Slicing Probabilistic Programs

Matthias Volk

RWTH Aachen University

February 4, 2015

Seminar Presentation

Contents

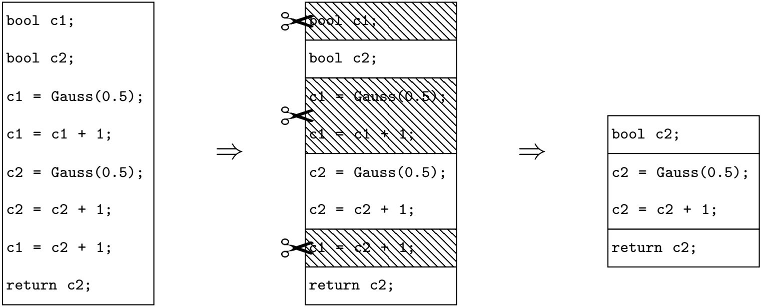
Contents

Introduction

- Based on "Slicing Probabilistic Programs" by Hur et al. [PLDI '14]
- Interested in probabilistic inference:
  determine probability distribution of program output
- For better analysis: reduce size of original program

Idea

Idea: program slicing



```
bool c1;
bool c2;
c1 = Gauss(0.5);
c1 = c1 + 1;
c2 = Gauss(0.5);
c2 = c2 + 1;
c1 = c2 + 1;
return c2;
```

⇒

```
bool c1;
bool c2;
c1 = Gauss(0.5);
c1 = c1 + 1;
c2 = Gauss(0.5);
c2 = c2 + 1;
c1 = c2 + 1;
return c2;
```

⇒

```
bool c2;
c2 = Gauss(0.5);
c2 = c2 + 1;
return c2;
```

⇒ Keep only parts influencing the program output

Goal

Goal: for program $\mathcal{P}$ obtain sliced program $\textsc{Sli}(\mathcal{P})$

Requirements for slice transformation $\textsc{Sli}$:

1. Correct: probability distribution of returned expression equal for $\mathcal{P}$ and $\textsc{Sli}(\mathcal{P})$
2. Efficient: computation of transformation is fast, sliced program is small

Probabilistic programs

Extend ordinary imperative programming language with:

1. probabilistic assignment
   $\texttt{x} \sim \texttt{Dist}(\bar{\theta})$
2. observe statement
   $\texttt{observe}(\varphi)$

Notice:

- Runs not fulfilling `observe` are blocked
- Probabilities of valid runs are rescaled
- `Bool` represented by integer with values 0 and 1

Example (Probabilistic program)

```
bool coin;
coin = Bernoulli(0.5);
observe(coin = 1);
return(coin);
```

Expected return value: 1
as `coin == 0` is blocked

Examples for probabilistic programs

Example (Without `observe`)

```
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);
```

Expected return value:
$\frac{1}{4} \cdot f(0) + \frac{2}{4} \cdot f(1) + \frac{1}{4} \cdot f(2) = 1$

Example (With `observe`)

```
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);
```

Expected return value:
$\frac{2}{3} \cdot f(1) + \frac{1}{3} \cdot f(2) = \frac{4}{3}$

Semantics of probabilistic programs

Definition (Unnormalized semantics for programs)

$[\![\mathcal{S}]\!] \in (\Sigma \to [0,1]) \to \Sigma \to [0,1]$

$$\text{e.g. } [\![x = \mathcal{E}]\!](f)(\sigma) := f(\sigma[x \leftarrow \sigma(\mathcal{E})])$$

where

- f is a return function
- σ is the (partial) valuation of all variables

Definition (Normalized semantics for programs)

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

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

with $\bot$ the empty state where default values are assigned to all variables

Slicing

"Usual" slicing in non-probabilistic programs relies on:

1. Data dependence
2. Control dependence

Questions:

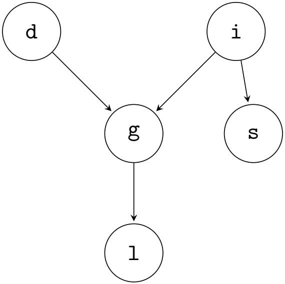
1. Do they suffice for probabilistic programs?
2. If not, what kind of dependence should be considered?
3. If so, do they deliver optimal (i. e., smallest) slicing?

