Slicing Probabilistic Programs

Matthias Volk

RWTH Aachen University, Germany
Matthias.Volk@rwth-aachen.de

Abstract. Probabilistic programs extend the declaration of programs by use of probabilities. Here the return value of a program is not a concrete value but a probability distribution. For efficiently analyzing these probabilistic programs the program should be made as small as possible by slicing it. Here only parts which influence the return value are kept whereas unnecessary parts are cut away.
In the given paper "Slicing Probabilistic Programs" [1] the authors Hur et al. extend the usual slicing for non-probabilistic programs to probabilistic ones. As the control and data dependence as used in the non-probabilistic setting are not enough to ensure correctness of the slicing, they introduce the concept of observe dependence which handles observe statements in the probabilistic program. Using these three notions of dependences altogether ensures a correct and efficient slicing of such probabilistic programs.

1 Introduction

Probabilistic programs extend the usual notion of programs, e.g. C or Java, by also allowing probabilistic assignments where variable values are drawn from probability distributions. Further on, variable values can be conditioned to certain values by observe statements. Probabilistic programs are widely used in statistics and machine learning, e. g., speech recognition, biology. An important problem in the domain of probabilistic programs is the so called *probabilistic inference* where the probability distribution of the returned expression should be determined (or approximated). As the examined programs can be complex a good way to reduce complexity would be to reduce the original program to a sliced program where only those parts which influence the result remain. Parts which do not influence the outcome of the program are eliminated. For non-probabilistic programs slicing as defined in [2] is well-understood. For the probabilistic setting the problem of slicing is investigated in [1] and reviewed in the present report. Here it does not suffice to apply the usual slicing where control and data dependence are considered. As we will see in certain examples in Section 3 applying the usual slicing is neither correct nor results in the smallest possible sliced program. Therefore a third dependence, the so called *observe dependence*, must be introduced. This dependence arises specifically from the observe statements which can condition the variables to certain values. Only when considering the influences arising from this statements we obtain a slicing transformation Sli, which is both correct and efficient.

We start by giving an introduction about probabilistic programs in Section 2. In Section 3 we will give an intuitive view about the slicing for probabilistic programs and the problems which arise there. Using the knowledge gained there we give a slicing transformation in Section 4. We end by giving a short evaluation in Section 5 and a conclusion in Section 6.

2 Preliminaries

We first introduce the concept of probabilistic programs by means of the probabilistic programming language `PROB`. This programming language is an imperative programming language similar to Dijkstra's Guarded Command Language (GCL) [3] which is enriched by probabilistic commands.

`PROB` makes use of the "normal" syntax of imperative programming language, but introduces two more commands:

1. The **probabilistic assignment** `x` $\sim$ `Dist(`$\bar{\theta}$`)`. Here a sample is drawn from distribution `Dist(`$\bar{\theta}$`)`, where $\bar{\theta}$ is a vector containing the distribution parameters. This sample is then assigned to variable `x`. An example for this is the statement `x` $\sim$ `Gaussian(`μ`,` σ^2`)` which draws a value from the Gaussian distribution with mean μ and variance σ^2. This value is then assigned to variable `x`.
2. The **observe statement** `observe(`φ`)` declares a condition φ which must be fulfilled by all valid execution of the program. An example for this is the statement `observe(x = false)` which means, that in every valid run the variable `x` must have the value `false` when this command is executed. Notice that the total probability of all valid runs can be less than 1. Therefore the probabilities of all valid runs are rescaled to sum up to 1. Intuitively the observe statement has the effect of considering a conditional distribution and is very similar to the `assume` statement used in program verification (even though the `assume` statement produces no probability re-scaling).

Before giving a formal definition of the probabilistic programming language `PROB` we illustrate the use of the language by means of some examples.

Example 1 (Coin tossing).

```
1 bool c1, c2;
2 c1 = Bernoulli(0.5);
3 c2 = Bernoulli(0.5);
4 return c1 + c2;
```

Fig. 1: Coin tossing.

This example as seen in Fig. 1 models the tossing of two fair coins by drawing from a Bernoulli distribution with mean 0.5. The resulting values are assigned to variables `c1` and `c2` respectively. The return value is the sum of both variables `c1` and `c2`. The semantics of the program is the expectation of its return value. Since we have a fair coin it is $Pr(c1 = false, c2 = false) = Pr(c1 = false, c2 = true) = Pr(c1 = true, c2 = false) = Pr(c1 = true, c2 = true) = \frac{1}{4}$. By setting $false = 0$ and $true = 1$ we obtain the expected return value $\frac{1}{4} \cdot (0+0) + \frac{1}{4} \cdot (0+1) + \frac{1}{4} \cdot (1+0) + \frac{1}{4} \cdot (1+1) = \frac{1}{4} + \frac{1}{4} + \frac{1}{4} \cdot 2 = 1$.

