}
$$

Figure 14. SSA Transformation.

of the program using the dependency graph. We call these variables as *influencers*. Recall from Section 3 that probabilistic programs have a return expression of the form $\texttt{return}(\mathcal{E})$. R is the set of free variables in $\mathcal{E}$, and our goal is to calculate all the variables in program P that influence R. Let $\mathcal{G}$ be the dependency graph of P calculated using the rules in Figure 9. The direct influencers of R denoted by $\textsc{DInf}(\mathcal{G})(R)$ is the set of variables obtained by traversing the edges in the dependency graph backwards starting with the return variables (this is done by the top two rules in Figure 10). Since the dependency graph includes control and data dependences, the set $\textsc{DInf}(\mathcal{G})(R)$ contains all the variables that influence the return variables through control and data dependences.

As mentioned in Section 2, a new set of dependences called observe dependences arise naturally in probabilistic programs. Let $\mathcal{O}$ be the set of observed variables of P calculated using the rules in Figure 9. The set of influencers of R is denoted by $\textsc{Inf}(\mathcal{O}, \mathcal{G})(R)$ and contains the set of all variables that influence the return variables through control, data and observe dependences. $\textsc{Inf}(\mathcal{O}, \mathcal{G})(R)$ is calculated inductively using the two rules in the bottom portion of Figure 10. The first rule (bottom left) merely includes every element from $\textsc{DInf}(\mathcal{G})(R)$ into $\textsc{Inf}(\mathcal{O}, \mathcal{G})(R)$. The second rule (bottom right) captures observe dependences. In particular, if y is in $\textsc{Inf}(\mathcal{O}, \mathcal{G})(R)$, and z is any observed variable, and y and another variable x are in $\textsc{DInf}(\mathcal{G})(z)$, then there is an indirect influence from x through the observed variable z to the set of return variables R. Thus, we add x to $\textsc{Inf}(\mathcal{O}, \mathcal{G})(R)$. This rule captures the essence of influence due to observe statements. This rule is inspired by the notion of active trails in Bayesian networks [19].

Slicing Transformation. Once the set of influencers have been calculated, the slicing transformation SLI is easy to define, and is

stated in Figure 11. This transformation takes as input the set of influencers $\textsc{Inf}(\mathcal{O},\mathcal{G})(R)$ calculated as shown above. It performs structural induction over the statements of the program and retains only the assignment and observe statements over variables that are in the input set of influencers. All other statements (which have variables that are not in the influencer set) are replaced by skip statements. In Section 5, we state and prove a theorem establishing that the SLI transformation is semantics preserving.

4.2 Pre-pass Transformations

The description of the slicing transformation above assumed that the program P is in SSA form and that all predicates in observe statements and while loops are boolean variables. Here, we describe a set of pre-pass transformations, which ensure that these assumptions are valid. In addition to ensuring the validity of these assumptions, we use a transformation called OBS in the pre-pass to add an assignment statement after every observe statement and reduce certain kind of dependencies (ultimately reducing the size of the slice).

More precisely, the steps in the pre-pass are as follows:

- Removing spurious dependences using the OBS transformation.
- Ensuring that boolean expressions in observe statements, as well as conditions of if-then-else and while-do statements are boolean variables (and not general boolean expressions) using the SVF (Single Variable Form) transformation.
- Translating into the SSA form.

The OBS transformation, given in Figure 12 prunes spurious dependences in the presence of observe statements of the form $\texttt{observe}(\texttt{x} = \mathcal{E})$. Recall that we saw such spurious dependences in Example 5 of Section 2, where even though variable `g` depends on variables `d` and `i`, these dependencies can be pruned due to the observe statement $\texttt{observe}(\texttt{g} = \texttt{false})$ in line 12. The transformation is very simple. Informally, it adds an assignment $\texttt{x} = \mathcal{E}$ immediately after every observe statement of the form $\texttt{observe}(\texttt{x} = \mathcal{E})$, and every while-do loop whose condition is of the form $\texttt{x} \neq \mathcal{E}$. A precise description of the OBS transformation is given in Figure 12. Intuitively, the introduced assignment statement blocks dependencies that do not need to be traversed. For every probabilistic program P, we can show that P and $\textsc{Obs}(P)$ are semantically equivalent by a simple induction on the structure of the program.

The SVF transformation shown in Figure 13 ensures that every condition in observe, if-then-else and while-do statements are replaced by a single boolean variable. In particular, for every condition in the above statements, the transformation creates a fresh variable (call to *freshvar*() in Figure 13) and stores the condition in that variable. As a result, it is safe to replace the condition with the new variable.

The SSA transformation we use is shown in Figure 14. It is a variant of the standard SSA transformation that avoids the use of phi-nodes by relaxing the SSA condition that there should be only one assignment for each variable. This relaxation, however, does not cause any inefficiency in slicing but allows us to have a simple *compositional* semantics for our probabilistic programming language.

4.3 Worked Out Examples

We present two worked out examples in Figures 15 and 16. In each example, we transform a given original program using the three pre-pass transformations OBS, SVF and SSA. Note that we do not include the return statement in the pre-pass transformations as they are independent of the return statement. After the pre-pass transformations, we consider two different return statements and slice the transformed program with respect to each return statement using the SLI transformation. In Figure 15 and 16, we also include dependence graph and influencer calculations used by the SLI transformation.

$$\textsc{Aux}(\texttt{skip}) := \texttt{skip}$$

$$\textsc{Aux}(x = \mathcal{E}) := \begin{cases} x = \mathcal{E} & \text{if } \textsc{DInf}(\mathcal{G})(x) \cap \mathcal{X} = \emptyset \\ \texttt{skip} & \text{otherwise} \end{cases}$$

$$\textsc{Aux}(x \sim \texttt{Dist}(\bar{\theta})) := \begin{cases} x \sim \texttt{Dist}(\bar{\theta}) & \text{if } \textsc{DInf}(\mathcal{G})(x) \cap \mathcal{X} = \emptyset \\ \texttt{skip} & \text{otherwise} \end{cases}$$

$$\textsc{Aux}(\texttt{observe}(x)) := \begin{cases} \texttt{observe}(x) & \text{if } \textsc{DInf}(\mathcal{G})(x) \cap \mathcal{X} = \emptyset \\ \texttt{skip} & \text{otherwise} \end{cases}$$

$$\textsc{Aux}(\mathcal{S}_1; \mathcal{S}_2) := \textsc{Aux}(\mathcal{S}_1); \textsc{Aux}(\mathcal{S}_2)$$

$$\textsc{Aux}(\texttt{if}\ x\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2) := \begin{cases} \texttt{skip} & \text{if } \textsc{Aux}(\mathcal{S}_1) = \textsc{Aux}(\mathcal{S}_2) = \texttt{skip} \\ \texttt{if}\ x\ \texttt{then}\ \textsc{Aux}(\mathcal{S}_1)\ \texttt{else}\ \textsc{Aux}(\mathcal{S}_2) & \text{otherwise} \end{cases}$$

$$\textsc{Aux}(\texttt{while}\ x\ \texttt{do}\ \mathcal{S}) := \begin{cases} \texttt{while}\ x\ \texttt{do}\ \textsc{Aux}(\mathcal{S}) & \text{if } \textsc{DInf}(\mathcal{G})(x) \cap \mathcal{X} = \emptyset \\ \texttt{skip} & \text{otherwise} \end{cases}$$

Figure 17. AUX Transformation.

