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

<table>
<thead>
<tr>
<th>Analysis of probabilistic programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>• May give ‘incorrect’ answers.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Probabilistic analysis of (deterministic) programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Speculative vs conservative answers.</td>
</tr>
</tbody>
</table>
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.
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.
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 $2 \cdot n!$

\begin{verbatim}
m := 1;
while (n>1) do
    m := m*n;
    n := n-1;
end

m := 2;
while (n>1) do
    m := m*n;
    n := n-1;
end
\end{verbatim}
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.
Parity Analysis: Determine at every program point whether a variable is \textit{even} or \textit{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 \textit{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” \textit{even}.

The purpose of a probabilistic analysis is a formal derivation of this intuition about the parity of \(m\) when the program terminates.
**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.
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.
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.
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.
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.
Consider the abstract values $\bot \leq \text{even}; \bot \leq \text{odd}; \text{odd} \leq \top$ and $\text{even} \leq \top$.

1: $m \mapsto \top$, $n \mapsto \top$
2: $m \mapsto \text{even}$, $n \mapsto \top$
3: $m \mapsto \text{even}$, $n \mapsto \top$
4: $m \mapsto \text{even}$, $n \mapsto \top$
5: $m \mapsto \text{even}$, $n \mapsto \top$
Consider the abstract values $\bot \leq \text{even}; \bot \leq \text{odd}; \text{odd} \leq \top$ and $\text{even} \leq \top$.

1: $m \mapsto \top$, $n \mapsto \top$
2: $m \mapsto \text{even}$, $n \mapsto \top$
3: $m \mapsto \text{even}$, $n \mapsto \top$
4: $m \mapsto \text{even}$, $n \mapsto \top$
5: $m \mapsto \text{even}$, $n \mapsto \top$
Consider the abstract values \( \bot \leq \textsf{even}; \bot \leq \textsf{odd}; \textsf{odd} \leq \top \)
and \( \textsf{even} \leq \top \).

1: \( m \mapsto \top, \quad n \mapsto \top \)
2: \( m \mapsto \textsf{even}, \quad n \mapsto \top \)
3: \( m \mapsto \textsf{even}, \quad n \mapsto \top \)
4: \( m \mapsto \textsf{even}, \quad n \mapsto \top \)
5: \( m \mapsto \textsf{even}, \quad n \mapsto \top \)
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: $m \mapsto T, \quad n \mapsto T$
2: $m \mapsto \text{odd}, \quad n \mapsto T$
3: 
4: 
5: $m \mapsto \text{odd}, \quad n \mapsto T$
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. \( m \mapsto \top, \quad n \mapsto \top \)
2. \( m \mapsto \text{odd}, \quad n \mapsto \top \)
3.  
4.  
5. \( m \mapsto \text{odd}, \quad n \mapsto \top \)
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{align*}
1 & : m \mapsto \top, \quad n \mapsto \top \\
2 & : m \mapsto \text{odd}, \quad n \mapsto \top \\
3 & : \\
4 & : \\
5 & : m \mapsto \text{odd}, \quad n \mapsto \top \\
\end{align*}
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.

<table>
<thead>
<tr>
<th>Line</th>
<th>( m )</th>
<th>( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \top )</td>
<td>( \top )</td>
</tr>
<tr>
<td>2</td>
<td><strong>odd</strong></td>
<td>( \top )</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td><strong>odd</strong></td>
<td>( \top )</td>
</tr>
</tbody>
</table>
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.
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.
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.
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
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.
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 \textit{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.
**Classical** RD analysis assumes that all edges of a flow graph can be traversed. This assumption may not be true in practice.