Example introduction

Model for a reference letter `l` for a student which depends on

- Course grade `g`
- Course difficulty `d`
- Student intelligence `i`
- SAT score `s`

The dependency graph:



```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
    g = Bernoulli(0.9);
else
    g = Bernoulli(0.5);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
    l = Bernoulli(0.1);
else
    l = Bernoulli(0.4);
return l;
```

“Usual” slicing works

- Back-trace from returned expression

```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
return s;
```

“Usual” slicing works

- Back-trace from returned expression
- Used variables: `s`

```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
return s;
```

“Usual” slicing works

- Back-trace from returned expression
- Used variables: `s`, `i`

```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
return s;
```

“Usual” slicing works

- Back-trace from returned expression
- Used variables: `s`, `i`

```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
return s;
```

“Usual” slicing works

- Back-trace from returned expression
- Used variables: `s`, `i`
- Slice parts with `d`, `l`, `g`

```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
return s;
```

“Usual” slicing works

```
bool i, s;
d = Bernoulli(0.6);
i = Bernoulli(0.7);




if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);




return s;
```

- Back-trace from returned expression
- Used variables: `s`, `i`
- Slice parts with `d`, `l`, `g`

“Usual” slicing incorrect

- Additional `observe`

```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
observe(l = true)
return s;
```

"Usual" slicing incorrect

- Additional `observe`
- Used variables: `s`

```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
observe(l = true)
return s;
```

“Usual” slicing incorrect

- Additional `observe`
- Used variables: `s`

```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
observe(l = true)
return s;
```

“Usual” slicing incorrect

- Additional `observe`
- Used variables: `s`, `i`

```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
observe(l = true)
return s;
```

“Usual” slicing incorrect

- Additional `observe`
- Used variables: `s`, `i`

```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
observe(l = true)
return s;
```

"Usual" slicing incorrect

- Additional `observe`
- Used variables: `s`, `i`
- Still `l` not relevant for return
  ⇒ Slice

```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
observe(l = true)
return s;
```

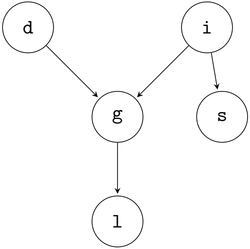
“Usual” slicing incorrect

- Additional `observe`
- Used variables: `s`, `i`
- Still `l` not relevant for return
  ⇒ Slice

```
bool i, s;
d = Bernoulli(0.6);
i = Bernoulli(0.7);




if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);





return s;
```

"Usual" slicing incorrect

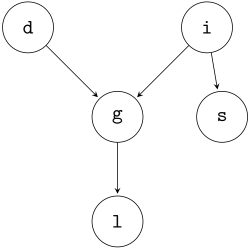
- Additional `observe`
- Used variables: `s`, `i`
- Still `l` not relevant for return
  ⇒ Slice
- Incorrect: `observe(l = true)` introduces new dependencies:



```
bool i, s;
d = Bernoulli(0.6);
i = Bernoulli(0.7);



if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);



return s;
```

"Usual" slicing incorrect

- Additional `observe`
- Used variables: `s`, `i`
- Still `l` not relevant for return
  ⇒ Slice
- Incorrect: `observe(l = true)` introduces new dependencies:



- Later on we analyze why standard slicing does not work here

```
bool i, s;
d = Bernoulli(0.6);
i = Bernoulli(0.7);



if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);



return s;
```

“Usual” slicing inefficient

- Now return `l` and additional `observe`

```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
observe(g = false);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
return l;
```

“Usual” slicing inefficient

- Now return `l` and additional `observe`
- Used variables: `l`

```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
observe(g = false);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
return l;
```

“Usual” slicing inefficient

- Now return `l` and additional `observe`
- Used variables: `l`

```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
observe(g = false);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
return l;
```

“Usual” slicing inefficient

- Now return `l` and additional `observe`
- Used variables: `l`, `g`

```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
observe(g = false);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
return l;
```

“Usual” slicing inefficient

- Now return `l` and additional `observe`
- Used variables: `l`, `g`

```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
observe(g = false);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
return l;
```

“Usual” slicing inefficient

- Now return `l` and additional `observe`
- Used variables: `l`, `g`

```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
observe(g = false);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
return l;
```

“Usual” slicing inefficient

- Now return `l` and additional `observe`
- Used variables: `l`, `g`, `i`, `d`

```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
observe(g = false);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
return l;
```

“Usual” slicing inefficient

- Now return `l` and additional `observe`
- Used variables: `l`, `g`, `i`, `d`

```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
observe(g = false);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
return l;
```

“Usual” slicing inefficient

- Now return `l` and additional `observe`
- Used variables: `l`, `g`, `i`, `d`
- Only variable `s` not needed
  ⇒ Slice

```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
observe(g = false);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
return l;
```

