Probabilistic Program Analysis

A Probabilistic Language and its Semantics

Alessandra Di Pierro
University of Verona, Italy
alessandra.dipierro@univr.it

Herbert Wiklicky
Imperial College London, UK
herbert@doc.ic.ac.uk

Why probabilistic analysis

- Analysis of probabilistic programs
- Obtaining probabilistic answers from the analysis of deterministic programs
- Compiler optimization via data speculative optimization.

Why probabilistic analysis

- Analysis of probabilistic programs
- Obtaining probabilistic answers from the analysis of deterministic programs
- Compiler optimization via data speculative optimization.

Why probabilistic analysis

- Analysis of probabilistic programs
- Obtaining probabilistic answers from the analysis of deterministic programs
- Compiler optimization via data speculative optimization.

Probabilistic Analysis

By introducing probability we are able to perform:
Probabilistic program analysis and probabilistic program analysis.

Analysis of probabilistic programs

- May give 'incorrect' answers.

Probabilistic analysis of (deterministic) programs

- Speculative vs conservative answers.

Probabilistic Analysis

By introducing probability we are able to perform:

Probabilistic program analysis and probabilistic program analysis.

Analysis of probabilistic programs

- May give 'incorrect' answers.

Probabilistic analysis of (deterministic) programs

- Speculative vs conservative answers.

Probabilistic Analysis

By introducing probability we are able to perform:
Probabilistic program analysis and probabilistic program analysis.

Analysis of probabilistic programs

- May give 'incorrect' answers.

Probabilistic analysis of (deterministic) programs

- Speculative vs conservative answers.

Probabilistic Analysis

By introducing probability we are able to perform:
Probabilistic program analysis and probabilistic program analysis.

Analysis of probabilistic programs

- May give ‘incorrect’ answers.

Probabilistic analysis of (deterministic) programs

- Speculative vs conservative answers.

Probabilistic Analysis

By introducing probability we are able to perform:
Probabilistic program analysis and probabilistic program analysis.

Analysis of probabilistic programs

- May give ‘incorrect’ answers.

Probabilistic analysis of (deterministic) programs

- Speculative vs conservative answers.

Probabilistic Analysis

By introducing probability we are able to perform:
Probabilistic program analysis and probabilistic program analysis.

Analysis of probabilistic programs

- May give ‘incorrect’ answers.

Probabilistic analysis of (deterministic) programs

- Speculative vs conservative answers.

A simple example

The two deterministic programs below compute the factorial $n!$ and the double factorial $2 \cdot n!$

```
m := 1;
while (n>1) do
  m := m*n;
 n := n-1;
od
```

```
m := 2;
while (n>1) do
  m := m*n;
  n := n-1;
od
```

A simple example

The two deterministic programs below compute the factorial $n!$ and the double factorial $2 \cdot n!$

```
m := 1;
while (n>1) do
  m := m*n;
 n := n-1;
od
```

```
m := 2;
while (n>1) do
  m := m*n;
  n := n-1;
od
```

Classical vs Probabilistic Results

Parity Analysis: Determine at every program point whether a variable is *even* or *odd*.

A safe classical analysis will detect (starting with m and n "unknown")

- that $m = 2 \times n!$ at the end of the second program is always *even*;
- that the parity of m is "unknown" at the end of the first program.

However, it is obvious that $m = n!$ is "nearly always" *even*.

The purpose of a probabilistic analysis is a formal derivation of this intuition about the parity of m when the program terminates.

Classical vs Probabilistic Results

Parity Analysis: Determine at every program point whether a variable is *even* or *odd*.

A safe classical analysis will detect (starting with m and n "unknown")

- that $m = 2 \times n!$ at the end of the second program is always *even*;
- that the parity of m is "unknown" at the end of the first program.

However, it is obvious that $m = n!$ is "nearly always" *even*.

The purpose of a probabilistic analysis is a formal derivation of this intuition about the parity of m when the program terminates.

Classical vs Probabilistic Results

Parity Analysis: Determine at every program point whether a variable is *even* or *odd*.
A safe classical analysis will detect (starting with m and n "unknown")

- that $m = 2 \times n!$ at the end of the second program is always *even*;
- that the parity of m is "unknown" at the end of the first program.

However, it is obvious that $m = n!$ is "nearly always" *even*.
The purpose of a probabilistic analysis is a formal derivation of this intuition about the parity of m when the program terminates.

Classical vs Probabilistic Results

Parity Analysis: Determine at every program point whether a variable is *even* or *odd*.
A safe classical analysis will detect (starting with m and n “unknown")

- that $m = 2 \times n!$ at the end of the second program is always *even*;
- that the parity of m is “unknown” at the end of the first program.

However, it is obvious that $m = n!$ is “nearly always” *even*.
The purpose of a probabilistic analysis is a formal derivation of this intuition about the parity of m when the program terminates.

Classical vs Probabilistic Results

Parity Analysis: Determine at every program point whether a variable is *even* or *odd*.
A safe classical analysis will detect (starting with m and n “unknown")

- that $m = 2 \times n!$ at the end of the second program is always *even*;
- that the parity of m is “unknown” at the end of the first program.

However, it is obvious that $m = n!$ is “nearly always” *even*.
The purpose of a probabilistic analysis is a formal derivation of this intuition about the parity of m when the program terminates.

Classical vs Probabilistic Results

Parity Analysis: Determine at every program point whether a variable is *even* or *odd*.
A safe classical analysis will detect (starting with m and n "unknown")

- that $m = 2 \times n!$ at the end of the second program is always *even*;
- that the parity of m is "unknown" at the end of the first program.

However, it is obvious that $m = n!$ is "nearly always" *even*.
The purpose of a probabilistic analysis is a formal derivation of this intuition about the parity of m when the program terminates.

Classical vs Probabilistic Results

Parity Analysis: Determine at every program point whether a variable is *even* or *odd*.
A safe classical analysis will detect (starting with m and n “unknown")

- that $m = 2 \times n!$ at the end of the second program is always *even*;
- that the parity of m is “unknown” at the end of the first program.

However, it is obvious that $m = n!$ is “nearly always” *even*.
The purpose of a probabilistic analysis is a formal derivation of this intuition about the parity of m when the program terminates.

Double Factorial: Data-flow Analysis

Consider the abstract values $\bot \leq$ **even**; $\bot \leq$ **odd**; **odd** $\leq \top$ and **even** $\leq \top$.

$$\begin{array}{llll} 1: & m \mapsto \top, & n \mapsto \top \\ 2: & m \mapsto \textbf{even}, & n \mapsto \top \\ 3: & & \\ 4: & & \\ 5: & m \mapsto \textbf{even}, & n \mapsto \top \end{array} \qquad \begin{array}{llll} 1: & m \mapsto \top, & n \mapsto \top \\ 2: & m \mapsto \textbf{even}, & n \mapsto \top \\ 3: & m \mapsto \textbf{even}, & n \mapsto \top \\ 4: & m \mapsto \textbf{even}, & n \mapsto \top \\ 5: & m \mapsto \textbf{even}, & n \mapsto \top \end{array}$$

Double Factorial: Data-flow Analysis

Consider the abstract values $\bot \leq \textbf{even}$; $\bot \leq \textbf{odd}$; $\textbf{odd} \leq \top$ and $\textbf{even} \leq \top$.

$$\begin{array}{lll}
1: & m \mapsto \top, & n \mapsto \top \\
2: & m \mapsto \textbf{even}, & n \mapsto \top \\
3: & & \\
4: & & \\
5: & m \mapsto \textbf{even}, & n \mapsto \top
\end{array}
\qquad
\begin{array}{lll}
1: & m \mapsto \top, & n \mapsto \top \\
2: & m \mapsto \textbf{even}, & n \mapsto \top \\
3: & m \mapsto \textbf{even}, & n \mapsto \top \\
4: & m \mapsto \textbf{even}, & n \mapsto \top \\
5: & m \mapsto \textbf{even}, & n \mapsto \top
\end{array}$$

Double Factorial: Data-flow Analysis

Consider the abstract values $\perp \leq$ **even**; $\perp \leq$ **odd**; **odd** $\leq \top$ and **even** $\leq \top$.

$$\begin{array}{lll} 1: & m \mapsto \top, & n \mapsto \top \\ 2: & m \mapsto \textbf{even}, & n \mapsto \top \\ 3: & & \\ 4: & & \\ 5: & m \mapsto \textbf{even}, & n \mapsto \top \end{array} \qquad \begin{array}{lll} 1: & m \mapsto \top, & n \mapsto \top \\ 2: & m \mapsto \textbf{even}, & n \mapsto \top \\ 3: & m \mapsto \textbf{even}, & n \mapsto \top \\ 4: & m \mapsto \textbf{even}, & n \mapsto \top \\ 5: & m \mapsto \textbf{even}, & n \mapsto \top \end{array}$$

Simple Factorial: Data-flow Analysis

If the loop is not executed we can guarantee that `m` is **odd**. If we execute the loop then the analysis will return $\top$ for the parity of `m` at label 5.

$$
\begin{array}{lll}
1: & m \mapsto \top, & n \mapsto \top \\
2: & m \mapsto \textbf{odd}, & n \mapsto \top \\
3: & & \\
4: & & \\
5: & m \mapsto \textbf{odd}, & n \mapsto \top
\end{array}
\qquad
\begin{array}{lll}
1: & m \mapsto \top, & n \mapsto \top \\
2: & m \mapsto \textbf{odd}, & n \mapsto \top \\
3: & m \mapsto \top, & n \mapsto \top \\
4: & m \mapsto \top, & n \mapsto \top \\
5: & m \mapsto \top, & n \mapsto \top
\end{array}
$$

Simple Factorial: Data-flow Analysis

If the loop is not executed we can guarantee that m is **odd**. If we execute the loop then the analysis will return $\top$ for the parity of m at label 5.