5. Correctness of the Slice Transformation

In this section, we prove that the SLI transformation is semantics preserving. It is important to note that this correctness proof only relies on the single variable form (SVF) assumption. If an input program is not in SSA form, then the SLI transformation could be inefficient, but still correct.

Let $\mathcal{X} = \textsc{Inf}(\mathcal{O},\mathcal{G})(R)$ for any given $\mathcal{O}$, $\mathcal{G}$ and R. We will simply write $\textsc{Sli}(\mathcal{S})$ for $\textsc{Sli}(\mathcal{S})(\mathcal{X})$. We also define $\textsc{Aux}(\mathcal{S})$ in Figure 17.

LEMMA 1. $\textsc{Inf}(\mathcal{O},\mathcal{G})(\mathcal{X}) \subseteq \mathcal{X}$.

Proof: We prove the goal by an induction on $\textsc{Inf}(\mathcal{O},\mathcal{G})(\mathcal{X})$. The step case holds obviously by definition of $\mathcal{X}$. For the base case, we need to show $\textsc{DInf}(\mathcal{G})(\mathcal{X}) \subseteq \mathcal{X}$, which in turn can be shown by an easy induction on $\textsc{DInf}(\mathcal{G})(\mathcal{X})$. ■

LEMMA 2. *For $\mathcal{S}, C$ with $\textsc{OVar}(\mathcal{S}) \subseteq \mathcal{O} \wedge \textsc{Dep}(\mathcal{S})(C) \subseteq \mathcal{G}$, if $\textsc{DInf}(\mathcal{G})(C) \not\subseteq \mathcal{X}$, then $\textsc{Sli}(\mathcal{S}) = \texttt{skip}$.*

Proof: One can easily prove the goal by an induction on $\mathcal{S}$ using the fact that $\forall x.\ (\forall y \in C.\ (y,x) \in \textsc{Dep}(\mathcal{S})(C)) \implies x \notin \mathcal{X}$, which follows from $\textsc{DInf}(\mathcal{G})(C) \not\subseteq \mathcal{X}$ by Lemma 1. ■

LEMMA 3. *For $\mathcal{S}, C$ with $\textsc{OVar}(\mathcal{S}) \subseteq \mathcal{O} \wedge \textsc{Dep}(\mathcal{S})(C) \subseteq \mathcal{G}$, if $\textsc{DInf}(\mathcal{G})(C) \cap \mathcal{X} \neq \emptyset$, then $\textsc{Aux}(\mathcal{S}) = \texttt{skip}$.*

Proof: One can easily prove the goal by an induction on $\mathcal{S}$ using the fact that $\forall x.\ (\forall y \in C.\ (y,x) \in \textsc{Dep}(\mathcal{S})(C)) \implies \textsc{DInf}(\mathcal{G})(x) \cap \mathcal{X} \neq \emptyset$, which follows from $\textsc{DInf}(\mathcal{G})(C) \cap \mathcal{X} \neq \emptyset$ by Lemma 1. ■

We define $\sigma|_X$ as follows:

$$\sigma|_X(x) = \begin{cases} \sigma(x) & \text{if } x \in X \\ \text{undefined} & \text{otherwise} \end{cases}$$

We say $f : \Sigma \to [0,1]$ decomposes into f_1 and f_2 w.r.t. Y if:

1. $\textsc{DInf}(\mathcal{G})(Y) \cap \mathcal{X} = \emptyset$;
2. $\forall \sigma.\ f_2(\sigma) = f_2(\sigma|_Y)$; and
3. $\forall \sigma.\ f(\sigma) = f_1(\sigma|_{\mathcal{X}}) \times f_2(\sigma|_Y)$.