Example 2 (Coin tossing with counting).
This example as shown in Fig. 2 extends the previous example by introducing the variable `count` which stores the number of coin tosses resulting in the value `true`. Here the program has the expected return value $Pr(c1 = false, c2 = false) \cdot 0 + Pr(c1 = true, c2 = false) \cdot 1 + Pr(c1 = false, c2 = true) \cdot 1 + Pr(c1 = true, c2 = true) \cdot 2 = \frac{1}{4} \cdot 0 + \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 = 1$.

Example 3 (Coin tossing with counting and observe).
This example as seen in Fig. 3 in comparison to the previous one has an additional observe statement. This ensures that at least one of the coin tosses has to result in the value `true`. Execution runs not fulfilling this condition are blocked whereas the probabilities of all valid runs are rescaled to sum up to 1, i. e., they are normalized. Therefore it is $Pr(c1 = false, c2 = false) = 0$ as this run is blocked. Further on the remaining possibilities get new probabilities as $Pr(c1 = true, c2 = false) = Pr(c1 = false, c2 = true) = Pr(c1 = true, c2 = true) = \frac{1}{3}$. This leads to an expected return value of $0 \cdot 0 + \frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 2 = \frac{4}{3}$.

Notice that the statement `observe(x)` can also be written as the equivalent statement `while(!x) skip`. This is due to the fact that non-terminating runs are ignored in probabilistic programs and the semantics is normalized w. r. t. the terminating runs.

```
bool c1, c2;
int count = 0;
c1 = Bernoulli(0.5);
if (c1) then
    count = count + 1;
c2 = Bernoulli(0.5);
if (c2) then
    count = count + 1;

return(count);
```

Fig. 2: Coin tossing with counting.

```
bool c1, c2;
int count = 0;
c1 = Bernoulli(0.5);
if (c1) then
    count = count + 1;
c2 = Bernoulli(0.5);
if (c2) then
    count = count + 1;
observe(c1 || c2);
return(count);
```

Fig. 3: Coin tossing with counting and observe.

2.1 Formal definition of PROB

We will now give the formal definition of the probabilistic programming language PROB. We start by introducing the syntax of PROB.

Definition 1 (Syntax of PROB).

x	$\in Vars$	*Variables*
uop	$::= \ldots$	*C unary operators*
bop	$::= \ldots$	*C binary operators*
φ, ψ	$::= \ldots$	*logical formula*
$\mathcal{E}$	$::=$	*expressions*
	$\mid x$	*variable*
	$\mid c$	*constant*
	$\mid$ uop $\mathcal{E}$	*unary operation*
	$\mid \mathcal{E}_1$ bop $\mathcal{E}_2$	*binary operation*
$\mathcal{S}$	$::=$	*statements*
	$\mid$ skip	*skip*
	$\mid x = \mathcal{E}$	*deterministic assignment*
	$\mid x \sim$ 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*

As mentioned before the syntax is very similar to a classic imperative programming language, e. g., C. A program consists of statements and a return expression. Statements include skips, deterministic and probabilistic assignments and observes as well as conditionals and loops. Statements can be combined in a sequential manner. Notice that additional "syntactic sugar", e. g., functions or arrays, can be defined in a straight-forward way but are not included in this core language.

We now introduce the semantics of PROB.

Definition 2 (Unnormalized semantics for statements).
$[\![S]\!] \in (\Sigma \to [0,1]) \to \Sigma \to [0,1]$

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

A state $\sigma \in \Sigma$ of a program is a (partial) valuation of all its variables as $\sigma : \mathit{Vars} \to \mathit{Val}$. This can be lifted to expressions as $\sigma : \mathit{Exprs} \to \mathit{Val}$. By having default values for uninitialized variables we make this lifting a total function. We now consider a probabilistic statement $\mathcal{S}$. Here the denotational semantics $[\![\mathcal{S}]\!](f)(\sigma)$ gives the expected return value of a program with statement $\mathcal{S}$, return expression f and initial state σ. The `skip` statement does not change the input state σ. Next the deterministic assignment first changes the state σ by applying the assignment and then applying f. The probabilistic assignment samples v from the distribution $\texttt{Dist}(\bar{\theta})$, executes the assignment with v as right-hand side value and then applies f. By integrating over all possible values of v we obtain the expected value. The `observe` statement behaves like a `skip` statement if the expression φ is evaluated to `true`. If this is not the case the returned value is 0 indicating that this run is not probable. The sequential and conditional statements are defined as expected. Last, the `while` loop has standard fixpoint semantics. Notice that up to this point, the semantics for the statements are unnormalized. But due to non-terminating runs and observe statements we must normalize this as seen next.

Definition 3 (Normalized semantics for programs).
$[\![\mathcal{S} \texttt{ return } \mathcal{E}]\!] \in (\mathcal{R} \to [0,1]) \to [0,1]$

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

where $\bot$ denotes the empty state.

As stated in the definition $\bot$ denotes the empty state where the default values are assigned to every variable. It is important for the definition to work that the behavior of the program is independent of the initial state.