1 :	$m \mapsto \top$,	$n \mapsto \top$	1 :	$m \mapsto \top$,	$n \mapsto \top$
2 :	$m \mapsto$ **odd**,	$n \mapsto \top$	2 :	$m \mapsto$ **odd**,	$n \mapsto \top$
3 :			3 :	$m \mapsto \top$,	$n \mapsto \top$
4 :			4 :	$m \mapsto \top$,	$n \mapsto \top$
5 :	$m \mapsto$ **odd**,	$n \mapsto \top$	5 :	$m \mapsto \top$,	$n \mapsto \top$

Simple Factorial: Data-flow Analysis

If the loop is not executed we can guarantee that `m` is **odd**. If we execute the loop then the analysis will return $\top$ for the parity of `m` at label 5.

1 :	$m \mapsto \top,$	$n \mapsto \top$	1 :	$m \mapsto \top,$	$n \mapsto \top$
2 :	$m \mapsto \textbf{odd},$	$n \mapsto \top$	2 :	$m \mapsto \textbf{odd},$	$n \mapsto \top$
3 :			3 :	$m \mapsto \top,$	$n \mapsto \top$
4 :			4 :	$m \mapsto \top,$	$n \mapsto \top$
5 :	$m \mapsto \textbf{odd},$	$n \mapsto \top$	5 :	$m \mapsto \top,$	$n \mapsto \top$

Simple Factorial: Data-flow Analysis

If the loop is not executed we can guarantee that `m` is **odd**. If we execute the loop then the analysis will return $\top$ for the parity of `m` at label 5.

$$
\begin{array}{lll}
1: & m \mapsto \top, & n \mapsto \top \\
2: & m \mapsto \textbf{odd}, & n \mapsto \top \\
3: & & \\
4: & & \\
5: & m \mapsto \textbf{odd}, & n \mapsto \top
\end{array}
\qquad
\begin{array}{lll}
1: & m \mapsto \top, & n \mapsto \top \\
2: & m \mapsto \textbf{odd}, & n \mapsto \top \\
3: & m \mapsto \top, & n \mapsto \top \\
4: & m \mapsto \top, & n \mapsto \top \\
5: & m \mapsto \top, & n \mapsto \top
\end{array}
$$

Conservatism of Data-flow Analysis

- A policy decision is safe or conservative if it never allows us to change what the program computes.
- Classical data-flow analyses computes solutions according to a ‘meet-over-all-paths’ approach
- This guarantees that any errors are in the safe direction
- Safe policies may, unfortunately, cause us to miss some code improvements that would retain the meaning of the program

Conservatism of Data-flow Analysis

- A policy decision is safe or conservative if it never allows us to change what the program computes.
- Classical data-flow analyses computes solutions according to a 'meet-over-all-paths' approach
- This guarantees that any errors are in the safe direction
- Safe policies may, unfortunately, cause us to miss some code improvements that would retain the meaning of the program

Conservatism of Data-flow Analysis

- A policy decision is safe or conservative if it never allows us to change what the program computes.
- Classical data-flow analyses computes solutions according to a ‘meet-over-all-paths’ approach
- This guarantees that any errors are in the safe direction
- Safe policies may, unfortunately, cause us to miss some code improvements that would retain the meaning of the program

Conservatism of Data-flow Analysis

- A policy decision is safe or conservative if it never allows us to change what the program computes.
- Classical data-flow analyses computes solutions according to a 'meet-over-all-paths' approach
- This guarantees that any errors are in the safe direction
- Safe policies may, unfortunately, cause us to miss some code improvements that would retain the meaning of the program

Speculative Optimisation

As compilers must always preserve the program semantics, they are forced to make conservative (i.e. pessimistic) assumptions. Instead:

- Implement a potentially unsafe optimisation
- Verify
- Recover if necessary

Speculative Optimisation

As compilers must always preserve the program semantics, they are forced to make conservative (i.e. pessimistic) assumptions. Instead:

- Implement a potentially unsafe optimisation
- Verify
- Recover if necessary

Speculative Optimisation

As compilers must always preserve the program semantics, they are forced to make conservative (i.e. pessimistic) assumptions. Instead:

- Implement a potentially unsafe optimisation
- Verify
- Recover if necessary

Speculative Optimisation

As compilers must always preserve the program semantics, they are forced to make conservative (i.e. pessimistic) assumptions. Instead:

- Implement a potentially unsafe optimisation
- Verify
- Recover if necessary

Example: Reaching Definitions

A definition *d* reaches a point *p* if there is a path from *d* to *p* such that *d* is not “killed" (i.e. if there is any other definition of x in the path).

A RD analysis determines for any program point *p* which statements that assign, or *may* assign, a value to a variable *x*, reach *p*.

Possible uses for code optimisation:

- a compiler can determine whether *x* is a constant at *p*;
- a debugger can determine whether *x*, used at *p*, may be an undefined variable.

Example: Reaching Definitions

A definition *d* reaches a point *p* if there is a path from *d* to *p* such that *d* is not “killed" (i.e. if there is any other definition of x in the path).

A RD analysis determines for any program point *p* which statements that assign, or *may* assign, a value to a variable *x*, reach *p*.

Possible uses for code optimisation:

- a compiler can determine whether *x* is a constant at *p*;
- a debugger can determine whether *x*, used at *p*, may be an undefined variable.

Example: Reaching Definitions

A definition *d* reaches a point *p* if there is a path from *d* to *p* such that *d* is not “killed" (i.e. if there is any other definition of x in the path).

A RD analysis determines for any program point *p* which statements that assign, or *may* assign, a value to a variable *x*, reach *p*.

Possible uses for code optimisation:

- a compiler can determine whether *x* is a constant at *p*;
- a debugger can determine whether *x*, used at *p*, may be an undefined variable.

Example: Reaching Definitions

A definition *d* reaches a point *p* if there is a path from *d* to *p* such that *d* is not “killed" (i.e. if there is any other definition of x in the path).

A RD analysis determines for any program point *p* which statements that assign, or *may* assign, a value to a variable *x*, reach *p*.

Possible uses for code optimisation:

- a compiler can determine whether *x* is a constant at *p*;
- a debugger can determine whether *x*, used at *p*, may be an undefined variable.

RD: Classical vs Probabilistic

Classical RD analysis assumes that all edges of a flow graph can be traversed. This assumption may not be true in practice.

```
if (a == b) statement 1;
else if (a == b) statement 2;
```

The second statement is actually never reached.

A Probabilistic RD analysis would allow us to use branching probabilities that could establish that the likelihood of taking the `else` path is lower than the `if` branch.

On this basis one could therefore '*speculate*' on whether considering also the unlikely path or not.

RD: Classical vs Probabilistic

Classical RD analysis assumes that all edges of a flow graph can be traversed. This assumption may not be true in practice.

```
if (a == b) statement 1;
else if (a == b) statement 2;
```

The second statement is actually never reached.

A Probabilistic RD analysis would allow us to use branching probabilities that could establish that the likelihood of taking the `else` path is lower than the `if` branch.

On this basis one could therefore '*speculate*' on whether considering also the unlikely path or not.

RD: Classical vs Probabilistic

Classical RD analysis assumes that all edges of a flow graph can be traversed. This assumption may not be true in practice.

```
if (a == b) statement 1;
else if (a == b) statement 2;
```

The second statement is actually never reached.

A Probabilistic RD analysis would allow us to use branching probabilities that could establish that the likelihood of taking the `else` path is lower than the `if` branch.

On this basis one could therefore '*speculate*' on whether considering also the unlikely path or not.

RD: Classical vs Probabilistic

Classical RD analysis assumes that all edges of a flow graph can be traversed. This assumption may not be true in practice.

```
if (a == b) statement 1;
else if (a == b) statement 2;
```

The second statement is actually never reached.