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

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

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

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

```c
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.
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 whether to perform a certain code optimisation or not.
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 whether 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 whether 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 whether to perform a certain code optimisation or not.
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 whether to perform a certain code optimisation or not.
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.
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.
Full programs contain optional variable declarations:

\[
P ::= \begin{align*}
\text{begin } S \text{ end} \\
\mid \text{var } D \text{ begin } S \text{ end}
\end{align*}
\]

Declarations are of the form:

\[
r ::= \begin{align*}
\text{bool} \\
\mid \text{int} \\
\mid \{ c_1, \ldots, c_n \} \\
\mid \{ c_1 .. c_n \}
\end{align*}
\]

\[
D ::= \begin{align*}
v : r \\
\mid v : r ; D
\end{align*}
\]

with \(c_i\) (integer) constants and \(r\) denoting ranges.
Full programs contain optional variable declarations:

\[
P ::= \text{begin } S \text{ end} \\
    \quad | \text{var } D \text{ begin } S \text{ end}
\]

Declarations are of the form:

\[
r ::= \text{bool} \\
    \quad | \text{int} \\
    \quad | \{ c_1, \ldots, c_n \} \\
    \quad | \{ c_1 .. c_n \}
\]

\[
D ::= v : r \\
    \quad | v : r ; D
\]

with \( c_i \) (integer) constants and \( r \) denoting ranges.
The syntax of statements $S$ is as follows:

$$S ::= \text{stop} \mid \text{skip} \mid \nu ::= a \mid S_1 ; S_2 \mid \text{choose } p_1 : S_1 \text{ or } p_2 : S_2 \text{ ro} \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi} \mid \text{while } b \text{ do } S \text{ od}$$

Where the $p_i$ are constants, representing choice probabilities.
The syntax of statements $S$ is as follows:

$$S ::= [\text{stop}]^\ell$$

$$[\text{skip}]^\ell$$

$$[v := a]^\ell$$

$$S_1 ; S_2$$

$$\text{choose}^\ell p_1 : S_1 \text{ or } p_2 : S_2 \text{ ro}$$

$$\text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 \text{ fi}$$

$$\text{while } [b]^\ell \text{ do } S \text{ od}$$

Where the $p_i$ are constants, representing choice probabilities.
\( \sigma \ni \textbf{State} = \text{Var} \rightarrow \mathbb{Z} \uplus \mathbb{B} \)

Evaluation \( \mathcal{E} \) of expressions \( e \) in state \( \sigma \):

\[
\begin{align*}
\mathcal{E}(n)\sigma &= n \\
\mathcal{E}(v)\sigma &= \sigma(v) \\
\mathcal{E}(a_1 \otimes a_2)\sigma &= \mathcal{E}(a_1)\sigma \otimes \mathcal{E}(a_2)\sigma \\
\mathcal{E}(\text{true})\sigma &= \text{tt} \\
\mathcal{E}(\text{false})\sigma &= \text{ff} \\
\mathcal{E}(\text{not } b)\sigma &= \neg \mathcal{E}(b)\sigma \\
\ldots &= \ldots
\end{align*}
\]
Evaluation of Expressions

\[ \sigma \ni \textbf{State} = \text{Var} \rightarrow \mathbb{Z} \sqcup \mathbb{B} \]

Evaluation \( \mathcal{E} \) of expressions \( e \) in state \( \sigma \):

\[
\begin{align*}
\mathcal{E}(n)\sigma &= n \\
\mathcal{E}(v)\sigma &= \sigma(v) \\
\mathcal{E}(a_1 \odot a_2)\sigma &= \mathcal{E}(a_1)\sigma \odot \mathcal{E}(a_2)\sigma \\
\mathcal{E}(\text{true})\sigma &= \text{tt} \\
\mathcal{E}(\text{false})\sigma &= \text{ff} \\
\mathcal{E}(\text{not } b)\sigma &= \neg \mathcal{E}(b)\sigma \\
\ldots &= \ldots
\end{align*}
\]
Evaluation of Expressions

\[ \sigma \ni \text{State} = \text{Var} \rightarrow \mathbb{Z} \cup \mathbb{B} \]

Evaluation \( \mathcal{E} \) of expressions \( e \) in state \( \sigma \):

\[
\begin{align*}
\mathcal{E}(n)_{\sigma} &= n \\
\mathcal{E}(v)_{\sigma} &= \sigma(v) \\
\mathcal{E}(a_1 \odot a_2)_{\sigma} &= \mathcal{E}(a_1)_{\sigma} \odot \mathcal{E}(a_2)_{\sigma} \\
\mathcal{E}(\text{true})_{\sigma} &= \text{tt} \\
\mathcal{E}(\text{false})_{\sigma} &= \text{ff} \\
\mathcal{E}(\text{not } b)_{\sigma} &= \neg \mathcal{E}(b)_{\sigma} \\
\ldots &= \ldots
\end{align*}
\]
\( \langle \text{skip}, \sigma \rangle \Rightarrow_1 \langle \text{stop}, \sigma \rangle \)

\( \langle \text{stop}, \sigma \rangle \Rightarrow_1 \langle \text{stop}, \sigma \rangle \)

\( \langle \nu := e, \sigma \rangle \Rightarrow_1 \langle \text{stop}, \sigma[\nu \mapsto \mathcal{E}(e)\sigma] \rangle \)

\[
\begin{align*}
R3_1 & : \langle S_1, \sigma \rangle \Rightarrow_p \langle S'_1, \sigma' \rangle \\
& \Rightarrow \langle S_1; S_2, \sigma \rangle \Rightarrow_p \langle S'_1; S_2, \sigma' \rangle
\end{align*}
\]

\[
\begin{align*}