3 Slicing

3.1 The usual slicing

First we introduce the usual notion of slicing. When analyzing a (probabilistic) program $\mathcal{P}$ we are interested in its return expression r. But instead of analyzing the complete program it would be better

to only consider those parts of the program relevant to the return expression. We therefore slice $\mathcal{P}$ and elude all those parts not influencing r. As a result we obtain a smaller sliced program $\textsc{Sli}(\mathcal{P})$. It is important that the Sli transformation is both correct and efficient. Correctness means, that the probability distribution for r is the same for $\mathcal{P}$ and $\textsc{Sli}(\mathcal{P})$. Efficient means, that the computation of the Sli transformation should be fast and the sliced program should be small so that r can be computed fast in $\textsc{Sli}(\mathcal{P})$.

We start by giving an intuition of program slicing by means of some examples. In the first example the usual definition of slicing [2] works for a probabilistic program as well.

Example 4 (Starting example).

```
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;
```

Fig. 4: Starting example.

```
1  bool i, s;
2  i = Bernoulli(0.7);
3  if (!i)
4      s = Bernoulli(0.2);
5  else
6      s = Bernoulli(0.95);
7  return s;
```

Fig. 5: Sliced program of Fig. 4.

This example as seen in Fig. 4 is adapted from [4] and represents a model for a reference letter `l` for a student. This reference letter depends on the course difficulty `d`, the student intelligence `i`, the course grade `g` and the SAT score `s`. The dependency graph looks as depicted in Fig. 6. Edges in the dependency graph depict control or data dependences.

As the variable `s` is returned by the program, we can see with help of the dependency graph, that `s` only depends on `i`. A first intuitive idea would be to keep only those parts of the original program where these two variables occur and slice away parts where `d`, `g` and `l` occur. This idea is similar to the one used in regular non-probabilistic program slicing, where one performs a sort of back-trace from the return value to compute control and data dependences. As a result we obtain the sliced program of Fig. 5.

We see that the size of this sliced program is only one third of the size of the original program and therefore allows an easier computation of the resulting probability distribution.

Now the question that arises is whether the usual slicing always works in the probabilistic setting as well. We can show that this is not the case with the following example.

Example 5 (Usual slicing is incorrect). The example in Fig. 7 is a variation of Example 4 where only the observe statement in Line 20 is added.

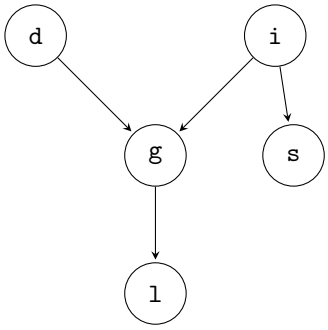

Fig. 6: Dependency graph for Fig. 4.

```
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;
```

Fig. 7: Usual slicing is incorrect.

Performing the usual slicing we would obtain exactly the same result as in Example 4 (see Fig. 5). The observe statement would be sliced as only the unnecessary variable `l` occurs here. But this result is not correct as the original program and the sliced one are not equivalent. The reason is that the observe statement introduces new dependencies which are not accounted for in the ordinary slicing technique. The observation of `l` influences the value of `g` which then indirectly influences `i` and finally leads to an influence on the resulting value of `s`. Further on a second "path of influence" flows from `d` to `i` and therefore to `s`. In the end, all variables are important for the final result and therefore the sliced program is exactly the same as the original program. We will have a closer look on the influence of observe statements later on.

Next we consider another example where the usual slicing is correct but not optimal as it does not result in the smallest possible program.

Example 6 (Usual slicing is inefficient). This example in Fig. 8 is a variation of the previous example in Fig. 4, but now the returned variable is `l` and we introduced an observe statement in Line 12 constraining the value of `g` to `false`.

As the returned variable is `l` the transitive dependences include `g`, `i` and `d`, but the variable `s` is not needed anymore. This results in the sliced program as stated in Fig. 9.

But having a closer look on the observe statement in Line 12 leads to an even smaller sliced program. As `g` is constrained to `false` we can stop traversing dependences at `g`. The values of `i` and `d` do not influence the resulting value of `g` as it is constrained later on. We obtain a smaller sliced program as seen in Fig. 10.

Further slicing this program leads to the final result in Fig. 11.

To conclude the examples we analyze the slicing of loopy programs.

Example 7 (Program with loops).

As seen in Fig. 12, `c` is repeatedly sampled and `b` is toggled until `c` becomes `false`. A simplified dependency graph is show in Fig. 13.

```
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;
```

Fig. 8: Usual slicing is inefficient.

```
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);
13 if (!g)
14     l = Bernoulli(0.1);
15 else
16     l = Bernoulli(0.4);
17 return l;
```

Fig. 9: Sliced program of Fig. 8.

```
1  bool l, g;
2  g = false;
3  if (!g)
4      l = Bernoulli(0.1);
5  else
6      l = Bernoulli(0.4);
7  return l;
```

Fig. 10: Further sliced program of Fig. 8.

```
1  bool l;
2  l = Bernoulli(0.1);
3  return l;
```

Fig. 11: Further sliced program of Fig. 10.