A Probabilistic RD analysis would allow us to use branching probabilities that could establish that the likelihood of taking the `else` path is lower than the `if` branch.

On this basis one could therefore '*speculate*' on whether considering also the unlikely path or not.

RD: Classical vs Probabilistic

Classical RD analysis assumes that all edges of a flow graph can be traversed. This assumption may not be true in practice.

```
if (a == b) statement 1;
else if (a == b) statement 2;
```

The second statement is actually never reached.

A Probabilistic RD analysis would allow us to use branching probabilities that could establish that the likelihood of taking the `else` path is lower than the `if` branch.

On this basis one could therefore '*speculate*' on whether considering also the unlikely path or not.

Example: Live Variables

A variable *x* is live at point *p* if the value of *x* at *p* could be used along some path in the flow graph starting at *p*.

A LV analysis determines for any program point *p* which variables *may* be live at the exit from *p*.

Possible use for code optimisation:

- register assignment
- register allocation

A Probabilistic LV analysis would make use of branching probabilities to estimate the likelihood that a certain variable is later used that could be used to '*speculate*' on wether to perform a certain code optimisation or not.

Example: Live Variables

A variable *x* is live at point *p* if the value of *x* at *p* could be used along some path in the flow graph starting at *p*.
A LV analysis determines for any program point *p* which variables *may* be live at the exit from *p*.
Possible use for code optimisation:

- register assignment
- register allocation

A Probabilistic LV analysis would make use of branching probabilities to estimate the likelihood that a certain variable is later used that could be used to '*speculate*' on wether to perform a certain code optimisation or not.

Example: Live Variables

A variable *x* is live at point *p* if the value of *x* at *p* could be used along some path in the flow graph starting at *p*.
A LV analysis determines for any program point *p* which variables *may* be live at the exit from *p*.
Possible use for code optimisation:

- register assignment
- register allocation

A Probabilistic LV analysis would make use of branching probabilities to estimate the likelihood that a certain variable is later used that could be used to '*speculate*' on wether to perform a certain code optimisation or not.

Example: Live Variables

A variable x is live at point p if the value of x at p could be used along some path in the flow graph starting at p.
A LV analysis determines for any program point p which variables *may* be live at the exit from p.
Possible use for code optimisation:

- register assignment
- register allocation

A Probabilistic LV analysis would make use of branching probabilities to estimate the likelihood that a certain variable is later used that could be used to '*speculate*' on wether to perform a certain code optimisation or not.

Example: Live Variables

A variable x is live at point p if the value of x at p could be used along some path in the flow graph starting at p.
A LV analysis determines for any program point p which variables *may* be live at the exit from p.
Possible use for code optimisation:

- register assignment
- register allocation

A Probabilistic LV analysis would make use of branching probabilities to estimate the likelihood that a certain variable is later used that could be used to '*speculate*' on wether to perform a certain code optimisation or not.

A Probabilistic Language

We present a simple imperative language with *probabilistic choice*.

We will use this language to define

- a probabilistic semantics
- probabilistic analysis techniques based on it.

A Probabilistic Language

We present a simple imperative language with *probabilistic choice*.

We will use this language to define

- a probabilistic semantics
- probabilistic analysis techniques based on it.

A Probabilistic Language

We present a simple imperative language with *probabilistic choice*.

We will use this language to define

- a probabilistic semantics
- probabilistic analysis techniques based on it.

A Probabilistic Language

We present a simple imperative language with *probabilistic choice*.

We will use this language to define

- a probabilistic semantics
- probabilistic analysis techniques based on it.

pWhile – Syntax I

Full programs contain optional variable declarations:

$$
\begin{array}{rcl}
P & ::= & \textbf{begin } S \textbf{ end} \\
 & | & \textbf{var } D \textbf{ begin } S \textbf{ end}
\end{array}
$$

Declarations are of the form:

$$
\begin{array}{rcl}
r & ::= & \textbf{bool} \\
 & | & \textbf{int} \\
 & | & \{ \, c_1, \ldots, c_n \, \} \\
 & | & \{ \, c_1 \,..\, c_n \, \} \\
D & ::= & v \textbf{ : } r \\
 & | & v \textbf{ : } r \textbf{ ; } D
\end{array}
$$

with c_i (integer) constants and r denoting ranges.

pWhile – Syntax I

Full programs contain optional variable declarations:

$$
\begin{array}{lcl}
P & ::= & \textbf{begin } S \textbf{ end} \\
 & | & \textbf{var } D \textbf{ begin } S \textbf{ end}
\end{array}
$$

Declarations are of the form:

$$
\begin{array}{lcl}
r & ::= & \textbf{bool} \\
 & | & \textbf{int} \\
 & | & \{\, c_1, \ldots, c_n \,\} \\
 & | & \{\, c_1 \,..\, c_n \,\} \\
D & ::= & v : r \\
 & | & v : r ;\ D
\end{array}
$$

with c_i (integer) constants and r denoting ranges.

pWhile – Syntax II

The syntax of statements S is as follows:

$$
\begin{array}{rcl}
S & ::= & \textbf{stop} \\
 & | & \textbf{skip} \\
 & | & v \ := a \\
 & | & S_1 ; \ S_2 \\
 & | & \textbf{choose } p_1 : S_1 \textbf{ or } p_2 : S_2 \textbf{ ro} \\
 & | & \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \\
 & | & \textbf{while } b \textbf{ do } S \textbf{ od}
\end{array}
$$

Where the p_i are constants, representing choice probabilities.

pWhile – Syntax II

The syntax of statements S is as follows:

$$
\begin{array}{rcl}
S & ::= & [\mathbf{stop}]^{\ell} \\
 & | & [\mathbf{skip}]^{\ell} \\
 & | & [v := a]^{\ell} \\
 & | & S_1 ; \ S_2 \\
 & | & \mathbf{choose}^{\ell} \ p_1 : S_1 \ \mathbf{or} \ p_2 : S_2 \ \mathbf{ro} \\
 & | & \mathbf{if} \ [b]^{\ell} \ \mathbf{then} \ S_1 \ \mathbf{else} \ S_2 \ \mathbf{fi} \\
 & | & \mathbf{while} \ [b]^{\ell} \ \mathbf{do} \ S \ \mathbf{od}
\end{array}
$$

Where the p_i are constants, representing choice probabilities.

Evaluation of Expressions

$$\sigma \ni \textbf{State} = \textbf{Var} \to \textbf{Z} \uplus \textbf{B}$$

Evaluation $\mathcal{E}$ of expressions e in state σ:

$$\begin{array}{rcl} \mathcal{E}(n)\sigma & = & n \\ \mathcal{E}(v)\sigma & = & \sigma(v) \\ \mathcal{E}(a_1 \odot a_2)\sigma & = & \mathcal{E}(a_1)\sigma \odot \mathcal{E}(a_2)\sigma \end{array}$$

$$\begin{array}{rcl} \mathcal{E}(\textbf{true})\sigma & = & \textbf{tt} \\ \mathcal{E}(\textbf{false})\sigma & = & \textbf{ff} \\ \mathcal{E}(\textbf{not}\ b)\sigma & = & \neg\mathcal{E}(b)\sigma \\ \ldots & = & \ldots \end{array}$$

Evaluation of Expressions

$$\sigma \ni \textbf{State} = \textbf{Var} \to \textbf{Z} \uplus \textbf{B}$$

Evaluation $\mathcal{E}$ of expressions e in state σ:

$$\begin{array}{rcl}
\mathcal{E}(n)\sigma & = & n \\
\mathcal{E}(v)\sigma & = & \sigma(v) \\
\mathcal{E}(a_1 \odot a_2)\sigma & = & \mathcal{E}(a_1)\sigma \odot \mathcal{E}(a_2)\sigma
\end{array}$$

$$\begin{array}{rcl}
\mathcal{E}(\textbf{true})\sigma & = & \textbf{tt} \\
\mathcal{E}(\textbf{false})\sigma & = & \textbf{ff} \\
\mathcal{E}(\textbf{not } b)\sigma & = & \neg\mathcal{E}(b)\sigma \\
\ldots & = & \ldots
\end{array}$$

Evaluation of Expressions

$$\sigma \ni \textbf{State} = \textbf{Var} \to \textbf{Z} \uplus \textbf{B}$$

Evaluation $\mathcal{E}$ of expressions *e* in state σ:

$$\begin{array}{rcl} \mathcal{E}(n)\sigma & = & n \\ \mathcal{E}(v)\sigma & = & \sigma(v) \\ \mathcal{E}(a_1 \odot a_2)\sigma & = & \mathcal{E}(a_1)\sigma \odot \mathcal{E}(a_2)\sigma \end{array}$$

$$\begin{array}{rcl} \mathcal{E}(\textbf{true})\sigma & = & \textbf{tt} \\ \mathcal{E}(\textbf{false})\sigma & = & \textbf{ff} \\ \mathcal{E}(\textbf{not}\ b)\sigma & = & \neg\mathcal{E}(b)\sigma \\ \ldots & = & \ldots \end{array}$$

pWhile – SOS Semantics I

R0 $\langle \textbf{skip}, \sigma\rangle \Rightarrow_1 \langle \textbf{stop}, \sigma\rangle$