“Usual” slicing inefficient

- Now return `l` and additional `observe`
- Used variables: `l`, `g`, `i`, `d`
- Only variable `s` not needed
  ⇒ Slice

```
bool d, i, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
observe(g = false);




if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
return l;
```

“Usual” slicing inefficient

- Now return `l` and additional `observe`
- Used variables: `l`, `g`, `i`, `d`
- Only variable `s` not needed
  ⇒ Slice
- Inefficient: `g` restricted to `false`
  ⇒ Slice previous parts influencing `g`

```
bool d, i, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
observe(g = false);




if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
return l;
```

"Usual" slicing inefficient

- Now return `l` and additional `observe`
- Used variables: `l`, `g`, `i`, `d`
- Only variable `s` not needed
  ⇒ Slice
- Inefficient: `g` restricted to `false`
  ⇒ Slice previous parts influencing `g`

```
bool l, g;


g = false;


if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
return l;
```

“Usual” slicing inefficient

- Now return `l` and additional `observe`
- Used variables: `l`, `g`, `i`, `d`
- Only variable `s` not needed
  ⇒ Slice
- Inefficient: `g` restricted to `false`
  ⇒ Slice previous parts influencing `g`
  ⇒ Slice subsequent parts on view of assumption `g == true`

```
bool l, g;



g = false;



if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
return l;
```

“Usual” slicing inefficient

- Now return `l` and additional `observe`
- Used variables: `l`, `g`, `i`, `d`
- Only variable `s` not needed
  ⇒ Slice
- Inefficient: `g` restricted to `false`
  ⇒ Slice previous parts influencing `g`
  ⇒ Slice subsequent parts on view of assumption `g == true`

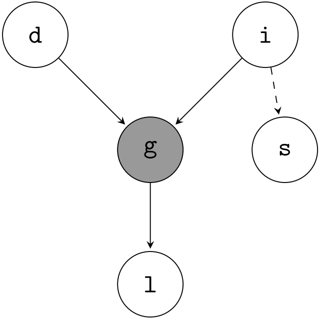
```
bool l;







g = false;







l = Bernoulli(0.1);


return l;
```

Contents

Observe dependence

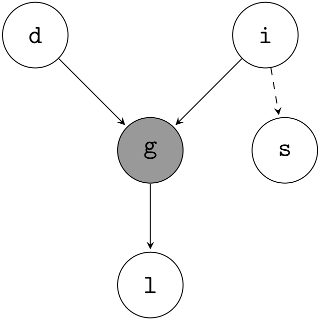
- "Usual" slicing uses control and data dependences (relation $\mathrm{DI}\textsc{nf}$)
- But for probabilistic setting neither correct nor optimal
- `Observe` statements are to blame!
- Solution:
  - Introduce observe dependences
  - Extend $\mathrm{DI}\textsc{nf}$ to relation called influencers ($\mathrm{I}\textsc{nf}$) with $\mathrm{I}\textsc{nf} \supseteq \mathrm{DI}\textsc{nf}$

Example for observe dependence



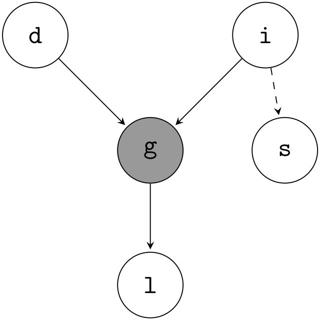
```
bool d, i, s, l, g;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
if (i && !d)
   g = Bernoulli(0.9);
else
   g = Bernoulli(0.5);
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);
if (!g)
   l = Bernoulli(0.1);
else
   l = Bernoulli(0.4);
observe(g = true)
return s;
```

Example for observe dependence



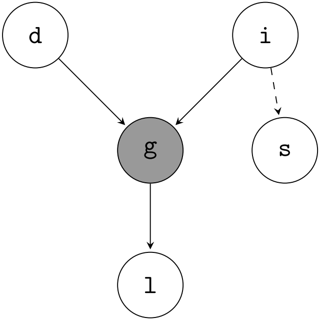
- g observed

Example for observe dependence



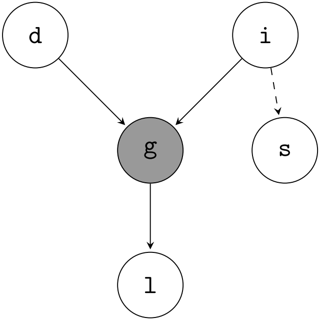
- g observed
- g depends on `d`, `i`
  `d`, `i` $\in \textsc{DInf}(g)$