We also say $H : (\Sigma \to [0,1]) \to (\Sigma \to [0,1])$ decomposes into

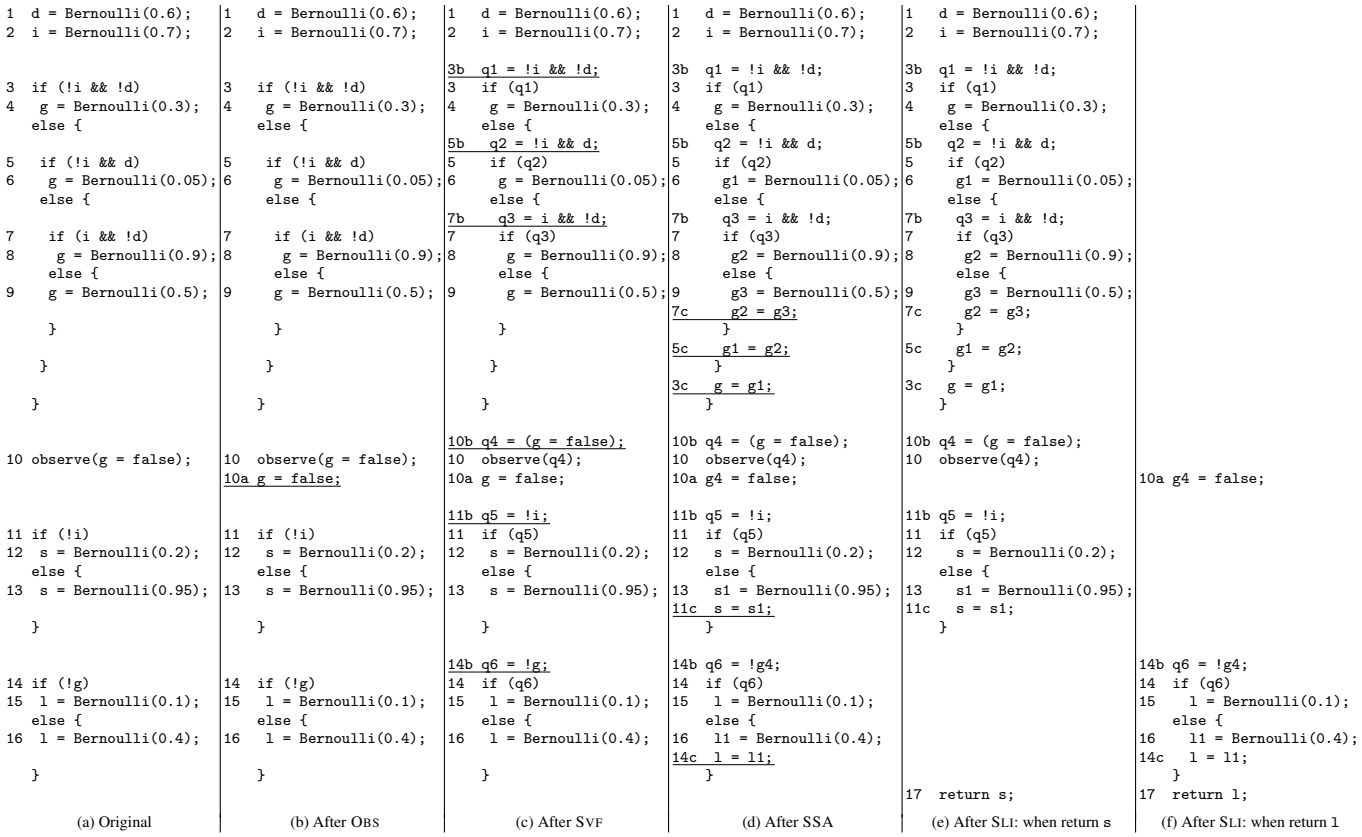


Figure 15. Worked out Example 1.

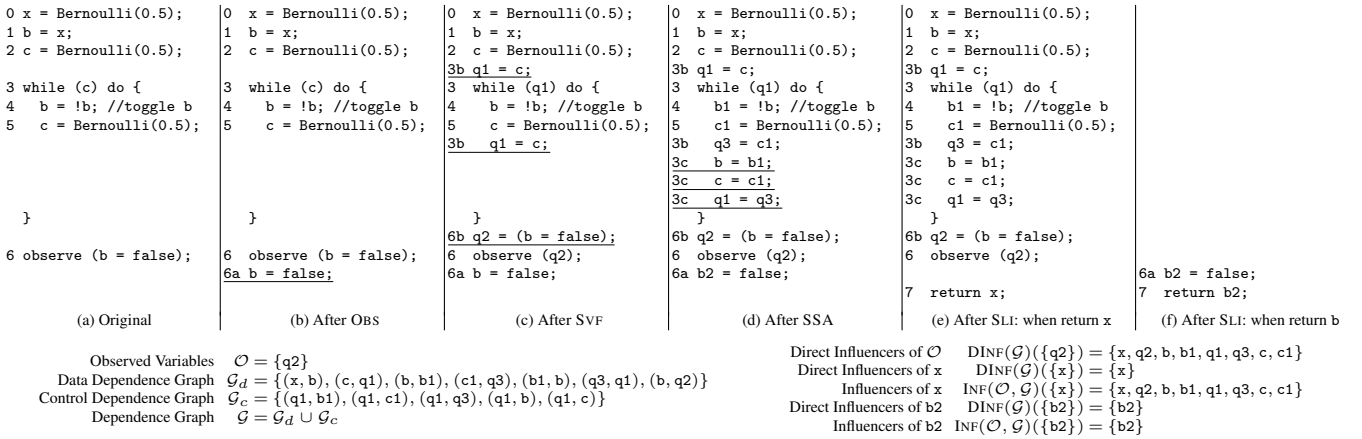


Figure 16. Worked out Example 2.

H_1 and H_2 w.r.t. Z if for any f that decomposes into f_1 and f_2 w.r.t. Y, $H(f)$ decomposes into $H_1(f_1)$ and $H_2(f_2)$ w.r.t. $Y \cup Z$. Note that if f decomposes into f_1 and f_2 w.r.t. Y, then f also decomposes into f_1 and f_2 w.r.t. Y', for any $Y' \supseteq Y$ such that $\text{DINF}(\mathcal{G})(Y') \cap \mathcal{X} = \emptyset$.

LEMMA 4. *For $\mathcal{S}, C$ with $\text{OVAR}(\mathcal{S}) \subseteq \mathcal{O} \wedge \text{DEP}(\mathcal{S})(C) \subseteq \mathcal{G}$, there exists some Z such that $[\![\mathcal{S}]\!]$ decomposes into $[\![\text{SLI}(\mathcal{S})]\!]$ and $[\![\text{AUX}(\mathcal{S})]\!]$ w.r.t. Z.*

Proof: We prove the goal by structural induction on $\mathcal{S}$. Let f be any function that decomposes into f_1 and f_2 w.r.t. Y.

• When $\mathcal{S}$ is $\texttt{skip}$.
The goal holds trivially with $Z = \emptyset$.

• When $\mathcal{S}$ is $\texttt{observe}(x)$.
If $x \in \mathcal{X}$, one can easily show the goal with $Z = \emptyset$. If $x \notin \mathcal{X}$, we have $\textsc{Dinf}(\mathcal{G})(x) \cap \mathcal{X} = \emptyset$ by Lemma 1 because $x \in \mathcal{O}$. Thus we can set $Z = \{x\}$ and the goal can be easily shown.

• When $\mathcal{S}$ is $x = \mathcal{E}$.
Suppose $\textsc{Dinf}(\mathcal{G})(x) \cap \mathcal{X} \neq \emptyset$ and set $Z = \emptyset$. Then we have $x \notin Y$. We can show the goal as follows:

$$\begin{aligned}
&[\![x = \mathcal{E}]\!](f)(\sigma) \\
&= f(\sigma[x \leftarrow \sigma(\mathcal{E})]) \\
&= f_1(\sigma[x \leftarrow \sigma(\mathcal{E})]|_{\mathcal{X}}) \times f_2(\sigma[x \leftarrow \sigma(\mathcal{E})]|_{Y}) \\
&= f_1(\sigma[x \leftarrow \sigma(\mathcal{E})]|_{\mathcal{X}}) \times f_2(\sigma|_{Y}) \qquad (\text{because } x \notin Y) \\
&= [\![\textsc{Sli}(x = \mathcal{E})]\!](f_1)(\sigma|_{\mathcal{X}}) \times [\![\textsc{Aux}(x = \mathcal{E})]\!](f_2)(\sigma|_{Y})
\end{aligned}$$

Here $[\![\textsc{Sli}(x = \mathcal{E})]\!](f_1)(\sigma|_{\mathcal{X}}) = f_1(\sigma[x \leftarrow \sigma(\mathcal{E})]|_{\mathcal{X}})$ can be shown easily when $x \notin \mathcal{X}$. It can also be shown when $x \in \mathcal{X}$ because $\textsc{Dinf}(\mathcal{G})(\mathcal{E}) \subseteq \textsc{Dinf}(\mathcal{G})(x) \subseteq \mathcal{X}$ by Lemma 1.