As the variable `b` is observed every part influencing `b` must be contained in the resulting sliced program. As `b` is changed in every iteration this part can not be left out and in the end the sliced program is the original program.

3.2 The observe dependence

As we have seen so far the usual definition of slicing guarantees neither an optimal nor a correct slicing. This is because the observe statements play a role which is not considered in the usual slicing.

Therefore the usual control and data dependences as captured by the relation DInf are not enough for probabilistic slicing. Thus, we extend the usual slicing technique by introducing the concept of observe dependences. We extend the relation DInf to a relation called *influencers* and denoted Inf. This captures the additional observe dependence as well. (Note that Inf $\supseteq$ DInf).

We explain the intuition behind this observe dependence with the following example.

Example 8 (Observe dependence). Consider the example in Fig. 14 derived from Example 4 by introducing the observe statement in Line 20. The corresponding dependency graph is depicted in Fig. 15.

```
1 bool x, b, c;
2 x = Bernoulli(0.5);
3 b = x;
4 c = Bernoulli(0.5);
5 while (c) do
6     b = !b;
7     c = Bernoulli(0.5);
8 observe(b = false);
9 return  x;
```

Fig. 12: Program with loops.

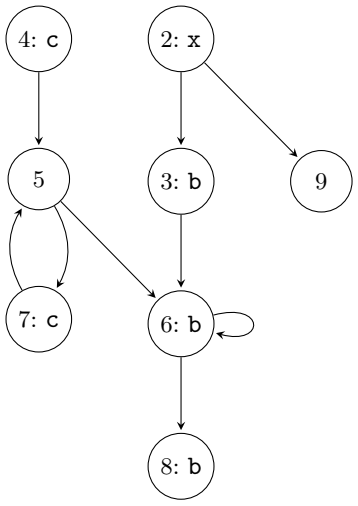

Fig. 13: Dependency graph for Fig. 12.

We have an observe statement `observe(g = true)` and two variables `d` and `i` such that `g` depends on both of them by the usual notion of control or data dependence, i. e., `d`, `i` $\in$ DInf(`g`). The return variable of the program is `s` and `s` depends on `i`, i. e., `i` $\in$ Inf(`s`).

There exists a path for influence to flow from `d` through `g` and `i` to `s`, i. e., `d` $\in$ Inf(`s`). By observing `g` we know its value and the knowledge about `d` influences the knowledge about `i` and vice versa. By knowing `g` and having knowledge about `d` we can draw conclusions about `i`. These additional flows of influence rise from the observe dependences. These observe dependences are related to the concept of *active trails* in Bayesian networks [4].

4 Slice transformation

In the previous section we have given an intuitive view on the slice transformation in the probabilistic setting. We now formalize this slice transformation and give an algorithm for computing the sliced program.

4.1 Preprocessing

First we restrict our programs to being in SSA form [5], i. e., each variable is assigned only once in the program. Further on we assume that all predicates in observe statements and while loops are single boolean variables, i. e., they have the form `observe(x)` and `while (y) do` $\mathcal{S}$ with `x`, `y` being boolean variables. To ensure these constraints we perform a preprocessing consisting of three steps where the first step prunes spurious dependences occurring through observe statements.

1. applying the Obs transformation to remove spurious dependences
2. for each predicate in an observe statement, conditional or while loop we introduce a fresh variable holding its value by applying the Svf transformation
3. convert the resulting program into SSA form by applying the Ssa transformation

We now explain these three transformations in greater detail starting with the Obs transformation, which is given in the following definition.

```
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(g = true);
21 return s;
```

Fig. 14: Observe dependence.

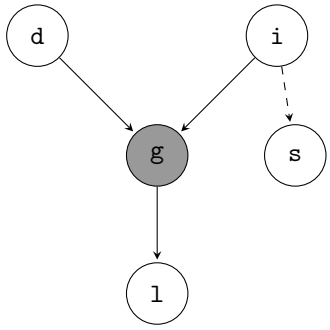

Fig. 15: Dependency graph for Fig. 14.

Definition 4 (OBS transformation).