Example for observe dependence



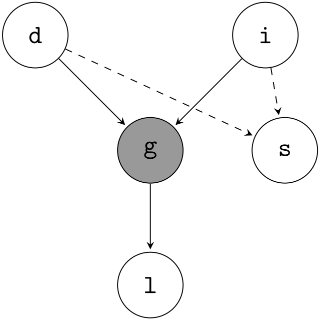
- g observed
- g depends on `d`, `i`
  `d`, `i` $\in \mathrm{DI\textsc{nf}}(\texttt{g})$
- Return variable `s` depends on `i`
  `i` $\in \mathrm{I\textsc{nf}}(\texttt{s})$

Example for observe dependence



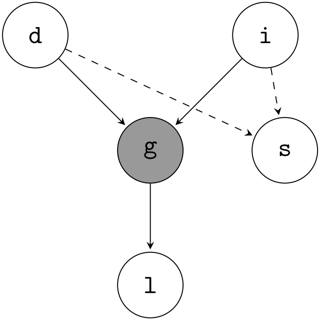
- `g` observed
- `g` depends on `d`, `i`
  `d`, `i` $\in \mathrm{DI}\textsc{nf}(\texttt{g})$
- Return variable `s` depends on `i`
  `i` $\in \mathrm{I}\textsc{nf}(\texttt{s})$
- Path of influence $\texttt{d} \rightarrow \texttt{g} \rightarrow \texttt{i} \rightarrow \texttt{s}$
  `d` $\in \mathrm{I}\textsc{nf}(\texttt{s})$

Example for observe dependence



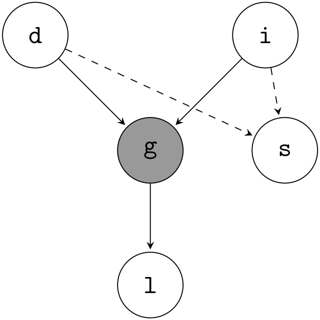
- g observed
- g depends on `d`, `i`
  `d`, `i` $\in \mathrm{DI}\textsc{nf}(\mathtt{g})$
- Return variable `s` depends on `i`
  `i` $\in \mathrm{I}\textsc{nf}(\mathtt{s})$
- Path of influence $\mathtt{d} \rightarrow \mathtt{g} \rightarrow \mathtt{i} \rightarrow \mathtt{s}$
  `d` $\in \mathrm{I}\textsc{nf}(\mathtt{s})$

Example for observe dependence



- g observed
- g depends on `d`, `i`
  `d`, `i` $\in \textrm{DI}\textsc{nf}(\mathtt{g})$
- Return variable `s` depends on `i`
  `i` $\in \textrm{I}\textsc{nf}(\mathtt{s})$
- Path of influence $\mathtt{d} \rightarrow \mathtt{g} \rightarrow \mathtt{i} \rightarrow \mathtt{s}$
  `d` $\in \textrm{I}\textsc{nf}(\mathtt{s})$
- Observe g and knowledge about `d`
  $\Rightarrow$ Draw conclusions about `i`
- And vice versa

Example for observe dependence



- g observed
- g depends on `d`, `i`
  `d`, `i` $\in \mathrm{DI\textsc{nf}}(\mathtt{g})$
- Return variable `s` depends on `i`
  `i` $\in \mathrm{I\textsc{nf}}(\mathtt{s})$
- Path of influence $\mathtt{d} \to \mathtt{g} \to \mathtt{i} \to \mathtt{s}$
  `d` $\in \mathrm{I\textsc{nf}}(\mathtt{s})$
- Observe g and knowledge about `d`
  $\Rightarrow$ Draw conclusions about `i`
- And vice versa
- Notice: Inspired by active trails in Bayesian networks

Algorithm

Preprocessing:

1. Obs transformation
2. Svf transformation
3. Ssa transformation

Algorithm

Preprocessing:

1. OBS transformation
   When possible: after `observe` and `while` add an assignment with the value of variables upon the exit of the instruction
2. SVF transformation
3. SSA transformation

Example (OBS Observe)

```
observe(x = True)
x = True
```

Example (OBS While)

```
while(x != True)
    ...
x = True
```

Algorithm

Preprocessing:

1. OBS transformation

2. SVF transformation
   Single variable form introduces fresh variables for every condition in `observe`, `if-then-else` and `while`
3. SSA transformation

Example (SVF)