Now suppose $\textsc{Dinf}(\mathcal{G})(x) \cap \mathcal{X} = \emptyset$ and set $Z = \textsc{Dinf}(\mathcal{G})(x)$. Using the fact that $\textsc{Dinf}(\mathcal{G})(\mathcal{E}) \subseteq \textsc{Dinf}(\mathcal{G})(x) \subseteq Y \cup Z$, one can easily show that

– $\forall \sigma.\ [\![\textsc{Aux}(\mathcal{S})]\!](f_2)(\sigma) = [\![\textsc{Aux}(\mathcal{S})]\!](f_2)(\sigma|_{Y \cup Z})$; and
– $\forall \sigma.\ [\![\mathcal{S}]\!](f)(\sigma) = [\![\textsc{Sli}(\mathcal{S})]\!](f_1)(\sigma|_{\mathcal{X}}) \times [\![\textsc{Aux}(\mathcal{S})]\!](f_2)(\sigma|_{Y \cup Z})$.

• When $\mathcal{S}$ is $x \sim \texttt{Dist}(\bar{\theta})$.
Suppose $\textsc{Dinf}(\mathcal{G})(x) \cap \mathcal{X} \neq \emptyset$ and set $Z = \emptyset$. Then we have $x \notin Y$. We can show the goal as follows:

$$\begin{aligned}
&[\![x \sim \texttt{Dist}(\bar{\theta})]\!](f)(\sigma) \\
&= \textstyle\int_{v \in \mathrm{Val}} \texttt{Dist}(\sigma(\bar{\theta}))(v) \times f(\sigma[x \leftarrow v])\, dv \\
&= \textstyle\int_{v \in \mathrm{Val}} \texttt{Dist}(\sigma(\bar{\theta}))(v) \times f_1(\sigma[x \leftarrow v]|_{\mathcal{X}}) \times f_2(\sigma[x \leftarrow v]|_{Y})\, dv \\
&= \textstyle\int_{v \in \mathrm{Val}} \texttt{Dist}(\sigma(\bar{\theta}))(v) \times f_1(\sigma[x \leftarrow v]|_{\mathcal{X}}) \times f_2(\sigma|_{Y})\, dv \\
&\qquad (\text{because } x \notin Y) \\
&= (\textstyle\int_{v \in \mathrm{Val}} \texttt{Dist}(\sigma(\bar{\theta}))(v) \times f_1(\sigma[x \leftarrow v]|_{\mathcal{X}})\, dv) \times f_2(\sigma|_{Y}) \\
&= [\![\textsc{Sli}(x \sim \texttt{Dist}(\bar{\theta}))]\!](f_1)(\sigma|_{\mathcal{X}}) \times [\![\textsc{Aux}(x \sim \texttt{Dist}(\bar{\theta}))]\!](f_2)(\sigma|_{Y})
\end{aligned}$$

Here

$$\begin{aligned}
&[\![\textsc{Sli}(x \sim \texttt{Dist}(\bar{\theta}))]\!](f_1)(\sigma|_{\mathcal{X}}) \\
&= \textstyle\int_{v \in \mathrm{Val}} \texttt{Dist}(\sigma(\bar{\theta}))(v) \times f_1(\sigma[x \leftarrow v]|_{\mathcal{X}})\, dv
\end{aligned}$$

can be shown easily when $x \notin \mathcal{X}$ because

$$\textstyle\int_{v \in \mathrm{Val}} \texttt{Dist}(\sigma(\bar{\theta}))(v)\, dv = 1\ .$$

It can also be shown when $x \in \mathcal{X}$ because $\textsc{Dinf}(\mathcal{G})(\bar{\theta}) \subseteq \textsc{Dinf}(\mathcal{G})(x) \subseteq \mathcal{X}$ by Lemma 1.

Now suppose $\textsc{Dinf}(\mathcal{G})(x) \cap \mathcal{X} = \emptyset$ and set $Z = \textsc{Dinf}(\mathcal{G})(x)$. Using the fact that $\textsc{Dinf}(\mathcal{G})(\bar{\theta}) \subseteq \textsc{Dinf}(\mathcal{G})(x) \subseteq Y \cup Z$, one can easily show that

– $\forall \sigma.\ [\![\textsc{Aux}(\mathcal{S})]\!](f_2)(\sigma) = [\![\textsc{Aux}(\mathcal{S})]\!](f_2)(\sigma|_{Y \cup Z})$; and
– $\forall \sigma.\ [\![\mathcal{S}]\!](f)(\sigma) = [\![\textsc{Sli}(\mathcal{S})]\!](f_1)(\sigma|_{\mathcal{X}}) \times [\![\textsc{Aux}(\mathcal{S})]\!](f_2)(\sigma|_{Y \cup Z})$.

• When $\mathcal{S}$ is $\mathcal{S}_1; \mathcal{S}_2$.
By the induction hypothesis, we have some Z_1 for $\mathcal{S}_1$ and some Z_2 for $\mathcal{S}_2$ that satisfy the decomposition property. Now we set $Z = Z_1 \cup Z_2$. Then for any f that decomposes into f_1 and f_2 w.r.t. Y, by the induction hypothesis, we have that $[\![\mathcal{S}_2]\!](f)$ decomposes into $[\![\textsc{Sli}(\mathcal{S}_2)]\!](f_1)$ and $[\![\textsc{Aux}(\mathcal{S}_2)]\!](f_2)$ w.r.t. $Y \cup Z_2$. Thus again by the induction hypothesis, we have that $[\![\mathcal{S}_1]\!]([\![\mathcal{S}_2]\!](f))$ decomposes into $[\![\textsc{Sli}(\mathcal{S}_1)]\!]([\![\textsc{Sli}(\mathcal{S}_2)]\!](f_1))$ and $[\![\textsc{Aux}(\mathcal{S}_1)]\!]([\![\textsc{Aux}(\mathcal{S}_2)]\!](f_2))$ w.r.t. $Y \cup Z_1 \cup Z_2$. We are done because

– $[\![\textsc{Sli}(\mathcal{S}_1; \mathcal{S}_2)]\!](f_1) = [\![\textsc{Sli}(\mathcal{S}_1)]\!]([\![\textsc{Sli}(\mathcal{S}_2)]\!](f_1))$; and
– $[\![\textsc{Aux}(\mathcal{S}_1; \mathcal{S}_2)]\!](f_2) = [\![\textsc{Aux}(\mathcal{S}_1)]\!]([\![\textsc{Aux}(\mathcal{S}_2)]\!](f_2))$.