R1 $\langle \textbf{stop}, \sigma\rangle \Rightarrow_1 \langle \textbf{stop}, \sigma\rangle$

R2 $\langle \mathtt{v} := e, \sigma\rangle \Rightarrow_1 \langle \textbf{stop}, \sigma[v \mapsto \mathcal{E}(e)\sigma]\rangle$

R3$_1$ $\dfrac{\langle S_1, \sigma\rangle \Rightarrow_p \langle S_1', \sigma'\rangle}{\langle S_1; S_2, \sigma\rangle \Rightarrow_p \langle S_1'; S_2, \sigma'\rangle}$

R3$_2$ $\dfrac{\langle S_1, \sigma\rangle \Rightarrow_p \langle \textbf{stop}, \sigma'\rangle}{\langle S_1; S_2, \sigma\rangle \Rightarrow_p \langle S_2, \sigma'\rangle}$

pWhile – SOS Semantics II

R4$_1$	$\langle \textbf{choose}\ p_1 : S_1\ \textbf{or}\ p_2 : S_2, \sigma\rangle \Rightarrow_{p_1} \langle S_1, \sigma\rangle$	
R4$_2$	$\langle \textbf{choose}\ p_1 : S_1\ \textbf{or}\ p_2 : S_2, \sigma\rangle \Rightarrow_{p_2} \langle S_2, \sigma\rangle$	
R5$_1$	$\langle \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2, \sigma\rangle \Rightarrow_1 \langle S_1, \sigma\rangle$	if $\mathcal{E}(b)\sigma = \textbf{tt}$
R5$_2$	$\langle \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2, \sigma\rangle \Rightarrow_1 \langle S_2, \sigma\rangle$	if $\mathcal{E}(b)\sigma = \textbf{ff}$
R6$_1$	$\langle \textbf{while}\ b\ \textbf{do}\ S, \sigma\rangle \Rightarrow_1 \langle S;\ \textbf{while}\ b\ \textbf{do}\ S, \sigma\rangle$	if $\mathcal{E}(b)\sigma = \textbf{tt}$
R6$_2$	$\langle \textbf{while}\ b\ \textbf{do}\ S, \sigma\rangle \Rightarrow_1 \langle \textbf{stop}, \sigma\rangle$	if $\mathcal{E}(b)\sigma = \textbf{ff}$

DTMC

Markov chains behave as transition systems where nondeterministic choices among successor states are replaced by probabilistic ones.

Equivalently: the successor state of a state s is chosen according to a probability distribution **d**.

d only depends on the current state s, and evolution does not depend on the history (memoryless property).

The name Discrete Time Markov Chain (DTMC) refers to the fact that Markov chains are used as a time-abstract model (like transition systems): each transition is assumed to take a single time unit.

DTMC

Markov chains behave as transition systems where nondeterministic choices among successor states are replaced by probabilistic ones.
Equivalently: the successor state of a state s is chosen according to a probability distribution **d**.

d only depends on the current state s, and evolution does not depend on the history (memoryless property).

The name Discrete Time Markov Chain (DTMC) refers to the fact that Markov chains are used as a time-abstract model (like transition systems): each transition is assumed to take a single time unit.

DTMC

Markov chains behave as transition systems where nondeterministic choices among successor states are replaced by probabilistic ones.
Equivalently: the successor state of a state s is chosen according to a probability distribution **d**.

d only depends on the current state s, and evolution does not depend on the history (memoryless property).

The name Discrete Time Markov Chain (DTMC) refers to the fact that Markov chains are used as a time-abstract model (like transition systems): each transition is assumed to take a single time unit.

DTMC

Markov chains behave as transition systems where nondeterministic choices among successor states are replaced by probabilistic ones.
Equivalently: the successor state of a state s is chosen according to a probability distribution $\mathbf{d}$.

$\mathbf{d}$ only depends on the current state s, and evolution does not depend on the history (memoryless property).

The name Discrete Time Markov Chain (DTMC) refers to the fact that Markov chains are used as a time-abstract model (like transition systems): each transition is assumed to take a single time unit.

DTMC

Markov chains behave as transition systems where nondeterministic choices among successor states are replaced by probabilistic ones.
Equivalently: the successor state of a state s is chosen according to a probability distribution **d**.

d only depends on the current state s, and evolution does not depend on the history (memoryless property).

The name Discrete Time Markov Chain (DTMC) refers to the fact that Markov chains are used as a time-abstract model (like transition systems): each transition is assumed to take a single time unit.

DTMC: Formal Definition

Definition

A DTMC is a tuple $(S, \mathbf{P}, \iota_{in})$ where

- S is a countable, nonempty set of states,
- $\mathbf{P} : S \times S \mapsto [0, 1]$ is the *transition probability* function such that for all $s \in S$
$$\sum_{s' \in S} \mathbf{P}(s, s') = 1,$$
- $\iota_{in} : S \mapsto [0, 1]$ is the *initial distribution*, s.t. $\sum_{s \in S} \iota_{in}(s) = 1$.

Paths in a DTMC are *maximal* (i.e. infinite) in the underlying directed graph.

DTMC: Formal Definition

Definition

A DTMC is a tuple $(S, \mathbf{P}, \iota_{in})$ where

- S is a countable, nonempty set of states,
- $\mathbf{P} : S \times S \mapsto [0, 1]$ is the *transition probability* function such that for all $s \in S$
$$\sum_{s' \in S} \mathbf{P}(s, s') = 1,$$
- $\iota_{in} : S \mapsto [0, 1]$ is the *initial distribution*, s.t. $\sum_{s \in S} \iota_{in}(s) = 1$.

Paths in a DTMC are *maximal* (i.e. infinite) in the underlying directed graph.

DTMC: Formal Definition

Definition

A DTMC is a tuple $(S, \mathbf{P}, \iota_{in})$ where

- S is a countable, nonempty set of states,
- $\mathbf{P} : S \times S \mapsto [0, 1]$ is the *transition probability* function such that for all $s \in S$
$$\sum_{s' \in S} \mathbf{P}(s, s') = 1,$$
- $\iota_{in} : S \mapsto [0, 1]$ is the *initial distribution*, s.t. $\sum_{s \in S} \iota_{in}(s) = 1$.

Paths in a DTMC are *maximal* (i.e. infinite) in the underlying directed graph.

DTMC: Formal Definition

Definition

A DTMC is a tuple $(S, \mathbf{P}, \iota_{in})$ where

- S is a countable, nonempty set of states,
- $\mathbf{P} : S \times S \mapsto [0, 1]$ is the *transition probability* function such that for all $s \in S$
$$\sum_{s' \in S} \mathbf{P}(s, s') = 1,$$
- $\iota_{in} : S \mapsto [0, 1]$ is the *initial distribution*, s.t. $\sum_{s \in S} \iota_{in}(s) = 1$.

Paths in a DTMC are *maximal* (i.e. infinite) in the underlying directed graph.

DTMC Semantics

Given a **pWhile** program, consider any enumeration of all its configurations (= pairs of statements and state) $C_1, C_2, C_3, \ldots \in$ **Conf**. Then

$$(\mathbf{T})_{ij} = \begin{cases} p & \text{if } C_i \Rightarrow_p C_j \\ 0 & \text{otherwise} \end{cases}$$

is the generator of a Discrete Time Markov Chain.

Transitions are implemented as

$$\mathbf{d}_n \cdot \mathbf{T} = \sum_i (\mathbf{d}_n)_i \cdot \mathbf{T}_{ij} = \mathbf{d}_{n+1}$$

where $\mathbf{d}_i$ is the probability distribution over **Conf** at the ith step.

DTMC Semantics

Given a **pWhile** program, consider any enumeration of all its configurations (= pairs of statements and state) $C_1, C_2, C_3, \ldots \in$ **Conf**. Then

$$(\mathbf{T})_{ij} = \begin{cases} p & \text{if } C_i = \langle S, \sigma \rangle \Rightarrow_p C_j = \langle S', \sigma' \rangle \\ 0 & \text{otherwise} \end{cases}$$

is the generator of a Discrete Time Markov Chain.

Transitions are implemented as

$$\mathbf{d}_n \cdot \mathbf{T} = \sum_i (\mathbf{d}_n)_i \cdot \mathbf{T}_{ij} = \mathbf{d}_{n+1}$$

where $\mathbf{d}_i$ is the probability distribution over **Conf** at the ith step.