Before:

```
observe(x = True)
```

After:

```
q1 = (x = True);
observe(q1);
```

Algorithm

Preprocessing:

1. OBS transformation
2. SVF transformation
3. SSA transformation
   Relaxed single assignment form: every variable is assigned only once

Example (SSA)

Before:

```
if (q1)
    x = True;
else
    if (q2)
        x = False;
    else
        x = True;
```

After:

```
if (q1)
    x = True;
else
    if (q2)
        x1 = False;
    else
        x2 = True;
        x1 = x2;
    x = x1;
```

Algorithm

Main transformation:

1. Calculate observed variables
2. Calculate dependence graph
3. Calculate influencers

Algorithm

Main transformation:

1. Calculate observed variables

   $$\textsc{OVar}(\underset{\underset{\text{program statement}}{\nearrow}}{\mathcal{S}})$$

   Accumulate conditionals of `observe` and `while`
2. Calculate dependence graph
3. Calculate influencers

Example ($\textsc{OVar}$ Observe)

```
observe(x = True)
```

$\Rightarrow \textsc{OVar}(\mathcal{S}) = \{x\}$

Example ($\textsc{OVar}$ While)

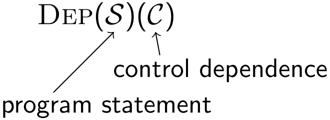
```
while(x != True)
    ...
```

$\Rightarrow \textsc{OVar}(\mathcal{S}) = \{x\}$

Algorithm

Main transformation:

1. Calculate observed variables
2. Calculate dependence graph



$\textsc{Dep}(\mathcal{S})(\mathcal{C})$

control dependence

program statement

Binary relation, calculates control and data dependence

3. Calculate influencers

Example ($\textsc{Dep}$ Assignment)

```
y = x
```

$\Rightarrow \textsc{Dep}(\mathcal{S})(\emptyset) = \{(x, y)\}$

Example ($\textsc{Dep}$ If-Then-Else)

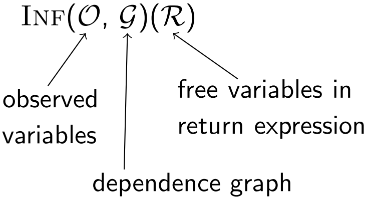
```
if(x != True)
    observe(y = True)
else
    observe(z = True)
```

$\Rightarrow \textsc{Dep}(\mathcal{S})(\emptyset) = \{(x, y), (x, z)\}$

Algorithm

Main transformation:

1. Calculate observed variables
2. Calculate dependence graph
3. Calculate influencers



$\textsc{Inf}(\mathcal{O}, \mathcal{G})(\mathcal{R})$

observed variables

dependence graph

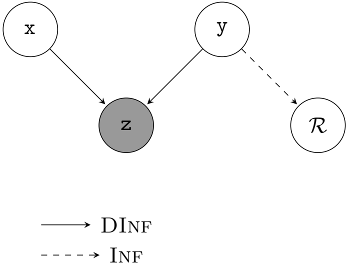
free variables in return expression

Direct influencers $\textsc{DInf}(\mathcal{G})(\mathcal{R})$:

- all variables reachable in $\mathcal{G}$ by backward traverse from $\mathcal{R}$

Influencers $\textsc{Inf}(\mathcal{O}, \mathcal{G})(\mathcal{R})$:

- every direct influencer is an influencer
- observe dependences:



Algorithm

Main transformation:

1. Calculate observed variables
2. Calculate dependence graph
3. Calculate influencers

$\textsc{Inf}(\mathcal{O}, \mathcal{G})(\mathcal{R})$

observed variables

dependence graph

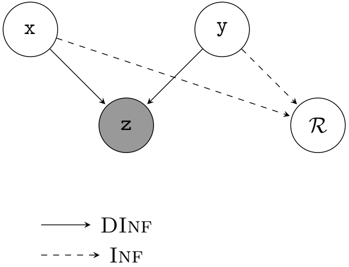
free variables in return expression

Direct influencers $\textsc{DInf}(\mathcal{G})(\mathcal{R})$:

- all variables reachable in $\mathcal{G}$ by backward traverse from $\mathcal{R}$

Influencers $\textsc{Inf}(\mathcal{O}, \mathcal{G})(\mathcal{R})$:

- every direct influencer is an influencer
- observe dependences:



Algorithm

Main transformation:

1. calculate observed variables $\mathcal{O} = \textsc{OVar}(\mathcal{S})$
2. calculate dependence graph $\mathcal{G} = \textsc{Dep}(\mathcal{S})(\emptyset)$
3. calculate influencers $\textsc{Inf}(\mathcal{O}, \mathcal{G})(\mathcal{R})$
   where $\mathcal{R}$= free variables of return expression
4. calculate slicing $\textsc{Sli}$:

$$\textsc{Sli}(\mathcal{S}\ \texttt{return}\ \mathcal{E}) = \textsc{Sli}(\mathcal{S})(\textsc{Inf}(\mathcal{O}, \mathcal{G})(\mathcal{R}))\ \texttt{return}\ \mathcal{E}$$

$\textsc{Sli}$

- keeps only those statements whose variables are in set of influencers
- removes other statements

Algorithm

Main transformation:

1. calculate observed variables $\mathcal{O}= \textsc{OVar}(\mathcal{S})$
2. calculate dependence graph $\mathcal{G}= \textsc{Dep}(\mathcal{S})(\emptyset)$
3. calculate influencers $\textsc{Inf}(\mathcal{O}, \mathcal{G})(\mathcal{R})$
   where $\mathcal{R}=$ free variables of return expression
4. calculate slicing $\textsc{Sli}$:

$$\textsc{Sli}(\mathcal{S}\ \texttt{return}\ \mathcal{E}) = \textsc{Sli}(\mathcal{S})(\textsc{Inf}(\mathcal{O},\mathcal{G})(\mathcal{R}))\ \texttt{return}\ \mathcal{E}$$

   $\textsc{Sli}$

   - keeps only those statements whose variables are in set of influencers
   - removes other statements

Definition ($\textsc{Sli}$ Assignment)

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

Algorithm

Main transformation:

1. calculate observed variables $\mathcal{O}= \textsc{OVar}(\mathcal{S})$
2. calculate dependence graph $\mathcal{G}= \textsc{Dep}(\mathcal{S})(\emptyset)$
3. calculate influencers $\textsc{Inf}(\mathcal{O}, \mathcal{G})(\mathcal{R})$
   where $\mathcal{R}=$ free variables of return expression
4. calculate slicing $\textsc{Sli}$:

$$\textsc{Sli}(\mathcal{S} \texttt{ return } \mathcal{E}) = \textsc{Sli}(\mathcal{S})(\textsc{Inf}(\mathcal{O},\mathcal{G})(\mathcal{R})) \texttt{ return } \mathcal{E}$$

$\textsc{Sli}$

- keeps only those statements whose variables are in set of influencers
- removes other statements

Definition ($\textsc{Sli}$ While)

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

Correctness

Theorem (Correctness of the transformation)

For a probabilistic program $P = \mathcal{S}$ $\texttt{return}$ $\mathcal{E}$ with $[\![\mathcal{S}]\!](\lambda\sigma.1)(\perp) \neq 0$, 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.

By induction on the statement structure (see Hur et al.). ☐

Complete example

Starting example

```
bool d, i, s, l, g;

d = Bernoulli(0.6);
i = Bernoulli(0.7);

if (i && !d)
    g = Bernoulli(0.9);
else
    g = Bernoulli(0.5);


observe(g = false);
```

```
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);


if (!g)
    l = Bernoulli(0.1);
else
    l = Bernoulli(0.4);

return l;
```

Complete example

After OBS

```
bool d, i, s, l, g;

d = Bernoulli(0.6);
i = Bernoulli(0.7);

if (i && !d)
    g = Bernoulli(0.9);
else
    g = Bernoulli(0.5);


observe(g = false);
g = false;
```

```
if (!i)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);


if (!g)
    l = Bernoulli(0.1);
else
    l = Bernoulli(0.4);

return l;
```

Complete example

After SVF

```
bool d, i, s, l, g, q1, q2;
bool q3, q4;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
q1 = (i && !d);
if (q1)
    g = Bernoulli(0.9);
else
    g = Bernoulli(0.5);

q2 = (g = false);
observe(q2);
g = false;
```

```
q3 = !i;
if (q3)
    s = Bernoulli(0.2);
else
    s = Bernoulli(0.95);

q4 = !g;
if (q4)
    l = Bernoulli(0.1);
else
    l = Bernoulli(0.4);

return l;
```

Complete example

After Ssa

```
bool d, i, s, l, g, q1, q2;
bool q3, q4, g1, g2, s1, l1;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
q1 = (i && !d);
if (q1)
    g = Bernoulli(0.9);
else
    g1 = Bernoulli(0.5);
    g = g1;
q2 = (g = false);
observe(q2);
g2 = false;
```

```
q3 = !i;
if (q3)
    s = Bernoulli(0.2);
else
    s1 = Bernoulli(0.95);
    s = s1;
q4 = !g2;
if (q4)
    l = Bernoulli(0.1);
else
    l1 = Bernoulli(0.4);
    l = l1;
return l;
```