• When $\mathcal{S}$ is $\texttt{if}\ x\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2$.
By the induction hypothesis, we have Z_1 for $\mathcal{S}_1$ and Z_2 for $\mathcal{S}_2$ with the decomposition property. Now we set $Z = Z_1 \cup Z_2$. Then for any f that decomposes into f_1 and f_2 w.r.t. Y, by the induction hypothesis we have that $[\![\mathcal{S}_1]\!](f)$ decomposes into $[\![\textsc{Sli}(\mathcal{S}_1)]\!](f_1)$ and $[\![\textsc{Aux}(\mathcal{S}_1)]\!](f_2)$ w.r.t. $Y \cup Z_1$ and $[\![\mathcal{S}_2]\!](f)$ decomposes into $[\![\textsc{Sli}(\mathcal{S}_2)]\!](f_1)$ and $[\![\textsc{Aux}(\mathcal{S}_2)]\!](f_2)$ w.r.t. $Y \cup Z_2$. One can easily show that for any σ

$$\begin{aligned}
&[\![\textsc{Aux}(\texttt{if}\ x\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2)]\!](f_2)(\sigma) \\
&= [\![\textsc{Aux}(\texttt{if}\ x\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2)]\!](f_2)(\sigma|_{Y \cup Z})\ .
\end{aligned}$$

Now we show that for any σ

$$\begin{aligned}
&[\![\texttt{if}\ x\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2]\!](f)(\sigma) \\
&= [\![\textsc{Sli}(\texttt{if}\ x\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2)]\!](f_1)(\sigma|_{\mathcal{X}}) \\
&\quad \times [\![\textsc{Aux}(\texttt{if}\ x\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2)]\!](f_2)(\sigma|_{Y \cup Z})\ .
\end{aligned}$$

If $x \in \mathcal{X}$, then we have $\textsc{Aux}(\mathcal{S}_1) = \textsc{Aux}(\mathcal{S}_2) = \texttt{skip}$ by Lemma 3 because $\textsc{Dep}(\mathcal{S}_1)(C \cup \{x\}) \cup \textsc{Dep}(\mathcal{S}_2)(C \cup \{x\}) \subseteq \mathcal{G}$. By case analysis on $\sigma(x) = \texttt{true}$ or not, we can easily show the goal. If $x \notin \mathcal{X}$, then we have $\textsc{Sli}(\mathcal{S}_1) = \textsc{Sli}(\mathcal{S}_2) = \texttt{skip}$ by Lemma 2. By case analysis on whether $\sigma(x) = \texttt{true}$ or not, we can easily show the goal.

• When $\mathcal{S}$ is $\texttt{while}\ x\ \texttt{do}\ \mathcal{S}_0$.
By the induction hypothesis, we have Z_0 for $\mathcal{S}_0$ with the decomposition property. We set $Z = Z_0$ if $x \in \mathcal{X}$; $Z = Z_0 \cup \{x\}$ otherwise. We have $\textsc{Dinf}(\mathcal{G})(Z) \cap \mathcal{X} = \emptyset$ because when $x \notin \mathcal{X}$ we have $\textsc{Dinf}(\mathcal{G})(x) \cap \mathcal{X} = \emptyset$ by Lemma 1 since $x \in \mathcal{O}$.

Then for any f that decomposes into f_1 and f_2 w.r.t. Y, we show $[\![\texttt{while}\ x\ \texttt{do}\ \mathcal{S}_0]\!](f)$ decomposes into $[\![\textsc{Sli}(\texttt{while}\ x\ \texttt{do}\ \mathcal{S}_0)]\!](f_1)$ and $[\![\textsc{Aux}(\texttt{while}\ x\ \texttt{do}\ \mathcal{S}_0)]\!](f_2)$ w.r.t. $Y \cup Z$. For this, we need to show, for any σ,

(1) $[\![\textsc{Aux}(\texttt{while}\ x\ \texttt{do}\ \mathcal{S}_0)]\!](f_2)(\sigma) = [\![\textsc{Aux}(\texttt{while}\ x\ \texttt{do}\ \mathcal{S}_0)]\!](f_2)(\sigma|_{Y \cup Z})$
(2) $[\![\texttt{while}\ x\ \texttt{do}\ \mathcal{S}_0]\!](f)(\sigma) =$
$[\![\textsc{Sli}(\texttt{while}\ x\ \texttt{do}\ \mathcal{S}_0)]\!](f_1)(\sigma|_{\mathcal{X}}) \times [\![\textsc{Aux}(\texttt{while}\ x\ \texttt{do}\ \mathcal{S}_0)]\!](f_2)(\sigma|_{Y \cup Z})$

Suppose $x \in \mathcal{X}$. Then we have $\textsc{Aux}(\mathcal{S}_0) = \texttt{skip}$ by Lemma 3. (1) holds clearly and for (2) it suffices to show for all n and σ

$$\begin{aligned}
&[\![\texttt{while}\ x\ \texttt{do}_n\ \mathcal{S}_0]\!](f)(\sigma) \\
&= [\![\texttt{while}\ x\ \texttt{do}_n\ \textsc{Sli}(\mathcal{S}_0)]\!](f_1)(\sigma|_{\mathcal{X}}) \times f_2(\sigma|_{Y \cup Z})
\end{aligned}$$

We prove this by induction on n. It holds obviously for the base case. For the step case, we need to show

$$\begin{aligned}
&[\![\texttt{if}\ x\ \texttt{then}\ \mathcal{S}_0; \texttt{while}\ x\ \texttt{do}_n\ \mathcal{S}_0\ \texttt{else}\ \texttt{skip}]\!](f)(\sigma) \\
&= [\![\texttt{if}\ x\ \texttt{then}\ \textsc{Sli}(\mathcal{S}_0); \texttt{while}\ x\ \texttt{do}_n\ \textsc{Sli}(\mathcal{S}_0)\ \texttt{else}\ \texttt{skip}]\!](f_1)(\sigma|_{\mathcal{X}}) \\
&\quad \times f_2(\sigma|_{Y \cup Z})
\end{aligned}$$

By the induction hypothesis, we have that $[\![\texttt{while}\ x\ \texttt{do}_n\ \mathcal{S}_0]\!](f)$ decomposes into $[\![\texttt{while}\ x\ \texttt{do}_n\ \textsc{Sli}(\mathcal{S}_0)]\!](f_1)$ and f_2 w.r.t. $Y \cup Z$. Since $[\![\mathcal{S}_0]\!]$ decomposes into $[\![\textsc{Sli}(\mathcal{S}_0)]\!]$ and $[\![\texttt{skip}]\!]$ w.r.t. Z_0, we have that $[\![\mathcal{S}_0; \texttt{while}\ x\ \texttt{do}_n\ \mathcal{S}_0]\!](f)$ decomposes into

$$[\![\textsc{Sli}(\mathcal{S}_0); \texttt{while}\ x\ \texttt{do}_n\ \textsc{Sli}(\mathcal{S}_0)]\!](f_1) \text{ and } f_2$$

w.r.t. $Y \cup Z$. From this one can easily show that

$$[\![\texttt{if}\ x\ \texttt{then}\ \mathcal{S}_0; \texttt{while}\ x\ \texttt{do}_n\ \mathcal{S}_0\ \texttt{else}\ \texttt{skip}]\!](f)$$

decomposes into

$$[\![\texttt{if}\ x\ \texttt{then}\ \textsc{Sli}(\mathcal{S}_0); \texttt{while}\ x\ \texttt{do}_n\ \textsc{Sli}(\mathcal{S}_0)\ \texttt{else}\ \texttt{skip}]\!](f_1) \text{ and } f_2$$

w.r.t. $Y \cup Z$ because $x \in \mathcal{X}$. Thus we are done.