R3_2 & : \langle S_1, \sigma \rangle \Rightarrow_p \langle \text{stop}, \sigma' \rangle \\
& \Rightarrow \langle S_1; S_2, \sigma \rangle \Rightarrow_p \langle S_2, \sigma' \rangle
\end{align*}
\]
\[ R_4_1 \langle \text{choose } p_1 : S_1 \text{ or } p_2 : S_2, \sigma \rangle \Rightarrow p_1 \langle S_1, \sigma \rangle \]

\[ R_4_2 \langle \text{choose } p_1 : S_1 \text{ or } p_2 : S_2, \sigma \rangle \Rightarrow p_2 \langle S_2, \sigma \rangle \]

\[ R_5_1 \langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma \rangle \Rightarrow \langle S_1, \sigma \rangle \quad \text{if } \mathcal{E}(b)\sigma = \text{tt} \]

\[ R_5_2 \langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma \rangle \Rightarrow \langle S_2, \sigma \rangle \quad \text{if } \mathcal{E}(b)\sigma = \text{ff} \]

\[ R_6_1 \langle \text{while } b \text{ do } S, \sigma \rangle \Rightarrow \langle S; \text{ while } b \text{ do } S, \sigma \rangle \quad \text{if } \mathcal{E}(b)\sigma = \text{tt} \]

\[ R_6_2 \langle \text{while } b \text{ do } S, \sigma \rangle \Rightarrow \langle \text{stop}, \sigma \rangle \quad \text{if } \mathcal{E}(b)\sigma = \text{ff} \]
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.
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.
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.
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.
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.
A DTMC is a tuple \((S, P, \nu_{in})\) where

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

Paths in a DTMC are maximal (i.e. infinite) in the underlying directed graph.
A DTMC is a tuple \((S, P, \nu_{\text{in}})\) where

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

Paths in a DTMC are maximal (i.e. infinite) in the underlying directed graph.
A DTMC is a tuple $(S, P, \nu_{in})$ where

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

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

A DTMC is a tuple \((S, P, \nu_{in})\) where

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

Paths in a DTMC are maximal (i.e. infinite) in the underlying directed graph.
Given a pWhile program, consider any enumeration of all its configurations (= pairs of statements and state) $C_1, C_2, C_3, \ldots \in \text{Conf}$. Then

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

is the generator of a Discrete Time Markov Chain.

Transitions are implemented as

$$d_n \cdot T = \sum_i (d_n)_i \cdot T_{ij} = d_{n+1}$$

where $d_i$ is the probability distribution over $\text{Conf}$ at the $i$th step.
Given a \texttt{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

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

is the generator of a Discrete Time Markov Chain.

Transitions are implemented as

$$ d_n \cdot T = \sum_i (d_n)_i \cdot T_{ij} = d_{n+1} $$

where $d_i$ is the probability distribution over $\textbf{Conf}$ at the $i$th step.
Given a pWhile program, consider any enumeration of all its configurations (= pairs of statements and state) \( C_1, C_2, C_3, \ldots \in \text{Conf} \). Then

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

is the generator of a Discrete Time Markov Chain.

Transitions are implemented as

\[
d_n \cdot T = \sum_i (d_n)_i \cdot T_{ij} = d_{n+1}
\]

where \( d_i \) is the probability distribution over \( \text{Conf} \) at the \( i \)th step.
Given a \texttt{pWhile} program, consider any enumeration of all its configurations (= pairs of statements and state) \( C_1, C_2, C_3, \ldots \in \text{Conf} \). Then

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

is the generator of a Discrete Time Markov Chain.

Transitions are implemented as

\[
d_n \cdot T = \sum_i (d_n)_i \cdot T_{ij} = d_{n+1}
\]

where \( d_i \) is the probability distribution over \text{Conf} at the \( i \)th step.
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

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

is the generator of a Discrete Time Markov Chain.

Transitions are implemented as

\[
d_n \cdot T = \sum_i (d_n)_i \cdot T_{ij} = d_{n+1}
\]

where \( d_i \) is the probability distribution over \( \textbf{Conf} \) at the \( i \)th step.
Given a `pWhile` program, consider any enumeration of all its configurations (= pairs of statements and state) $C_1, C_2, C_3, \ldots \in \text{Conf}$. Then

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

is the generator of a Discrete Time Markov Chain.

Transitions are implemented as

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

where $\mathbf{d}_i$ is the probability distribution over $\text{Conf}$ at the $i$th step.
Let us investigate the possible transitions of the following labelled program (with $x \in \{0, 1\}$):