$$
\begin{aligned}
\textsc{Obs}(\texttt{observe}(\mathcal{E})) &:= \texttt{observe}(\mathcal{E});\ \textsc{ObserveSet}(\mathcal{E}) \\
\textsc{Obs}(\texttt{while}\ \mathcal{E}\ \texttt{do}\ \mathcal{S}) &:= \texttt{while}\ \mathcal{E}\ \texttt{do}\ \textsc{Obs}(\mathcal{S});\ \textsc{WhileSet}(\mathcal{E}) \\
\textsc{Obs}(S_1;\ \mathcal{S}_2) &:= \textsc{Obs}(\mathcal{S}_1);\ \textsc{Obs}(\mathcal{S}_2) \\
\textsc{Obs}(\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2) &:= \texttt{if}\ \mathcal{E}\ \texttt{then}\ \textsc{Obs}(\mathcal{S}_1)\ \texttt{else}\ \textsc{Obs}(\mathcal{S}_2) \\
\textsc{Obs}(\mathcal{S}) &:= \mathcal{S},\ \textit{otherwise}
\end{aligned}
$$

with

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

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

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

The intuition behind this Obs transformation is that after observe statements and while loops we can in some cases add an assignment which states the value of the variables upon the exit of these statements. After an observe statement which is an equality between a program variable and a closed expression, e. g., $\texttt{observe}(x = \mathcal{E})$, we add this equality as an assignment, e. g., $x = \mathcal{E}$. After the guard of a loop which is an inequality between a program variable and a closed expression, e. g., $\texttt{while}(x \neq \mathcal{E})$, we add the equality as an assignment, e. g., $x = \mathcal{E}$. We therefore can stop traversing certain dependences when the value of certain variables are know. As an example consider the previous example as seen in Fig. 8 where the observe statements in Line 12 ensures `g = false`. We then stopped traversing the dependences as the previous variable assignments were not important anymore. By introducing

assignments after every observe statement and while loop we ensure that these variable values are considered while computing the dependences.

The second transformation step is the Svf transformation into *single variable form* which is defined as follows.

Definition 5 (SVF transformation).

$$
\begin{aligned}
\textsc{Svf}(\texttt{observe}(\mathcal{E})) &:= \textbf{let } x' \in \mathit{freshvar}() \textbf{ in } x' = \mathcal{E};\ \texttt{observe}(x') \\
\textsc{Svf}(\texttt{while } \mathcal{E} \texttt{ do } \mathcal{S}) &:= \textbf{let } x' \in \mathit{freshvar}() \textbf{ in } x' = \mathcal{E};\ \texttt{while } x' \texttt{ do } (\mathcal{S};\ x' = \mathcal{E}) \\
\textsc{Svf}(S_1;\ \mathcal{S}_2) &:= \textsc{Svf}(\mathcal{S}_1);\ \textsc{Svf}(\mathcal{S}_2) \\
\textsc{Svf}(\texttt{if } \mathcal{E} \texttt{ then } \mathcal{S}_1 \texttt{ else } \mathcal{S}_2) &:= \textbf{let } x' \in \mathit{freshvar}() \textbf{ in } x' = \mathcal{E};\ \texttt{if } x' \texttt{ then } \textsc{Svf}(\mathcal{S}_1) \texttt{ else } \textsc{Svf}(\mathcal{S}_2) \\
\textsc{Svf}(\mathcal{S}) &:= \mathcal{S},\ \mathit{otherwise} \\
\\
\textsc{Svf}(\mathcal{S} \texttt{ return } \mathcal{E}) &:= \textsc{Svf}(\mathcal{S}) \texttt{ return } \mathcal{E}
\end{aligned}
$$

The Svf transformation introduces for every condition in observe, if-then-else or while loops a fresh variable (by calling *freshvar*()) and stores the condition in this variables. Thus we ensure that we only have single boolean variables in there which is important for the correctness proof.

The last transformation is the Ssa transformation which ensures that there is only one assignment per variable.

Definition 6 (SSA transformation). $\textsc{Ssa}(S) \in \mathbb{P}(\mathit{Vars}) \times \mathit{Ren} \to \mathbb{P}(\mathit{Vars}) \times \mathit{Ren} \times \mathit{Statement\ with\ Ren} = \mathit{Vars} \to \mathit{Vars}$

$$
\begin{aligned}
\textsc{Ssa}(\texttt{skip})(X,\rho) &:= (X, \rho, \texttt{skip}) \\
\textsc{Ssa}(\texttt{observe}(\mathcal{E}))(X,\rho) &:= (X, \rho, \texttt{observe}(\rho(\mathcal{E}))) \\
\textsc{Ssa}(x = \mathcal{E})(X,\rho) &:= \textbf{let } x' \notin X \textbf{ in } (X \cup \{x'\}, \rho[x \mapsto x'], x' = \rho(\mathcal{E})) \\
\textsc{Ssa}(x \sim \texttt{Dist}(\bar{\theta}))(X,\rho) &:= \textbf{let } x' \notin X \textbf{ in } (X \cup \{x'\}, \rho[x \mapsto x'], x' \sim \texttt{Dist}(\rho(\mathcal{E}))) \\
\textsc{Ssa}(S_1;\ \mathcal{S}_2)(X,\rho) &:= \textbf{let } (X', \rho', S_1') = \textsc{Ssa}(S_1)(X,\rho) \textbf{ and} \\
&\quad\ \textbf{let } (X'', \rho'', S_2') = \textsc{Ssa}(S_2)(X', \rho') \textbf{ in } (X'', \rho'', S_1';\ S_2') \\
\textsc{Ssa}(\texttt{if } \mathcal{E} \texttt{ then } \mathcal{S}_1 \texttt{ else } \mathcal{S}_2)(X,\rho) &:= \textbf{let } (X', \rho', S_1') = \textsc{Ssa}(S_1)(X,\rho) \textbf{ and} \\
&\quad\ \textbf{let } (X'', \rho'', S_2') = \textsc{Ssa}(S_2)(X', \rho') \textbf{ and} \\
&\quad\ \textbf{let } S_2'' = \textsc{Merge}(\rho', \rho'') \\
&\quad\ \textbf{in } (X'', \rho', \texttt{if } \rho(\mathcal{E}) \texttt{ then } S_1' \texttt{ else } (S_2';\ S_2'')) \\
\textsc{Ssa}(\texttt{while } \mathcal{E} \texttt{ do } \mathcal{S})(X,\rho) &:= \textbf{let } (X', \rho', S') = \textsc{Ssa}(S)(X,\rho) \textbf{ and} \\
&\quad\ \textbf{let } S'' = \textsc{Merge}(\rho, \rho') \textbf{ in } (X', \rho, \texttt{while } \rho(\mathcal{E}) \texttt{ do } (S';\ S''))
\end{aligned}
$$