Now suppose $x \notin \mathcal{X}$. Then we have $\textsc{Dinf}(\mathcal{G})(x) \cap \mathcal{X} = \emptyset$ by Lemma 1 since $x \in \mathcal{O}$. We also have $\textsc{Sli}(\mathcal{S}_0) = \texttt{skip}$ by

Lemma 2. (1) and (2) follows from showing, for all n and σ,

$$\begin{aligned}&[\![\texttt{while}\, x\, \texttt{do}_n\, \textsc{Aux}(\mathcal{S}_0)]\!](f_2)(\sigma)\\ =\ &[\![\texttt{while}\, x\, \texttt{do}_n\, \textsc{Aux}(\mathcal{S}_0)]\!](f_2)(\sigma|_{Y\cup Z})\end{aligned}$$

and

$$\begin{aligned}&[\![\texttt{while}\, x\, \texttt{do}_n\, \mathcal{S}_0]\!](f)(\sigma)\\ =\ &f_1(\sigma|_{\mathcal{X}}) \times [\![\texttt{while}\, x\, \texttt{do}_n\, \textsc{Aux}(\mathcal{S}_0)]\!](f_2)(\sigma|_{Y\cup Z})\ .\end{aligned}$$

We prove this by induction on n. It holds obviously for the base case. For the step case, we need to show

$[\![\texttt{if}\, x\, \texttt{then}\, \textsc{Aux}(\mathcal{S}_0); \texttt{while}\, x\, \texttt{do}_n\, \textsc{Aux}(\mathcal{S}_0)\, \texttt{else skip}]\!](f_2)(\sigma) =$
$[\![\texttt{if}\, x\, \texttt{then}\, \textsc{Aux}(\mathcal{S}_0); \texttt{while}\, x\, \texttt{do}_n\, \textsc{Aux}(\mathcal{S}_0)\, \texttt{else skip}]\!](f_2)(\sigma|_{Y\cup Z})$

and

$[\![\texttt{if}\, x\, \texttt{then}\, \mathcal{S}_0; \texttt{while}\, x\, \texttt{do}_n\, \mathcal{S}_0\, \texttt{else skip}]\!](f)(\sigma) = f_1(\sigma|_{\mathcal{X}}) \times$
$[\![\texttt{if}\, x\, \texttt{then}\, \textsc{Aux}(\mathcal{S}_0); \texttt{while}\, x\, \texttt{do}_n\, \textsc{Aux}(\mathcal{S}_0)\, \texttt{else skip}]\!](f_2)(\sigma|_{Y\cup Z})$.

By the induction hypothesis, we have that $[\![\texttt{while}\, x\, \texttt{do}_n\, \mathcal{S}_0]\!](f)$ decomposes into f_1 and $[\![\texttt{while}\, x\, \texttt{do}_n\, \textsc{Aux}(\mathcal{S}_0)]\!](f_2)$ w.r.t. $Y \cup Z$. Since $[\![\mathcal{S}_0]\!]$ decomposes into $[\![\texttt{skip}]\!]$ and $[\![\textsc{Aux}(\mathcal{S}_0)]\!]$ w.r.t. Z_0, we have that $[\![\mathcal{S}_0; \texttt{while}\, x\, \texttt{do}_n\, \mathcal{S}_0]\!](f)$ decomposes into f_1 and $[\![\textsc{Aux}(\mathcal{S}_0); \texttt{while}\, x\, \texttt{do}_n\, \textsc{Aux}(\mathcal{S}_0)]\!](f_2)$ w.r.t. $Y \cup Z$. From this one can easily show $[\![\texttt{if}\, x\, \texttt{then}\, \mathcal{S}_0; \texttt{while}\, x\, \texttt{do}_n\, \mathcal{S}_0\, \texttt{else skip}]\!](f)$ decomposes into

f_1 and $[\![\texttt{if}\, x\, \texttt{then}\, \textsc{Aux}(\mathcal{S}_0); \texttt{while}\, x\, \texttt{do}_n\, \textsc{Aux}(\mathcal{S}_0)\, \texttt{else skip}]\!](f_2)$

w.r.t. $Y \cup Z$ because $x \in Z$. Thus we are done. ■

THEOREM 1. *For a probabilistic program $\mathcal{P} = \mathcal{S}\ \texttt{return}\ \mathcal{E}$ with $[\![\mathcal{S}]\!](\lambda\sigma.1)(\bot) \neq 0$, we have that $\mathcal{P}$ and $\textsc{Sli}(\mathcal{P})$ are semantically equivalent, i.e.,*

$$[\![\mathcal{S}\ \texttt{return}\ \mathcal{E}]\!] = [\![\textsc{Sli}(\mathcal{S})(\mathcal{X})\ \texttt{return}\ \mathcal{E}]\!]$$

for $\mathcal{X} = \textsc{Inf}(\textsc{OVar}(\mathcal{S}), \textsc{Dep}(\mathcal{S})(\emptyset))(\textsc{Fv}(\mathcal{E}))$.

Proof: Let f be an arbitrary function of type $\mathbb{R} \to [0,1]$. Since $\lambda\sigma.\, f(\sigma(\mathcal{E}))$ decomposes into $\lambda\sigma.\, f(\sigma(\mathcal{E}))$ and $\lambda\sigma.1$ w.r.t. $\emptyset$, and $\lambda\sigma.1$ decomposes into $\lambda\sigma.1$ and $\lambda\sigma.1$ w.r.t. $\emptyset$, we have, by Lemma 4:

$[\![\mathcal{S}]\!](\lambda\sigma.\, f(\sigma(\mathcal{E})))(\bot) = [\![\textsc{Sli}(\mathcal{S})]\!](\lambda\sigma.\, f(\sigma(\mathcal{E})))(\bot) \times [\![\textsc{Aux}(\mathcal{S})]\!](\lambda\sigma.1)(\bot)$
$[\![\mathcal{S}]\!](\lambda\sigma.1)(\bot) = [\![\textsc{Sli}(\mathcal{S})]\!](\lambda\sigma.1)(\bot) \times [\![\textsc{Aux}(\mathcal{S})]\!](\lambda\sigma.1)(\bot)\ .$