```plaintext
if $x = 0$ then
  [$x := 0$];
else
  [$x := 1$];
fi;
[stop]
```
Example DTMC

\[
\langle x = 0, [x = 0] \rangle \ldots \\
\langle x = 0, [x := 0] \rangle \ldots \\
\langle x = 0, [x := 1] \rangle \ldots \\
\langle x = 0, [\text{stop}] \rangle \ldots \\
\langle x = 1, [x = 0] \rangle \ldots \\
\langle x = 1, [x := 0] \rangle \ldots \\
\langle x = 1, [x := 1] \rangle \ldots \\
\langle x = 1, [\text{stop}] \rangle \ldots
\]

\[
\begin{pmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
\]
Example Transition

\[
\begin{pmatrix}
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\end{pmatrix}
\]

We get: \((0 0 0 0 0 0 0 0 1)\).
Example Transition

\[
\begin{pmatrix}
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{pmatrix}
\]

We get: \( \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \).
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 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 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 $x$-th power of the number stored in $y$:

\[
\begin{align*}
\text{[ } z & := 1 \text{]}; \\
\text{while } [ x > 1 ] & \text{ do (}
\quad [ z := z*y ]; \\
\quad [ x := x-1 ];
\end{align*}
\]

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

\[
flow(power) = \{(1, 2), (2, 3), (3, 4), (4, 2)\}
\]
Consider the following program, \( \text{power} \), computing the \( x \)-th power of the number stored in \( y \):

\[
[ z := 1 ]^1; \\
\textbf{while } [ x > 1 ]^2 \textbf{ do } ( \\
[ z := z*y ]^3; \\
[ x := x-1 ]^4); 
\]

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

\( \text{flow}(\text{power}) = \{(1, 2), (2, 3), (3, 4), (4, 2)\} \)
Flow Graph

\[
\begin{align*}
[z := 1]^1 & \\
[x > 0]^2 & \quad \text{no} \\
[z := z \times y]^3 & \quad \text{yes} \\
[x := x - 1]^4 &
\end{align*}
\]
Consider the following labelled program:

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

Its probabilistic control flow is given by:

\[ \text{flow}(P) = \{ \langle 1, 1, 2 \rangle, \langle 1, 1, 5 \rangle, \langle 2, \frac{1}{3}, 3 \rangle, \langle 2, \frac{2}{3}, 4 \rangle, \langle 3, 1, 1 \rangle, \langle 4, 1, 1 \rangle \}. \]
Consider the following labelled program:

1: while \([z < 100]\) do
2: \([\text{choose}] 1/3 : [x := 3] \text{ or } 2/3 : [x := 1] \) ro
3: od
4: \([\text{stop}]\)

Its probabilistic control flow is given by:

\[ \text{flow}(P) = \{ \langle 1, 1, 2 \rangle, \langle 1, 1, 5 \rangle, \langle 2, \frac{1}{3}, 3 \rangle, \langle 2, \frac{2}{3}, 4 \rangle, \langle 3, 1, 1 \rangle, \langle 4, 1, 1 \rangle \}. \]
\[
\begin{align*}
\text{init}([\text{skip}]^\ell) &= \ell \\
\text{init}([\text{stop}]^\ell) &= \ell \\
\text{init}([\nu : = e]^\ell) &= \ell \\
\text{init}(S_1; S_2) &= \text{init}(S_1) \\
\text{init}([\text{choose}]^\ell \ p_1 : S_1 \ \text{or} \ p_2 : S_2) &= \ell \\
\text{init}([\text{if}]^\ell \ \text{then} \ S_1 \ \text{else} \ S_2) &= \ell \\
\text{init}([\text{while}]^\ell \ \text{do} \ S) &= \ell
\end{align*}
\]
\[ \text{\texttt{final}}([\texttt{skip}]^\ell) = \{\ell\} \]
\[ \text{\texttt{final}}([\texttt{stop}]^\ell) = \{\ell\} \]
\[ \text{\texttt{final}}([\texttt{v := e}]^\ell) = \{\ell\} \]