Complete example

$$\textsc{OVar}(\mathsf{S}) = \{\mathsf{q2}\}$$

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

Complete example

DEP(S) not depicted

```
bool d, i, s, l, g, q1, q2;
bool q3, q4, g1, g2, s1, l1;
d = Bernoulli(0.6);
i = Bernoulli(0.7);
q1 = (i && !d);
if (q1)
    g = Bernoulli(0.9);
else
    g1 = Bernoulli(0.5);
    g = g1;
q2 = (g = false);
observe(q2);
g2 = false;
```

```
q3 = !i;
if (q3)
    s = Bernoulli(0.2);
else
    s1 = Bernoulli(0.95);
    s = s1;
q4 = !g2;
if (q4)
    l = Bernoulli(0.1);
else
    l1 = Bernoulli(0.4);
    l = l1;
return l;
```

Complete example

$$\textsc{Inf}(\{\overset{\textsc{OVar}(\mathsf{S})}{\mathsf{q2}}\}, \overset{\text{dependence graph}}{\mathcal{G}})(\{\overset{\mathcal{R}}{\mathsf{l}}\}) = \{\mathsf{l}, \mathsf{l1}, \mathsf{q4}, \mathsf{g2}\}$$

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

Complete example

After **Sli**

```
bool l;
bool q4, g2, l1;
```

```
g2 = false;
```

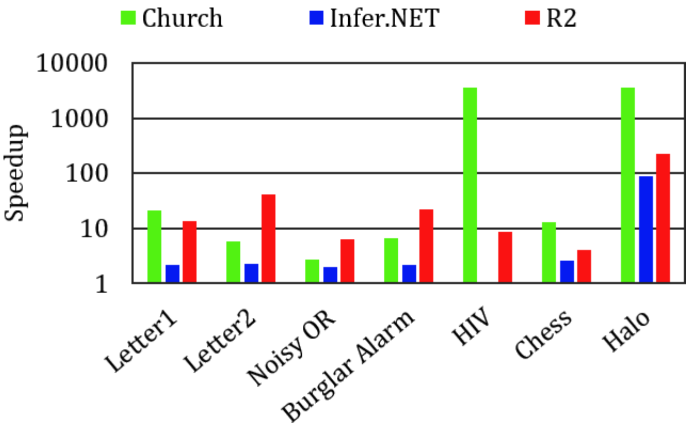
```
q4 = !g2;
if (q4)
    l = Bernoulli(0.1);
else
    l1 = Bernoulli(0.4);
    l = l1;
return l;
```

Contents

Evaluation

- Implemented as source-to-source transformation in R2 probabilistic programming language
- Additionally implemented in Church and Infer.NET
- Examples:
    - Example from slides
    - Noisy OR and Burglar Alarm (small)
    - Bayesian Linear Regression, HIV, Chess, Halo (bigger)
- Compare inference time between original and sliced program

Evaluation



- Speedup for all three tools
- General applicability, not only for R2
- But: No indication for computation time of SLI in paper

Contents

Conclusion

- Slicing programs leads to smaller programs
  $\Rightarrow$ Easier and faster to analyze
- "Usual" concept of control and data dependence not sufficient
  $\Rightarrow$ Introduce observe dependence
- $\textsc{Sli}$ transformation slices probabilistic programs using observe dependences

Future work: Probabilistic data slicing

- Probabilistic program $\mathcal{P} = \mathcal{C}(\mathcal{D})$ with code $\mathcal{C}$ and data $\mathcal{D}$
- Task: compute slice $\textsc{Sli}(\mathcal{P}) = \mathcal{C}'(\mathcal{D}')$ w. r. t. to returned variables with:
  - $\mathcal{C}'$ is transformation of $\mathcal{C}$
  - $\mathcal{D}' \subseteq \mathcal{D}$
  - Paper considered only closed programs, i. e., no input given
- Helpful when data changes but not underlying code and query

Syntax

Definition (Syntax of PROB)

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

Semantics

Definition (Unnormalized semantics for statements)