Since $[\![\mathcal{S}]\!](\lambda\sigma.1)(\bot) \neq 0$, we have $[\![\textsc{Sli}(\mathcal{S})]\!](\lambda\sigma.1)(\bot) \neq 0$ and $[\![\textsc{Aux}(\mathcal{S})]\!](\lambda\sigma.1)(\bot) \neq 0$. Thus we have

$$\begin{aligned}[\![\mathcal{P}]\!](f) &= \frac{[\![\mathcal{S}]\!](\lambda\sigma.\, f(\sigma(\mathcal{E})))(\bot)}{[\![\mathcal{S}]\!](\lambda\sigma.1)(\bot)}\\ &= \frac{[\![\textsc{Sli}(\mathcal{S})]\!](\lambda\sigma.\, f(\sigma(\mathcal{E})))(\bot) \times [\![\textsc{Aux}(\mathcal{S})]\!](\lambda\sigma.1)(\bot)}{[\![\textsc{Sli}(\mathcal{S})]\!](\lambda\sigma.1)(\bot) \times [\![\textsc{Aux}(\mathcal{S})]\!](\lambda\sigma.1)(\bot)}\\ &= \frac{[\![\textsc{Sli}(\mathcal{S})]\!](\lambda\sigma.\, f(\sigma(\mathcal{E})))(\bot)}{[\![\textsc{Sli}(\mathcal{S})]\!](\lambda\sigma.1)(\bot)}\\ &= [\![\textsc{Sli}(\mathcal{S})(\mathcal{X})\ \texttt{return}\ (\mathcal{E})]\!](f)\end{aligned}$$

■

6. Evaluation

We have implemented the SLI transformation as a source-to-source transformation in the R2 probabilistic programming system [25]. The goal of our evaluation is to measure the improvement in inference time due to the SLI transformation. All experiments were performed on a 2.00 GHz Intel 3rd Gen Core i7 processor system with 8 GB RAM running Microsoft Windows 8.

Benchmarks. Table 1 shows the benchmarks we used for the evaluation. The first two benchmarks are the two motivating examples from Section 2. The third and fourth benchmarks are small examples taken from papers [17, 27]. The last four benchmarks—Bayesian Linear Regression, HIV, Chess and Halo—are larger, and are routinely used to test scalability of techniques in the Bayesian inference community.

Name	Description	Slicing criterion (R: Return, O: Observe)
Ex3	Example 3 in Figure 2	**R**: variable s, **O**: unchanged
Ex5	Example 5 in Figure 4(a)	**R**: variable l, **O**: unchanged
Noisy OR	Given a DAG, each node is a noisy-or of its parents [17]	**R**: subset of nodes in the DAG, **O**: unchanged
Burglar Alarm	Burglary model, having observed an alarm, earthquake etc. [27]	**R**: event corresponding to whether a person wakes up, **O**: unchanged
Bayesian Linear Regression	Linear regression via Bayesian inference (on 1000 points) [23].	**R**: unchanged, **O**: subset of data points (100 points)
HIV	A multi-level linear model with interaction and varying slope and intercept (for 369 measurements and 84 persons) [15]	**R** HIV levels for 10 persons, **O**: unchanged
Chess	Skill rating system for a Chess tournament consisting of 77 players and 2926 games [14]	**R**: skills of 3 particular players, **O**: unchanged
Halo	Skill rating system for a tournament consisting of 31 teams, at most 4 players per team [14]	**R**: skills of 4 particular players, **O**: unchanged

Table 1. Benchmark programs and slicing criteria. "Unchanged" in the slicing criteria means that the set of return variables or observations for the respective benchmark is unchanged.

The return variables of the program and the set of observations (columns 2 and 3 of Table 1) implicitly specify the slicing criterion. For instance, for the HIV model, we arbitrarily picked and returned the HIV levels of 10 persons (as opposed to all 84 persons) and the SLI transformation is applied with respect to this criterion, and retained all the observations from [15], which are measurements of immunity levels for all the persons. Similarly, for the other examples, we picked arbitrarily a subset of variables as return variables (and retained the observations present in these models). By changing the return variables and observed data, we can experiment with other slicing criteria. Our results did not change significantly with particular choices of return variables (we tried several sets at random), so we just present our results for one choice.

Tools. We experimented with our own system [25], which uses an imperative language for specifying probabilistic models and performs inference using a combination of program analysis and MCMC sampling [21]. We also experimented with two other systems whose implementations are available:

- Church [11]: A probabilistic programming system based on Scheme that uses MCMC based inference.
- Infer.NET [23]: A .NET library for expressing graphical models and performing inference using variants of the belief propagation algorithm.

Results. Figure 18 shows the improvement in performance of inference on all the benchmarks. The y-axis shows the speedup in inference time due to slicing (shown on a logarithmic scale). The SLI transformation uniformly and significantly improves the performance of all inference engines over all benchmarks. That the improvement is seen not only in our own inference tool R2, but also in other tools Church and Infer.NET (which use very different techniques for inference) shows the robustness of the SLI transformation.

It is important to reiterate that the observe dependence is crucial for this performance improvement—as illustrated in Section 2. All the observe statements in the bigger benchmarks are control dependent on the return statement. Thus, if we had to preserve terminating behaviors, the constructed slices will have to include all the observe statements. Instead, our notion of observe dependence was able to slice away significantly a larger portion of the program.

In addition to measuring runtimes for inference, we also measured the rate at which the inference converges for these examples. Since R2 uses sampling (the same is true for Church as well), we

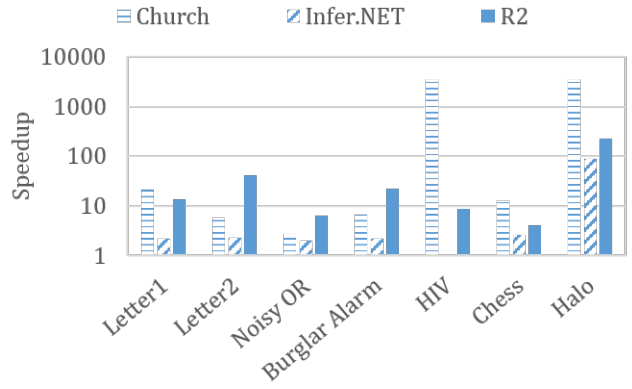


Figure 18. Inference speedup due to the SLI transformation—the y-axis shows the speedup in inference time (on a logarithmic scale) due to slicing. Since Church does not support the Gamma distribution, the corresponding column for Bayesian Linear Regression is not present. For the HIV and Halo benchmarks, Church does not terminate on the original programs, but terminates on the sliced programs.

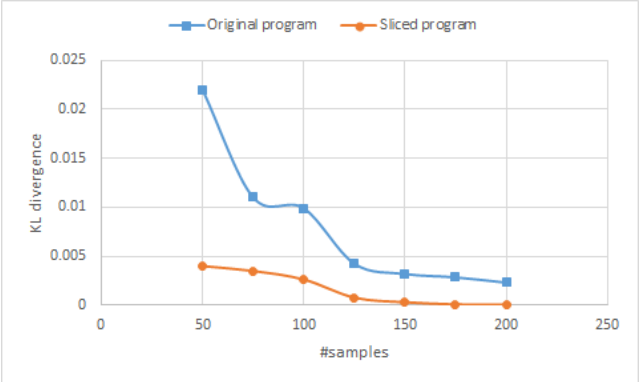


Figure 19. Illustration of convergence rate of inference (for the R2 tool) over the sliced and original versions of Burglar Alarm.

can plot the KL divergence (a distance metric between two distributions [6]) between the estimated answer and actual answer, versus the number of samples. Figure 19 shows such a plot for the Burglar Alarm benchmark on R2. It can be seen from the figure that the inference over the sliced program converges faster than inference over the original program. We also observe similar behavior for the other benchmarks as well.