\[ \text{\texttt{final}}(S_1; S_2) = \text{\texttt{final}}(S_2) \]
\[ \text{\texttt{final}}([\texttt{choose}]^\ell \ p_1 : S_1 \texttt{ or } p_2 : S_2) = \text{\texttt{final}}(S_1) \cup \text{\texttt{final}}(S_2) \]
\[ \text{\texttt{final}}([\texttt{if}]^\ell \ [b] \texttt{ then } S_1 \texttt{ else } S_2) = \text{\texttt{final}}(S_1) \cup \text{\texttt{final}}(S_2) \]
\[ \text{\texttt{final}}([\texttt{while}]^\ell \ [b] \texttt{ do } S) = \{\ell\} \]
The probabilistic control flow is defined by the function:

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

- \(flow([\text{skip}]^\ell) = \emptyset\)
- \(flow([\text{stop}]^\ell) = \{(\ell, 1, \ell)\}\)
- \(flow([\text{v} := e]^\ell) = \emptyset\)
- \(flow(S_1; S_2) = flow(S_1) \cup flow(S_2) \cup \{(\ell, 1, \text{init}(S_2)) \mid \ell \in \text{final}(S_1)\}\)
The probabilistic control flow is defined by the function:

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

- \(\text{flow}([\text{skip}]^\ell) = \emptyset\)
- \(\text{flow}([\text{stop}]^\ell) = \{\langle \ell, 1, \ell \rangle\}\)
- \(\text{flow}([v := e]^\ell) = \emptyset\)
- \(\text{flow}(S_1; S_2) = \text{flow}(S_1) \cup \text{flow}(S_2) \cup \{((\ell, 1, \text{init}(S_2))) | \ell \in \text{final}(S_1)\}\)
Flow II — Control Transfer

\[
\text{flow}(\text{[choose]}^\ell p_1 : S_1 \text{ or } p_2 : S_2) = \text{flow}(S_1) \cup \text{flow}(S_2) \cup \\
\cup \{(\ell, p_1, \text{init}(S_1)), (\ell, p_2, \text{init}(S_2))\}
\]

\[
\text{flow}(\text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2) = \text{flow}(S_1) \cup \text{flow}(S_2) \cup \\
\cup \{(\ell, 1, \text{init}(S_1)), (\ell, 1, \text{init}(S_2))\}
\]

\[
\text{flow}(\text{while } [b]^\ell \text{ do } S) = \text{flow}(S) \cup \\
\cup \{(\ell, 1, \text{init}(S))\} \\
\cup \{(\ell', 1, \ell) | \ell' \in \text{final}(S)\}
\]
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 matrix representation of the SOS semantics of a \texttt{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 \textit{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

\[
\text{Dist}(\text{State} \times \text{Lab}) \subseteq \mathcal{V}(\text{State} \times \text{Lab}).
\]

Assuming \( v = |\text{Var}| \) finite,

\[
\text{State} = (\mathbb{Z} + \mathbb{B})^v = \text{Value}_1 \times \text{Value}_2 \ldots \times \text{Value}_v
\]

with \( \text{Value}_i = \mathbb{Z} \) or \( \mathbb{B} \).

Thus, we can represent the space of configurations as

\[
\text{Dist}(\text{Value}_1 \times \ldots \times \text{Value}_v \times \text{Lab}) \subseteq \mathcal{V}(\text{Value}_1) \otimes \ldots \otimes \mathcal{V}(\text{Value}_v) \otimes \mathcal{V}(\text{Lab}).
\]
For a **pWhile** program $P$ we can identify configurations with elements in

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

Assuming $\nu = |\text{Var}|$ finite,

$$\text{State} = (\mathbb{Z} + \mathbb{B})^\nu = \text{Value}_1 \times \text{Value}_2 \ldots \times \text{Value}_\nu$$

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

Thus, we can represent the space of configurations as

$$\text{Dist}(\text{Value}_1 \times \ldots \times \text{Value}_\nu \times \text{Lab}) \subseteq \mathcal{V}(\text{Value}_1) \otimes \ldots \otimes \mathcal{V}(\text{Value}_\nu) \otimes \mathcal{V}(\text{Lab}).$$