DTMC Semantics

Given a **pWhile** program, consider any enumeration of all its configurations (= pairs of statements and state) $C_1, C_2, C_3, \ldots \in$ **Conf**. Then

$$(\mathbf{T})_{ij} = \begin{cases} p & \text{if } C_i \Rightarrow_p C_j \\ 0 & \text{otherwise} \end{cases}$$

is the generator of a Discrete Time Markov Chain.

Transitions are implemented as

$$\mathbf{d}_n \cdot \mathbf{T} = \sum_i (\mathbf{d}_n)_i \cdot \mathbf{T}_{ij} = \mathbf{d}_{n+1}$$

where $\mathbf{d}_i$ is the probability distribution over **Conf** at the ith step.

DTMC Semantics

Given a **pWhile** program, consider any enumeration of all its configurations (= pairs of statements and state) $C_1, C_2, C_3, \ldots \in \textbf{Conf}$. Then

$$(\mathbf{T})_{ij} = \begin{cases} p & \text{if } C_i \Rightarrow_p C_j \\ 0 & \text{otherwise} \end{cases}$$

is the generator of a Discrete Time Markov Chain.

Transitions are implemented as

$$\mathbf{d}_n \cdot \mathbf{T} = \sum_i (\mathbf{d}_n)_i \cdot \mathbf{T}_{ij} = \mathbf{d}_{n+1}$$

where $\mathbf{d}_i$ is the probability distribution over **Conf** at the ith step.

DTMC Semantics

Given a **pWhile** program, consider any enumeration of all its configurations (= pairs of statements and state) $C_1, C_2, C_3, \ldots \in$ **Conf**. Then

$$(\mathbf{T})_{ij} = \begin{cases} p & \text{if } C_i \Rightarrow_p C_j \\ 0 & \text{otherwise} \end{cases}$$

is the generator of a Discrete Time Markov Chain.

Transitions are implemented as

$$\mathbf{d}_n \cdot \mathbf{T} = \sum_i (\mathbf{d}_n)_i \cdot \mathbf{T}_{ij} = \mathbf{d}_{n+1}$$

where $\mathbf{d}_i$ is the probability distribution over **Conf** at the ith step.

DTMC Semantics

Given a **pWhile** program, consider any enumeration of all its configurations (= pairs of statements and state) $C_1, C_2, C_3, \ldots \in \mathbf{Conf}$. Then

$$(\mathbf{T})_{ij} = \begin{cases} p & \text{if } C_i \Rightarrow_p C_j \\ 0 & \text{otherwise} \end{cases}$$

is the generator of a Discrete Time Markov Chain.

Transitions are implemented as

$$\mathbf{d}_n \cdot \mathbf{T} = \sum_i (\mathbf{d}_n)_i \cdot \mathbf{T}_{ij} = \mathbf{d}_{n+1}$$

where $\mathbf{d}_i$ is the probability distribution over **Conf** at the *i*th step.

Example Program

Let us investigate the possible transitions of the following labelled program (with $\mathbf{x} \in \{0, 1\}$):

$$
\begin{array}{l}
\textbf{if } [\mathbf{x} = 0] \textbf{ then} \\
\quad [\mathbf{x} := 0]; \\
\textbf{else} \\
\quad [\mathbf{x} := 1]; \\
\textbf{fi}; \\
[\textbf{stop}]
\end{array}
$$

Example DTMC

$$
\begin{array}{lc}
\langle x=0,[\mathbf{x}=0]\rangle & \ldots \\
\langle x=0,[\mathbf{x}:=0]\rangle & \ldots \\
\langle x=0,[\mathbf{x}:=1]\rangle & \ldots \\
\langle x=0,[\mathbf{stop}]\rangle & \ldots \\
\langle x=1,[\mathbf{x}=0]\rangle & \ldots \\
\langle x=1,[\mathbf{x}:=0]\rangle & \ldots \\
\langle x=1,[\mathbf{x}:=1]\rangle & \ldots \\
\langle x=1,[\mathbf{stop}]\rangle & \ldots
\end{array}
\quad
\begin{pmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
$$

Example Transition

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

We get: $\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$.

Example Transition

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

We get: $\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$.

Dataflow Analysis

Dataflow analyses work by calculating an assignment of abstract states to the edges of a control-flow graph.

Depending on whether the analysis is forward or backward, either the direct or the inverse control-flow graph of a given program is used and the calculation takes place by propagating abstract states across the nodes of the graph in the appropriate direction.

Probabilistic dataflow analyses work in the same way, but calculation is carried out by propagating probabilities together with abstract states.

Dataflow Analysis

Dataflow analyses work by calculating an assignment of abstract states to the edges of a control-flow graph.

Depending on whether the analysis is forward or backward, either the direct or the inverse control-flow graph of a given program is used and the calculation takes place by propagating abstract states across the nodes of the graph in the appropriate direction.

Probabilistic dataflow analyses work in the same way, but calculation is carried out by propagating probabilities together with abstract states.

Dataflow Analysis

Dataflow analyses work by calculating an assignment of abstract states to the edges of a control-flow graph.

Depending on whether the analysis is forward or backward, either the direct or the inverse control-flow graph of a given program is used and the calculation takes place by propagating abstract states across the nodes of the graph in the appropriate direction.

Probabilistic dataflow analyses work in the same way, but calculation is carried out by propagating probabilities together with abstract states.

An Example

Consider the following program, $\texttt{power}$, computing the $\texttt{x}$-th power of the number stored in $\texttt{y}$:

$$
\begin{array}{l}
[\, \texttt{z} := \texttt{1} \,]^1; \\
\textbf{while } [\texttt{x} > \texttt{1}]^2 \textbf{ do } (\\
\qquad [\, \texttt{z} := \texttt{z}*\texttt{y} \,]^3; \\
\qquad [\, \texttt{x} := \texttt{x}-\texttt{1} \,]^4);
\end{array}
$$

We have $\mathit{labels}(\texttt{power}) = \{1, 2, 3, 4\}$, $\mathit{init}(\texttt{power}) = 1$, and $\mathit{final}(\texttt{power}) = \{2\}$. The function *flow* produces the set:

$$\mathit{flow}(\texttt{power}) = \{(1, 2), (2, 3), (3, 4), (4, 2)\}$$

An Example

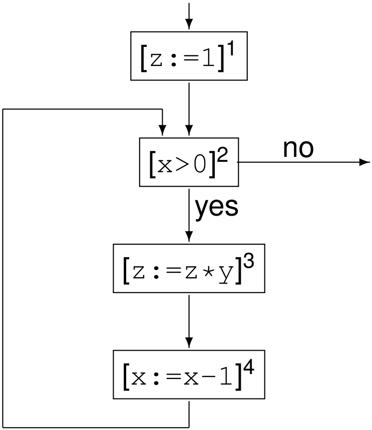
Consider the following program, `power`, computing the `x`-th power of the number stored in `y`:

$$
\begin{array}{l}
[\,\texttt{z := 1}\,]^1; \\
\textbf{while } [\texttt{x > 1}]^2 \textbf{ do } (\\
\qquad [\,\texttt{z := z*y}\,]^3; \\
\qquad [\,\texttt{x := x-1}\,]^4);
\end{array}
$$

We have *labels*($\texttt{power}$) $= \{1, 2, 3, 4\}$, *init*($\texttt{power}$) $= 1$, and *final*($\texttt{power}$) $= \{2\}$. The function *flow* produces the set:

$$\mathit{flow}(\texttt{power}) = \{(1, 2), (2, 3), (3, 4), (4, 2)\}$$

Flow Graph



Probabilistic Control Flow

Consider the following labelled program:

while $[\mathbf{z} < 100]^1$ **do**
  $[\textbf{choose}]^2\ \frac{1}{3} : [\mathbf{x} := 3]^3$ **or** $\frac{2}{3} : [\mathbf{x} := 1]^4$ **ro**
od
$[\textbf{stop}]^5$

Its probabilistic control flow is given by:

$$flow(P) = \{\langle 1, 1, 2\rangle, \langle 1, 1, 5\rangle, \langle 2, \frac{1}{3}, 3\rangle, \langle 2, \frac{2}{3}, 4\rangle, \langle 3, 1, 1\rangle, \langle 4, 1, 1\rangle\}.$$

Probabilistic Control Flow

Consider the following labelled program:

while $[z < 100]^1$ **do**
    $[\textbf{choose}]^2\ \frac{1}{3} : [\mathbf{x} := 3]^3$ **or** $\frac{2}{3} : [\mathbf{x} := 1]^4$ **ro**
od
$[\textbf{stop}]^5$

Its probabilistic control flow is given by:

$$\mathit{flow}(P) = \{\langle 1, 1, 2\rangle, \langle 1, 1, 5\rangle, \langle 2, \frac{1}{3}, 3\rangle, \langle 2, \frac{2}{3}, 4\rangle, \langle 3, 1, 1\rangle, \langle 4, 1, 1\rangle\}.$$