7. Related work

Our work is inspired by the notion of influence characterized by active trails [19] in Bayesian networks as well as traditional program slicing [9]. In contrast to non-termination preserving slicing techniques [13], the SLI transformation does not preserve non-termination behaviors of the program. Instead, SLI preserves the normalized probability distribution of the output over terminating behaviors which is crucial for slicing probabilistic programs.

There has been a significant interest in the development of probabilistic programming systems [2, 10–12, 18, 20, 23, 26]. Inference for programs in these systems can be done using static analysis techniques [5, 22, 28] such as abstract interpretation and data flow analysis, or dynamic analysis techniques [4, 11] such as MCMC sampling algorithms [21]. Our slicing algorithm provides value to both static and dynamic inference techniques. Our main motivation for this paper is to define the SLI transformation so that it can be used as a subroutine by existing inference engines to improve the efficiency of inference.

Query sensitive inference has been explored in the case of Markov Logic Networks (which are special instances of probabilistic programs). Recent work employs counterexample-guided abstraction refinement (CEGAR) during inference to simplify the model in a query sensitive manner [3]. However, it is interesting to note that existing inference engines (in particular, Church and Infer.NET) do not seem to exploit the knowledge of the query in order to efficiently slice the probabilistic model, as shown by our experimental results.

8. Concluding Remarks

We have identified a new notion of dependence called observe dependence in probabilistic programs, and combined it with traditional notions of control and data dependences to define a slicing transformation SLI. We have shown that SLI is semantics preserving. Our empirical results also demonstrate that it greatly improves the performance of inference by slicing away irrelevant parts before doing inference, when we are interested in only a subset of the the variables r as output (or return values) of the program.

A probabilistic program P typically encodes a set of observations from real world data. We can use the notation $P = C(D)$ to denote a program P which has a code template C and data D (which is read from a file or a database). Suppose we are interested in only a subset r of the variables of the program, and return these variables from the program. It is interesting to ask if we can construct a slice $\text{SLI}(P) = C'(D')$ with respect to the return variables r, where C' is some transformation of C and $D' \subseteq D$. Such a slicing operator will be of great use to practitioners who routinely process different data sets with a fixed probabilistic model and ask the same query for all data sets. We call this problem *probabilistic data slicing* and plan to explore it in future work.

References

[1] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.

[2] J. Borgström, A. D. Gordon, M. Greenberg, J. Margetson, and J. Van Gael. Measure transformer semantics for Bayesian machine learning. In *European Symposium on Programming (ESOP)*, pages 77–96, 2011.

[3] A. Chaganty, A. Lal, A. V. Nori, and S. K. Rajamani. Combining relational learning with smt solvers using CEGAR. In *Computer Aided Verification (CAV)*, pages 447–462, 2013.

[4] A. T. Chaganty, A. V. Nori, and S. K. Rajamani. Efficiently sampling probabilistic programs via program analysis. In *Artificial Intelligence and Statistics (AISTATS)*, 2013.

[5] G. Claret, S. K. Rajamani, A. V. Nori, A. D. Gordon, and J. Borgström. Bayesian inference for probabilistic programs via symbolic execution. In *Foundations of Software Engineering (FSE)*, 2013.

[6] T. M. Cover and J. Thomas. *Elements of Information Theory*. Wiley, 1991.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Principles of Programming Languages (POPL)*, pages 25–35, 1989.

[8] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. *Compaq SRC Research Report 159*, 1998.

[9] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *Transactions on Programming Languages and Systems (TOPLAS).*, 9(3):319–349, 1987.

[10] W. R. Gilks, A. Thomas, and D. J. Spiegelhalter. A language and program for complex Bayesian modelling. *The Statistician*, 43(1):169–177, 1994.

[11] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *Uncertainty in Artificial Intelligence (UAI)*, pages 220–229, 2008.

[12] A. D. Gordon, M. Aizatulin, J. Borgström, G. Claret, T. Graepel, A. V. Nori, S. K. Rajamani, and C. Russo. A model-learner pattern for Bayesian reasoning. In *Principles of Programming Languages (POPL)*, pages 403–416, 2013.

[13] J. Hatcliff, J. C. Corbett, M. B. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with jvm concurrency primitives. In *Static Analysis Symposium (SAS)*, pages 1–18, 1999.

[14] R. Herbrich, T. Minka, and T. Graepel. TrueSkill: A Bayesian skill rating system. In *Neural Information Processing Systems (NIPS)*, pages 569–576, 2006.

[15] M. D. Hoffman and A. Gelman. The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, in press, 2013.

[16] S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[17] O. Kiselyov and C. Shan. Monolingual probabilistic programming using generalized coroutines. In *Uncertainty in Artificial Intelligence (UAI)*, pages 285–292, 2009.

[18] S. Kok, M. Sumner, M. Richardson, P. Singla, H. Poon, D. Lowd, and P. Domingos. The Alchemy system for statistical relational AI, 2007. http://alchemy.cs.washington.edu.

[19] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[20] D. Koller, D. A. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *National Conference on Artificial Intelligence (AAAI)*, pages 740–747, 1997.

[21] D. J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, New York, NY, USA, 2002.

[22] P. Mardziel, S. Magill, M. Hicks, and M. Srivatsa. Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation. *Journal of Computer Security*, January 2013.

[23] T. Minka, J. Winn, J. Guiver, and A. Kannan. Infer.NET 2.3, Nov. 2009. Software available from http://research.microsoft.com/infernet.

[24] G. Nelson. A generalization of Dijkstra's calculus. *Transactions on Programming Languages and Systems (TOPLAS).*, 11(4):517–561, 1989.

[25] A. V. Nori, C.-K. Hur, S. K. Rajamani, and S. Samuel. The R2 project. 2014. http://research.microsoft.com/r2.

[26] A. Pfeffer. The design and implementation of IBAL: A general-purpose probabilistic language. In L. Getoor and B. Taskar, editors, *Introduction to Statistical Relational Learning*. MIT Press, 2007.

[27] J. Pfeffer. *Probabilistic Reasoning in Intelligence Systems*. Morgan Kaufmann, 1996.

[28] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis of probabilistic programs: Inferring whole program properties from finitely many executions. In *Programming Languages Design and Implementation (PLDI)*, 2013.

[29] M. Weiser. Program slicing. In *International Conference on Software Engineering (ICSE)*, pages 439–449, 1981.