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

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

The above Ssa transformation is a variant of the transformation in [5], where the phi-nodes are not used by relaxing the Ssa condition that there should be only one assignment per variable.

To get a better understand of the preprocessing we give an example.

Example 9 (Preprocessing). We apply the preprocessing to the example from Fig. 8. After applying the first transformation Obs we obtain the result shown in Fig. 16. Only the Line 15 right after the observe statement was added. The next step is applying the Svf transformation which results in Fig. 17. Here the fresh variables `q1, ..., q6` are introduced. The set of observed variables is $\mathcal{O} = \{q4\}$. Finally the result after the Ssa transformation can be seen in Fig. 18.

```
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
7      if (!i && d)
8          g = Bernoulli(0.05);
9      else
10         if (i && !d)
11             g = Bernoulli(0.9);
12         else
13             g = Bernoulli(0.5);
14 observe(g = false);
15 g = false;
16 if (!i)
17     s = Bernoulli(0.2);
18 else
19     s = Bernoulli(0.95);
20 if (!g)
21     l = Bernoulli(0.1);
22 else
23     l = Bernoulli(0.4);
24 return l;
```

Fig. 16: After Obs.

4.2 Main transformation

After the preprocessing step we compute the slicing of the program. To this end, we first compute the dependency graph, which allows identifying all variables that influence the return expression. As illustrated before we do not only need control and data dependence from usual slicing but also our new observe dependence. The whole slicing transformation is defined by the Sli transformation.

First we compute the dependency graph as well as the set of observed variables.

```
bool d, i, s, l, g;
bool q1, q2, q3, q4, q5, q6;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
q1 = (!i && !d);
if (q1)
    g = Bernoulli(0.3);
else
    q2 = (!i && d);
    if (q2)
        g = Bernoulli(0.05);
    else
        q3 = (i && !d);
        if (q3)
            g = Bernoulli(0.9);
        else
            g = Bernoulli(0.5);
q4 = (g = false);
observe(q4);
g = false;
q5 = !i;
if (q5)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
q6 = !g;
if (q6)
    l = Bernoulli(0.1);
else
    l = Bernoulli(0.4);
return l;
```

Fig. 17: After SVF.

```
bool d, i, s, l, g, q1, q2, q3, q4;
bool q5, q6, g1, g2, g3, g4, s1, l1;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
q1 = (!i && !d);
if (q1)
    g = Bernoulli(0.3);
else
    q2 (!i && d);
    if (q2)
        g1 = Bernoulli(0.05);
    else
        q3 = (i && !d);
        if (q3)
            g2 = Bernoulli(0.9);
        else
            g3 = Bernoulli(0.5);
            g2 = g3;
        g1 = g2;
    g = g1;
q4 = (g = false);
observe(q4);
g4 = false;
q5 = !i;
if (q5)
    s = Bernoulli(0.2);
else
    s1 = Bernoulli(0.95);
    s = s1;
q6 = !g4;
if (q6)
    l = Bernoulli(0.1);
else
    l1 = Bernoulli(0.4);
    l = l1;
return l;
```

Fig. 18: After SSA.

Definition 7 (Calculation of observed variables and dependency graph).
$\text{OVAR}(\mathcal{S}) \in \mathbb{P}(\mathit{Vars})$

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

$\textsc{Dep}(\mathcal{S}) \in \mathbb{P}(\mathit{Vars}) \to \mathbb{P}(\mathit{Vars} \times \mathit{Vars})$

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

The observed variables OVar are calculated by structural induction over the statements of the program and accumulating the conditionals of observe statements and while loops. The dependency graph Dep is a binary relation over the variables. It takes the control dependences $\mathcal{C}$ of the current statement as as an argument and calculates the control and data dependences. For example $(x, y) \in \textsc{Dep}(\mathcal{S})(\emptyset)$ means that there is a data or control flow from x to y at some point in $\mathcal{S}$. The calculation of Dep is done by accumulating variables of the guards of if-then-else statements and while loops. In deterministic and probabilistic assignments the data dependence from right-hand side to the left-hand side as well as the control dependence is added. Finally the observe statement accumulates the control dependences.