Init — First Label

$$
\begin{aligned}
\mathit{init}([\mathbf{skip}]^{\ell}) &= \ell \\
\mathit{init}([\mathbf{stop}]^{\ell}) &= \ell \\
\mathit{init}([\mathrm{v}\,{:=}\,e]^{\ell}) &= \ell \\
\mathit{init}(S_1; S_2) &= \mathit{init}(S_1) \\
\mathit{init}([\mathbf{choose}]^{\ell}\ p_1 : S_1 \ \mathbf{or}\ p_2 : S_2) &= \ell \\
\mathit{init}(\mathbf{if}\ [b]^{\ell}\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2) &= \ell \\
\mathit{init}(\mathbf{while}\ [b]^{\ell}\ \mathbf{do}\ S) &= \ell
\end{aligned}
$$

Final — Last Labels

$$
\begin{array}{rcl}
\mathit{final}([\textbf{skip}]^\ell) & = & \{\ell\} \\
\mathit{final}([\textbf{stop}]^\ell) & = & \{\ell\} \\
\mathit{final}([\mathtt{v} := e]^\ell) & = & \{\ell\} \\
\mathit{final}(S_1 ; S_2) & = & \mathit{final}(S_2) \\
\mathit{final}([\textbf{choose}]^\ell \; p_1 : S_1 \textbf{ or } p_2 : S_2) & = & \mathit{final}(S_1) \cup \mathit{final}(S_2) \\
\mathit{final}(\textbf{if } [b]^\ell \textbf{ then } S_1 \textbf{ else } S_2) & = & \mathit{final}(S_1) \cup \mathit{final}(S_2) \\
\mathit{final}(\textbf{while } [b]^\ell \textbf{ do } S) & = & \{\ell\}
\end{array}
$$

Flow I — Control Transfer

The probabilistic control flow is defined by the function:

$$flow : \textbf{Stmt} \rightarrow \mathcal{P}(\textbf{Lab} \times [0,1] \times \textbf{Lab})$$

$$\begin{array}{rcl}
flow([\textbf{skip}]^{\ell}) & = & \emptyset \\
flow([\textbf{stop}]^{\ell}) & = & \{\langle \ell, 1, \ell \rangle\} \\
flow([\mathrm{v} := e]^{\ell}) & = & \emptyset \\
flow(S_1 ; S_2) & = & flow(S_1) \cup flow(S_2) \cup \\
 & \cup & \{(\ell, 1, init(S_2)) \mid \ell \in final(S_1)\}
\end{array}$$

Flow I — Control Transfer

The probabilistic control flow is defined by the function:

$$\mathit{flow} : \textbf{Stmt} \to \mathcal{P}(\textbf{Lab} \times [0,1] \times \textbf{Lab})$$

$$\begin{array}{rcl}
\mathit{flow}([\textbf{skip}]^{\ell}) & = & \emptyset \\
\mathit{flow}([\textbf{stop}]^{\ell}) & = & \{\langle \ell, 1, \ell \rangle\} \\
\mathit{flow}([\mathrm{v} \,\textbf{:=}\, e]^{\ell}) & = & \emptyset \\
\mathit{flow}(S_1 ; S_2) & = & \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \\
 & \cup & \{(\ell, 1, \mathit{init}(S_2)) \mid \ell \in \mathit{final}(S_1)\}
\end{array}$$

Flow II — Control Transfer

$$
\begin{array}{rcl}
\mathit{flow}([\textbf{choose}]^{\ell}\ p_1 : S_1\ \textbf{or}\ p_2 : S_2) & = & \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \\
& \cup & \{(\ell, p_1, \mathit{init}(S_1)), (\ell, p_2, \mathit{init}(S_2)) \\
\mathit{flow}(\textbf{if}\ [b]^{\ell}\ \textbf{then}\ S_1\ \textbf{else}\ S_2) & = & \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \\
& \cup & \{(\ell, 1, \mathit{init}(S_1)), (\ell, 1, \mathit{init}(S_2))\} \\
\mathit{flow}(\textbf{while}\ [b]^{\ell}\ \textbf{do}\ S) & = & \mathit{flow}(S) \cup \\
& \cup & \{(\ell, 1, \mathit{init}(S))\} \\
& \cup & \{(\ell', 1, \ell) \mid \ell' \in \mathit{final}(S)\}
\end{array}
$$

Linear Operator Semantics (LOS)

The matrix representation of the SOS semantics of a **pWhile** program is not ‘compositional’.

In order to be able to analyse programs by analysing its parts, a more useful semantics is one resulting from the composition of different linear operators each expressing a particular operation contributing to the overall behaviour of the program.

Linear Operator Semantics (LOS)

The matrix representation of the SOS semantics of a **pWhile** program is not 'compositional'.

In order to be able to analyse programs by analysing its parts, a more useful semantics is one resulting from the composition of different linear operators each expressing a particular operation contributing to the overall behaviour of the program.

The Space of Configurations

For a **pWhile** program P we can identify configurations with elements in

$$\mathbf{Dist}(\mathbf{State} \times \mathbf{Lab}) \subseteq \mathcal{V}(\mathbf{State} \times \mathbf{Lab}).$$

Assuming $v = |\mathbf{Var}|$ finite,

$$\mathbf{State} = (\mathbb{Z} + \mathbb{B})^v = \mathbf{Value}_1 \times \mathbf{Value}_2 \ldots \times \mathbf{Value}_v$$

with $\mathbf{Value}_i = \mathbb{Z}$ or $\mathbb{B}$.

Thus, we can represent the space of configurations as

$$\mathbf{Dist}(\mathbf{Value}_1 \times \ldots \times \mathbf{Value}_v \times \mathbf{Lab}) \quad \subseteq$$
$$\mathcal{V}(\mathbf{Value}_1) \otimes \ldots \otimes \mathcal{V}(\mathbf{Value}_v) \otimes \mathcal{V}(\mathbf{Lab}).$$

The Space of Configurations

For a **pWhile** program P we can identify configurations with elements in

$$\mathbf{Dist}(\mathbf{State} \times \mathbf{Lab}) \subseteq \mathcal{V}(\mathbf{State} \times \mathbf{Lab}).$$

Assuming $v = |\mathbf{Var}|$ finite,

$$\mathbf{State} = (\mathbb{Z} + \mathbb{B})^v = \mathbf{Value}_1 \times \mathbf{Value}_2 \ldots \times \mathbf{Value}_v$$

with $\mathbf{Value}_i = \mathbb{Z}$ or $\mathbb{B}$.

Thus, we can represent the space of configurations as

$$\mathbf{Dist}(\mathbf{Value}_1 \times \ldots \times \mathbf{Value}_v \times \mathbf{Lab}) \quad \subseteq$$
$$\mathcal{V}(\mathbf{Value}_1) \otimes \ldots \otimes \mathcal{V}(\mathbf{Value}_v) \otimes \mathcal{V}(\mathbf{Lab}).$$

The Space of Configurations

For a **pWhile** program P we can identify configurations with elements in

$$\textbf{Dist}(\textbf{State} \times \textbf{Lab}) \subseteq \mathcal{V}(\textbf{State} \times \textbf{Lab}).$$

Assuming $v = |\textbf{Var}|$ finite,

$$\textbf{State} = (\underline{\mathbb{Z}} + \mathbb{B})^v = \textbf{Value}_1 \times \textbf{Value}_2 \ldots \times \textbf{Value}_v$$

with $\textbf{Value}_i = \underline{\mathbb{Z}}$ or $\mathbb{B}$.

Thus, we can represent the space of configurations as

$$\textbf{Dist}(\textbf{Value}_1 \times \ldots \times \textbf{Value}_v \times \textbf{Lab}) \quad \subseteq$$
$$\mathcal{V}(\textbf{Value}_1) \otimes \ldots \otimes \mathcal{V}(\textbf{Value}_v) \otimes \mathcal{V}(\textbf{Lab}).$$

The Space of Configurations

For a **pWhile** program P we can identify configurations with elements in

$$\mathbf{Dist}(\mathbf{State} \times \mathbf{Lab}) \subseteq \mathcal{V}(\mathbf{State} \times \mathbf{Lab}).$$

Assuming $v = |\mathbf{Var}|$ finite,

$$\mathbf{State} = (\underline{\mathbb{Z}} + \mathbb{B})^v = \mathbf{Value}_1 \times \mathbf{Value}_2 \ldots \times \mathbf{Value}_v$$

with $\mathbf{Value}_i = \underline{\mathbb{Z}}$ or $\mathbb{B}$.