$[\![S]\!] \in (\Sigma \to [0,1]) \to \Sigma \to [0,1]$

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

where

$$\begin{aligned}
\texttt{while } \mathcal{E} \texttt{ do}_0\ \mathcal{S} &= \texttt{observe}(\texttt{false}) \\
\texttt{while } \mathcal{E} \texttt{ do}_{n+1}\ \mathcal{S} &= \texttt{if } \mathcal{E} \texttt{ then } (\mathcal{S};\ \texttt{while } \mathcal{E} \texttt{ do}_n\ \mathcal{S}) \texttt{ else } (\texttt{skip})
\end{aligned}$$

Obs transformation

Definition (Obs transformation)

$$
\begin{aligned}
\textsc{Obs}(\texttt{observe}(\mathcal{E})) &:= \texttt{observe}(\mathcal{E});\ \textsc{ObserveSet}(\mathcal{E})\\
\textsc{Obs}(\texttt{while}\ \mathcal{E}\ \texttt{do}\ \mathcal{S}) &:= \texttt{while}\ \mathcal{E}\ \texttt{do}\ \textsc{Obs}(\mathcal{S});\ \textsc{WhileSet}(\mathcal{E})\\
\textsc{Obs}(\mathcal{S}_1;\ \mathcal{S}_2) &:= \textsc{Obs}(\mathcal{S}_1);\ \textsc{Obs}(\mathcal{S}_2)\\
\textsc{Obs}(\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2) &:= \texttt{if}\ \mathcal{E}\ \texttt{then}\ \textsc{Obs}(\mathcal{S}_1)\ \texttt{else}\ \textsc{Obs}(\mathcal{S}_2)\\
\textsc{Obs}(\mathcal{S}) &:= \mathcal{S},\ \text{otherwise}\\
\text{with}&\\
\textsc{ObserveSet}(\mathcal{E}) &:= \begin{cases} x = \mathcal{E}' & \text{if } \mathcal{E} \text{ is } (x = \mathcal{E}') \text{ or } (\mathcal{E}' = x),\ \mathcal{E}' \text{ has no vars}\\ \texttt{skip} & \text{otherwise}\end{cases}\\
\textsc{WhileSet}(\mathcal{E}) &:= \begin{cases} x = \mathcal{E}' & \text{if } \mathcal{E} \text{ is } (x \neq \mathcal{E}') \text{ or } (\mathcal{E}' \neq x),\ \mathcal{E}' \text{ has no vars}\\ \texttt{skip} & \text{otherwise}\end{cases}
\end{aligned}
$$

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

$\textsc{Svf}$ transformation

Definition ($\textsc{Svf}$ transformation)

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

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

$\textsc{Ssa}$ transformation

Definition ($\textsc{Ssa}$ transformation)

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

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

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

SSA transformation

Definition (SSA transformation continued)

$$
\begin{aligned}
\textsc{Ssa}(\texttt{if } \mathcal{E} \texttt{ then } \mathcal{S}_1 \texttt{ else } \mathcal{S}_2)(X,\rho) := {} & \textbf{let } (X', \rho', S_1') = \textsc{Ssa}(S_1)(X,\rho) \textbf{ and}\\
& \textbf{let } (X'', \rho'', S_2') = \textsc{Ssa}(S_2)(X',\rho') \textbf{ and}\\
& \textbf{let } S_2'' = \textsc{Merge}(\rho',\rho'')\\
& \textbf{in } (X'', \rho', \texttt{if } \rho(\mathcal{E}) \texttt{ then } S_1' \texttt{ else } (S_2';\ S_2''))\\
\textsc{Ssa}(\texttt{while } \mathcal{E} \texttt{ do } \mathcal{S})(X,\rho) := {} & \textbf{let } (X',\rho',S') = \textsc{Ssa}(S)(X,\rho) \textbf{ and}\\
& \textbf{let } S'' = \textsc{Merge}(\rho,\rho') \textbf{ in}\\
& \qquad (X', \rho, \texttt{while } \rho(\mathcal{E}) \texttt{ do } (S';\ S''))
\end{aligned}
$$

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

Observed variables calculation

Definition ($\textsc{OVar}$)

$\textsc{OVar}(\mathcal{S}) \in \mathbb{P}(\mathit{Vars})$

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

Dependency graph calculation

Definition ($\textsc{Dep}$)

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

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

Influencer calculation

Definition (Inf)

- 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

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

$\textsc{Sli}$ transformation

Definition ($\textsc{Sli}$ transformation)

$\textsc{Sli}(\mathcal{S}) \in \mathbb{P}(\mathit{Vars}) \to \text{Statement}$

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

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

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

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

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

$\textsc{Sli}$ transformation

Definition ($\textsc{Sli}$ transformation continued)

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

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

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

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