So far we have only considered the control and data dependences. Now we are interested in the variables that influence the return expression of the program. We call these variables *influencers*. For the return statement $\texttt{return}(\mathcal{E})$ let $\mathcal{R}$ be the set of free variables in $\mathcal{E}$. Now we want to compute the influencers of $\mathcal{R}$ which depend on the program statement $\mathcal{S}$. We distinguish between direct influencers DInf and influencers Inf. The set of direct influencers of $\mathcal{R}$ for dependency graph $\mathcal{G}$ is denoted by $\textsc{DInf}(\mathcal{G})(\mathcal{R})$. It consists of all the variables which can be reached in $\mathcal{G}$ by backward traverse from $\mathcal{R}$. Notice that $\mathcal{G}$ depends on the program statement $\mathcal{S}$, i. e., $\mathcal{G} = \textsc{Dep}(\mathcal{S})(\emptyset)$.

Definition 8 (Influencer calculation).

– *Direct influencers*

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

– *Influencers*

$$\frac{x \in \textsc{DInf}(\mathcal{G})(\mathcal{R})}{x \in \textsc{Inf}(\mathcal{O}, \mathcal{G})(\mathcal{R})} \qquad \frac{x, y \in \textsc{DInf}(\mathcal{G})(\{z\}) \quad z \in \mathcal{O} \quad y \in \textsc{Inf}(\mathcal{O}, \mathcal{G})(\mathcal{R})}{x \in \textsc{Inf}(\mathcal{O}, \mathcal{G})(\mathcal{R})}$$

The first rule of the direct influencers states that every variable in the return expression is part of the direct influencers. The second rule defines the backward traverse in the dependency graph to accumulate all direct influencers. As a result the set $\textsc{DInf}(\mathcal{G})(\mathcal{R})$ consists of all variables that influence the return variables through control and data dependences. But as seen before this is not enough to ensure correctness of the slicing and we have to consider indirect influencers of observe statements as well. Let $\mathcal{O}$ be the set of all observed variables. The set of all influencers is denoted by $\textsc{Inf}(\mathcal{O}, \mathcal{G})(\mathcal{R})$ and is computed by the second part of the influencer calculation. Here the first rule states that every direct influencer is also an influencer. The second rule is more interesting as it captures observe dependences. As seen before in Fig. 15 if we have an observed variable z, y is already an influencer of $\mathcal{R}$ and x and y influence z directly then x is also an influencer of $\mathcal{R}$..

Now we can define the slicing transformation Sli.

Definition 9 (SLI transformation).
$\textsc{Sli}(\mathcal{S}) \in \mathbb{P}(\mathit{Vars}) \to \mathit{Statement}$

$$\textsc{Sli}(\texttt{skip})(X) := \texttt{skip}$$

$$\textsc{Sli}(x = \mathcal{E})(X) := \begin{cases} x = \mathcal{E} & \mathit{if}\ x \in X \\ \texttt{skip} & \mathit{otherwise} \end{cases}$$

$$\textsc{Sli}(x \sim \texttt{Dist}(\bar{\theta}))(X) := \begin{cases} x \sim \texttt{Dist}(\bar{\theta}) & \mathit{if}\ x \in X \\ \texttt{skip} & \mathit{otherwise} \end{cases}$$

$$\textsc{Sli}(\texttt{observe}(x))(X) := \begin{cases} \texttt{observe}(x) & \mathit{if}\ x \in X \\ \texttt{skip} & \mathit{otherwise} \end{cases}$$

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

$$\textsc{Sli}(\texttt{if}\ x\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2)(X) :=$$

$$\begin{cases} \texttt{skip} & \mathit{if}\ \textsc{Sli}(\mathcal{S}_1)(X) = \textsc{Sli}(\mathcal{S}_2)(X) = \texttt{skip} \\ \texttt{if}\ x\ \texttt{then}\ \textsc{Sli}(\mathcal{S}_1)(X)\ \texttt{else}\ \textsc{Sli}(\mathcal{S}_2)(X) & \mathit{otherwise} \end{cases}$$

$$\textsc{Sli}(\texttt{while}\ x\ \texttt{do}\ \mathcal{S})(X) := \begin{cases} \texttt{while}\ x\ \texttt{do}\ \textsc{Sli}(\mathcal{S})(X) & \mathit{if}\ x \in X \\ \texttt{skip} & \mathit{otherwise} \end{cases}$$

$$\textsc{Sli}(\mathcal{S}\ \texttt{return}\ \mathcal{E}) := \textsc{Sli}(\mathcal{S})(\textsc{Inf}(\mathcal{O}, \mathcal{G})(\mathcal{R}))\ \texttt{return}\ \mathcal{E}$$
$$\mathit{where}\ \mathcal{O} = \textsc{OVar}(\mathcal{S}), \mathcal{G} = \textsc{Dep}(\mathcal{S})(\emptyset), \mathcal{R} = \textsc{Fv}(\mathcal{E})$$