Thus, we can represent the space of configurations as

$$\mathbf{Dist}(\mathbf{Value}_1 \times \ldots \times \mathbf{Value}_v \times \mathbf{Lab}) \quad \subseteq$$
$$\mathcal{V}(\mathbf{Value}_1) \otimes \ldots \otimes \mathcal{V}(\mathbf{Value}_v) \otimes \mathcal{V}(\mathbf{Lab}).$$

Tensor Product

Given a $n \times m$ matrix $\mathbf{A}$ and a $k \times l$ matrix $\mathbf{B}$:

$$\mathbf{A} = \begin{pmatrix} a_{11} & \dots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nm} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & \dots & b_{1l} \\ \vdots & \ddots & \vdots \\ b_{k1} & \dots & b_{kl} \end{pmatrix}$$

The tensor product $\mathbf{A} \otimes \mathbf{B}$ is a $nk \times ml$ matrix:

$$\mathbf{A} \otimes \mathbf{B} = \begin{pmatrix} a_{11}\mathbf{B} & \dots & a_{1m}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{n1}\mathbf{B} & \dots & a_{nm}\mathbf{B} \end{pmatrix}$$

Special cases are square matrices ($n = m$ and $k = l$) and vectors (row $n = k = 1$, column $m = l = 1$).

Tensor Product

Given a $n \times m$ matrix $\mathbf{A}$ and a $k \times l$ matrix $\mathbf{B}$:

$$\mathbf{A} = \begin{pmatrix} a_{11} & \dots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nm} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & \dots & b_{1l} \\ \vdots & \ddots & \vdots \\ b_{k1} & \dots & b_{kl} \end{pmatrix}$$

The tensor product $\mathbf{A} \otimes \mathbf{B}$ is a $nk \times ml$ matrix:

$$\mathbf{A} \otimes \mathbf{B} = \begin{pmatrix} a_{11}\mathbf{B} & \dots & a_{1m}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{n1}\mathbf{B} & \dots & a_{nm}\mathbf{B} \end{pmatrix}$$

Special cases are square matrices ($n = m$ and $k = l$) and vectors (row $n = k = 1$, column $m = l = 1$).

Tensor Product

Given a $n \times m$ matrix $\mathbf{A}$ and a $k \times l$ matrix $\mathbf{B}$:

$$\mathbf{A} = \begin{pmatrix} a_{11} & \dots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nm} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & \dots & b_{1l} \\ \vdots & \ddots & \vdots \\ b_{k1} & \dots & b_{kl} \end{pmatrix}$$

The tensor product $\mathbf{A} \otimes \mathbf{B}$ is a $nk \times ml$ matrix:

$$\mathbf{A} \otimes \mathbf{B} = \begin{pmatrix} a_{11}\mathbf{B} & \dots & a_{1m}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{n1}\mathbf{B} & \dots & a_{nm}\mathbf{B} \end{pmatrix}$$

Special cases are square matrices ($n = m$ and $k = l$) and vectors (row $n = k = 1$, column $m = l = 1$).

A Linear Operator based on *flow*

$$\mathbf{T}(P) = \sum_{\langle i, p_{ij}, j\rangle \in \mathit{flow}(P)} p_{ij} \cdot \mathbf{T}(\ell_i, \ell_j),$$

where

$$\mathbf{T}(\ell_i, \ell_j) = \mathbf{N} \otimes \mathbf{E}(\ell_i, \ell_j),$$

with **N** an operator representing a state update while the second factor realises the transfer of control from label ℓ_i to label ℓ_j.

A Linear Operator based on *flow*

$$\mathbf{T}(P) = \sum_{\langle i, p_{ij}, j\rangle \in \mathit{flow}(P)} p_{ij} \cdot \mathbf{T}(\ell_i, \ell_j),$$

where

$$\mathbf{T}(\ell_i, \ell_j) = \mathbf{N} \otimes \mathbf{E}(\ell_i, \ell_j),$$

with **N** an operator representing a state update while the second factor realises the transfer of control from label ℓ_i to label ℓ_j.

A Linear Operator based on *flow*

$$\mathbf{T}(P) = \sum_{\langle i, p_{ij}, j\rangle \in \mathit{flow}(P)} p_{ij} \cdot \mathbf{T}(\ell_i, \ell_j),$$

where

$$\mathbf{T}(\ell_i, \ell_j) = \mathbf{N} \otimes \mathbf{E}(\ell_i, \ell_j),$$

with **N** an operator representing a state update while the second factor realises the transfer of control from label ℓ_i to label ℓ_j.

Transfer Operators

$$
\begin{array}{rcll}
\mathbf{T}(\langle \ell_1, p, \ell_2 \rangle) & = & \mathbf{I} \otimes \mathbf{E}(\ell_1, \ell_2) & \text{for } [\mathbf{skip}]^{\ell_1} \\
\mathbf{T}(\langle \ell_1, p, \ell_2 \rangle) & = & \mathbf{U}(\mathtt{x} \leftarrow a) \otimes \mathbf{E}(\ell_1, \ell_2) & \text{for } [\mathtt{x} \leftarrow a]^{\ell_1} \\
\mathbf{T}(\langle \ell, p, \ell_t \rangle) & = & \mathbf{P}(b = \mathbf{true}) \otimes \mathbf{E}(\ell, \ell_t) & \text{for } [b]^{\ell} \\
\mathbf{T}(\langle \ell, p, \ell_f \rangle) & = & \mathbf{P}(b = \mathbf{false}) \otimes \mathbf{E}(\ell, \ell_f) & \text{for } [b]^{\ell} \\
\mathbf{T}(\langle \ell, p_k, \ell_k \rangle) & = & \mathbf{I} \otimes \mathbf{E}(\ell, \ell_k) & \text{for } [\mathbf{choose}]^{\ell} \\
\mathbf{T}(\langle \ell, p, \ell \rangle & = & \mathbf{I} \otimes \mathbf{E}(\ell, \ell) & \text{for } [\mathbf{stop}]^{\ell}
\end{array}
$$

Projection Operators

Filtering out *relevant* probabilities, i.e. only for states/values which fulfill a certain condition. Use diagonal matrix:

$$(\mathbf{P})_{ii} = \begin{cases} 1 & \text{if condition holds for } c_i \in \textbf{Value} \\ 0 & \text{otherwise.} \end{cases}$$

$$\begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \end{pmatrix}^T \cdot \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 \\ d_2 \\ d_3 \\ 0 \\ d_5 \\ 0 \end{pmatrix}^T$$

Projection Operators

Filtering out *relevant* probabilities, i.e. only for states/values which fulfill a certain condition. Use diagonal matrix:

$$(\mathbf{P})_{ii} = \begin{cases} 1 & \text{if condition holds for } c_i \in \textbf{Value} \\ 0 & \text{otherwise.} \end{cases}$$

$$\begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \end{pmatrix}^T \cdot \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 \\ d_2 \\ d_3 \\ 0 \\ d_5 \\ 0 \end{pmatrix}^T$$

Tests and Filters

Select a certain value $c \in \mathbf{Value}_k$ for variable x_k:

$$(\mathbf{P}(c))_{ij} = \begin{cases} 1 & \text{if } i = c = j \\ 0 & \text{otherwise.} \end{cases}$$

Select a certain classical state $\sigma \in \mathbf{State}$:

$$\mathbf{P}(\sigma) = \bigotimes_{i=1}^{v} \mathbf{P}(\sigma(\mathrm{x}_i))$$

Select states where expression $e = a \mid b$ evaluates to c:

$$\mathbf{P}(e = c) = \sum_{\mathcal{E}(e)\sigma = c} \mathbf{P}(\sigma)$$

Tests and Filters

Select a certain value $c \in \mathbf{Value}_k$ for variable x_k:

$$(\mathbf{P}(c))_{ij} = \begin{cases} 1 & \text{if } i = c = j \\ 0 & \text{otherwise.} \end{cases}$$

Select a certain classical state $\sigma \in \mathbf{State}$:

$$\mathbf{P}(\sigma) = \bigotimes_{i=1}^{v} \mathbf{P}(\sigma(\mathrm{x}_i))$$

Select states where expression $e = a \mid b$ evaluates to c:

$$\mathbf{P}(e = c) = \sum_{\mathcal{E}(e)\sigma = c} \mathbf{P}(\sigma)$$

Tests and Filters

Select a certain value $c \in \mathbf{Value}_k$ for variable x_k:

$$(\mathbf{P}(c))_{ij} = \begin{cases} 1 & \text{if } i = c = j \\ 0 & \text{otherwise.} \end{cases}$$

Select a certain classical state $\sigma \in \mathbf{State}$:

$$\mathbf{P}(\sigma) = \bigotimes_{i=1}^{v} \mathbf{P}(\sigma(\mathrm{x}_i))$$

Select states where expression $e = a \mid b$ evaluates to c:

$$\mathbf{P}(e = c) = \sum_{\mathcal{E}(e)\sigma = c} \mathbf{P}(\sigma)$$

Updates

Modify the value of variable x_k to a constant $c \in \textbf{Value}_k$:

$$(\mathbf{U}(c))_{ij} = \begin{cases} 1 & \text{if } j = c \\ 0 & \text{otherwise.} \end{cases}$$

Set value of variable $\mathtt{x}_k \in \textbf{Var}$ to constant $c \in \textbf{Value}$:

$$\mathbf{U}(\mathtt{x}_k \leftarrow c) = \left(\bigotimes_{i=1}^{k-1} \mathbf{I} \right) \otimes \mathbf{U}(c) \otimes \left(\bigotimes_{i=k+1}^{v} \mathbf{I} \right)$$

Set value of variable $\mathtt{x}_k \in \textbf{Var}$ to value given by $e = a \mid b$:

$$\mathbf{U}(\mathtt{x}_k \leftarrow e) = \sum_{c} \mathbf{P}(e = c)\mathbf{U}(\mathtt{x}_k \leftarrow c)$$

Updates

Modify the value of variable x_k to a constant $c \in \mathbf{Value}_k$:

$$(\mathbf{U}(c))_{ij} = \begin{cases} 1 & \text{if } j = c \\ 0 & \text{otherwise.} \end{cases}$$

Set value of variable $\mathtt{x}_k \in$ **Var** to constant $c \in$ **Value**:

$$\mathbf{U}(\mathtt{x}_k \leftarrow c) = \left(\bigotimes_{i=1}^{k-1} \mathbf{I} \right) \otimes \mathbf{U}(c) \otimes \left(\bigotimes_{i=k+1}^{v} \mathbf{I} \right)$$

Set value of variable $\mathtt{x}_k \in$ **Var** to value given by $e = a \mid b$:

$$\mathbf{U}(\mathtt{x}_k \leftarrow e) = \sum_c \mathbf{P}(e = c)\mathbf{U}(\mathtt{x}_k \leftarrow c)$$

Updates

Modify the value of variable x_k to a constant $c \in \textbf{Value}_k$:

$$(\mathbf{U}(c))_{ij} = \begin{cases} 1 & \text{if } j = c \\ 0 & \text{otherwise.} \end{cases}$$

Set value of variable $\mathtt{x}_k \in \textbf{Var}$ to constant $c \in \textbf{Value}$:

$$\mathbf{U}(\mathtt{x}_k \leftarrow c) = \left(\bigotimes_{i=1}^{k-1} \mathbf{I} \right) \otimes \mathbf{U}(c) \otimes \left(\bigotimes_{i=k+1}^{v} \mathbf{I} \right)$$

Set value of variable $\mathtt{x}_k \in \textbf{Var}$ to value given by $e = a \mid b$:

$$\mathbf{U}(\mathtt{x}_k \leftarrow e) = \sum_c \mathbf{P}(e = c)\mathbf{U}(\mathtt{x}_k \leftarrow c)$$

An Example

if $[\mathrm{x} == 0]^a$ **then**
  $[x \leftarrow 0]^b$;
else
  $[x \leftarrow 1]^c$;
fi;
$[\textbf{stop}]^d$

$$
\begin{aligned}
\mathbf{T}(P) &= \mathbf{P}(\mathrm{x} = 0) \otimes \mathbf{E}(a, b) + \\
&+ \mathbf{P}(\mathrm{x} \neq 0) \otimes \mathbf{E}(a, c) + \\
&+ \mathbf{U}(x \leftarrow 0) \otimes \mathbf{E}(b, d) + \\
&+ \mathbf{U}(x \leftarrow 1) \otimes \mathbf{E}(c, d) + \\
&+ \mathbf{I} \otimes \mathbf{E}(d, d)
\end{aligned}
$$

An Example

if $[\mathrm{x} == 0]^a$ **then**
    $[x \leftarrow 0]^b$;
else
    $[x \leftarrow 1]^c$;
fi;
$[\mathbf{stop}]^d$

$$
\begin{aligned}
\mathbf{T}(P) \quad = \quad & \mathbf{P}(\mathrm{x} = 0) \otimes \mathbf{E}(a, b) + \\
+ \quad & \mathbf{P}(\mathrm{x} \neq 0) \otimes \mathbf{E}(a, c) + \\
+ \quad & \mathbf{U}(x \leftarrow 0) \otimes \mathbf{E}(b, d) + \\
+ \quad & \mathbf{U}(x \leftarrow 1) \otimes \mathbf{E}(c, d) + \\
+ \quad & \mathbf{I} \otimes \mathbf{E}(d, d)
\end{aligned}
$$

An Example

$$
\begin{aligned}
\mathbf{T}(P) &= \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \otimes \mathbf{E}(a,b) + \\
&+ \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \otimes \mathbf{E}(a,c) + \\
&+ \left(\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \otimes \mathbf{E}(b,d) \right) + \\
&+ \left(\begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \otimes \mathbf{E}(c,d) \right) + \\
&+ (\mathbf{I} \otimes \mathbf{E}(d,d))
\end{aligned}
$$

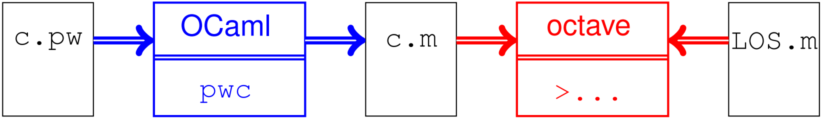
An Example

$$
\begin{aligned}
\mathbf{T}(P) \quad = \quad & \left(\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \right) \\
+ \quad & \left(\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \right) \\
+ \quad & \left(\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \right) \\
+ \quad & \left(\begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \right) \\
+ \quad & \left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \right)
\end{aligned}
$$

LOS and DTMC

$$
\begin{array}{lc}
\langle x = 0, [\mathbf{x} = 0] \rangle & \ldots \\
\langle x = 0, [\mathbf{x} := 0] \rangle & \ldots \\
\langle x = 0, [\mathbf{x} := 1] \rangle & \ldots \\
\langle x = 0, [\mathbf{stop}] \rangle & \ldots \\
\langle x = 1, [\mathbf{x} = 0] \rangle & \ldots \\
\langle x = 1, [\mathbf{x} := 0] \rangle & \ldots \\
\langle x = 1, [\mathbf{x} := 1] \rangle & \ldots \\
\langle x = 1, [\mathbf{stop}]^d \rangle & \ldots
\end{array}
\quad
\begin{pmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
$$

Written in OCaml produces an `octave` file `c.m` which specify the LOS matrices **U**, **P**, etc. for a pWhile program `c.pw`.



We can use the interactive interface of `octave` and definitions of standard operations in `LOS.m` to analyse matrices in `c.m`.

Exploiting sparse matrix representation to handle programs with about 3 to 5 variables, up to 10 values and program fragments with something like 20 lines/labels.

Factorial

Consider the program F for calculating the factorial of n:

```
var
  m : {0..2};
  n : {0..2};

begin
m := 1;
while (n>1) do
  m := m*n;
  n := n-1;
od;
stop; # looping
end
```

Control Flow and LOS for F

$$\mathit{flow}(F) \;=\; \{(1,1,2),(2,1,3),(3,1,4),(4,1,2),(2,1,5),(5,1,5)\}$$

$$\begin{aligned}
\mathbf{T}(F) \;=\; & \mathbf{U}(m \leftarrow 1) \otimes \mathbf{E}(1,2) + \\
& \mathbf{P}((n > 1)) \otimes \mathbf{E}(2,3) + \\
& \mathbf{U}(m \leftarrow (m * n)) \otimes \mathbf{E}(3,4) + \\
& \mathbf{U}(n \leftarrow (n - 1)) \otimes \mathbf{E}(4,2) + \\
& \mathbf{P}((n <= 1)) \otimes \mathbf{E}(2,5) + \\
& \mathbf{I} \otimes \mathbf{E}(5,5)
\end{aligned}$$

Control Flow and LOS for F

$$flow(F) = \{(1,1,2),(2,1,3),(3,1,4),(4,1,2),(2,1,5),(5,1,5)\}$$

$$\begin{aligned}
\mathbf{T}(F) = \; & \mathbf{U}(m \leftarrow 1) \otimes \mathbf{E}(1,2) + \\
& \mathbf{P}((n > 1)) \otimes \mathbf{E}(2,3) + \\
& \mathbf{U}(m \leftarrow (m * n)) \otimes \mathbf{E}(3,4) + \\
& \mathbf{U}(n \leftarrow (n - 1)) \otimes \mathbf{E}(4,2) + \\
& \mathbf{P}((n <= 1)) \otimes \mathbf{E}(2,5) + \\
& \mathbf{I} \otimes \mathbf{E}(5,5)
\end{aligned}$$

Introducing PAI

The matrix $\mathbf{T}(F)$ is very big already for small n.

n	$\dim(\mathbf{T}(F))$
2	45×45
3	140×140
4	625×625
5	3630×3630
6	25235×25235
7	201640×201640
8	1814445×1814445
9	18144050×18144050

We will show how we can drastically reduce the dimension of the LOS by using *Probabilistic Abstract Interpretation* (next talk).