The slicing transformation has the set of influencers $\textsc{Inf}(\mathcal{O}, \mathcal{G})(\mathcal{R})$ as input with dependency graph $\mathcal{G}$, the set of observed variables $\mathcal{O}$ and the set of variables in the return statement $\mathcal{R}$. The transformation only keeps those statements whose variables are in the set of influencers and replaces all other unnecessary statements with `skip`.

For a better understanding of the Sli transformation we continue the Example 9 from the preprocessing.

Example 10 (SLI transformation). The slicing results in program given in Fig. 19. As mentioned earlier the conditioning of the variable `g` in the observe statements cuts away great parts of the original program.

```
1 bool l, q6, g4, l1;
2 g4 = false;
3 q6 = !g4;
4 if (q6)
5     l = Bernoulli(0.1);
6 else
7     l1 = Bernoulli(0.4);
8     l = l1;
9 return l;
```

Fig. 19: SLI transformation.

The correctness of the transformation is stated in the following theorem.

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

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

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

Proof. The proof is done by induction on the statement structure. For further details see [1].

5 Evaluation

To show the applicability of the slicing transformation for probabilistic programs this approach with the Sli transformation was implemented as a source-to-source transformation in the R2 probabilistic programming language [6]. Additionally the transformation was implemented in two other tools Church [7] and Infer.NET [8].

For the evaluation several benchmarks were used. In addition to the two examples of Fig. 4 and Fig. 8 two small examples (Noisy OR and Burglar Alarm) and four bigger examples (Bayesian Linear Regression, HIV, Chess and Halo) were taken.

The evaluation compares the inference time, i. e., the time needed for computing the probability distribution of the return expression, on all benchmarks between the original program and the sliced version. The results show that the Sli transformation leads to a speedup for all benchmarks up to factor 10. Especially for the Halo benchmark the speedup actually is over 100. As this improvement can be seen for all three tools this emphasizes that the Sli transformation is robust and not only applicable to R2.

These results show that slicing probabilistic programs improves the inference time and therefore the analysis performance. Unfortunately the authors do not state in the paper how fast the Sli transformation can be computed. This is also an important point for measuring the performance and applicability of the slicing.

6 Conclusion

6.1 Related Work

The work was inspired by the traditional concept of slicing [9] as well as the concept of *influence* as seen by active trails in Bayesian networks [4].

6.2 Conclusion

In [1] the authors extend the concept of slicing of programs to probabilistic programs. Slicing programs leads to smaller programs while keeping their semantics and makes analysis of them easier and faster. The usual concept of control and data dependences is not sufficient to guarantee correctness of the sliced program. Therefore the new notion of observe dependence was introduced. By applying the Sli transformation which makes use of this new dependence it is possible to compute the slice of a probabilistic program.

In the future the problem of *probabilistic data slicing* arises. Here we have a probabilistic program $\mathcal{P} = \mathcal{C}(\mathcal{D})$ with code $\mathcal{C}$ and data $\mathcal{D}$. Now the question is to compute a slice $\textsc{Sli}(\mathcal{P}) = \mathcal{C}'(\mathcal{D}')$ w. r. t. certain returned variables where $\mathcal{C}'$ is a transformation of $\mathcal{C}$ and $\mathcal{D}' \subseteq \mathcal{D}$. Then the slice would be helpful for program where the data changes but the underlying code and the query stay the same.

References

1. Hur, C.K., Nori, A.V., Rajamani, S.K., Samuel, S.: Slicing probabilistic programs. In: Programming Language Design and Implementation (PLDI), ACM (2014)
2. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering. ICSE '81, Piscataway, NJ, USA, IEEE Press (1981) 439–449
3. Dijkstra, E.W.: Guarded commands, non-determinacy and formal derivation of programs. Comm. ACM **18**(8) (1975) 453–457
4. Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. MIT Press (2009)
5. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: An efficient method of computing static single assignment form. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '89, New York, NY, USA, ACM (1989) 25–35
6. Nori, A.V., Hur, C.K., Rajamani, S.K., Samuel, S.: R2: An efficient mcmc sampler for probabilistic programs. In: AAAI Conference on Artificial Intelligence (AAAI), AAAI (2014)
7. Goodman, N.D., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: A language for generative models. In: In UAI. (2008) 220–229
8. Minka, T., Winn, J., Guiver, J., Webster, S., Zaykov, Y., Yangel, B., Spengler, A., Bronskill, J.: Infer.NET 2.6 (2014) Microsoft Research Cambridge. http://research.microsoft.com/infernet.
9. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3) (1987) 319–349