The Space of Configurations

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

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

Assuming $\nu = |\text{Var}|$ finite,

$$\text{State} = (\mathbb{Z} + \mathbb{B})^\nu = \text{Value}_1 \times \text{Value}_2 \ldots \times \text{Value}_\nu$$

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

Thus, we can represent the space of configurations as

$$\text{Dist}(\text{Value}_1 \times \ldots \times \text{Value}_\nu \times \text{Lab}) \subseteq \mathcal{V}(\text{Value}_1) \otimes \ldots \otimes \mathcal{V}(\text{Value}_\nu) \otimes \mathcal{V}(\text{Lab}).$$
The Space of Configurations

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

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

Assuming $\nu = |\text{Var}|$ finite,

$$\text{State} = (\mathbb{Z} + \mathbb{B})^\nu = \text{Value}_1 \times \text{Value}_2 \ldots \times \text{Value}_\nu$$

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

Thus, we can represent the space of configurations as

$$\text{Dist}((\text{Value}_1 \times \ldots \times \text{Value}_\nu \times \text{Lab})) \subseteq \mathcal{V}(\text{Value}_1) \otimes \ldots \otimes \mathcal{V}(\text{Value}_\nu) \otimes \mathcal{V}((\text{Lab}))$$
Given a $n \times m$ matrix $A$ and a $k \times l$ matrix $B$:

\[
A = \begin{pmatrix}
a_{11} & \cdots & a_{1m} \\
\vdots & \ddots & \vdots \\
a_{n1} & \cdots & a_{nm}
\end{pmatrix} \quad B = \begin{pmatrix}
b_{11} & \cdots & b_{1l} \\
\vdots & \ddots & \vdots \\
b_{k1} & \cdots & b_{kl}
\end{pmatrix}
\]

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

\[
A \otimes B = \begin{pmatrix}
a_{11}B & \cdots & a_{1m}B \\
\vdots & \ddots & \vdots \\
a_{n1}B & \cdots & a_{nm}B
\end{pmatrix}
\]

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 $A$ and a $k \times l$ matrix $B$:

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

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

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

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 $A$ and a $k \times l$ matrix $B$:

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

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

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

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*

\[ T(P) = \sum_{\langle i, p_{ij}, j \rangle \in \text{flow}(P)} p_{ij} \cdot T(\ell_i, \ell_j), \]

where

\[ T(\ell_i, \ell_j) = N \otimes E(\ell_i, \ell_j), \]

with \( N \) an operator representing a state update while the second factor realises the transfer of control from label \( \ell_i \) to label \( \ell_j \).
A Linear Operator based on \textit{flow}

\[
T(P) = \sum_{\langle i, p_{ij}, j \rangle \in \text{flow}(P)} p_{ij} \cdot T(\ell_i, \ell_j),
\]

where

\[
T(\ell_i, \ell_j) = N \otimes E(\ell_i, \ell_j),
\]

with \(N\) an operator representing a state update while the second factor realises the transfer of control from label \(\ell_i\) to label \(\ell_j\).
A Linear Operator based on \textit{flow}

\[
T(P) = \sum_{\langle i, p_{ij}, j \rangle \in \text{flow}(P)} p_{ij} \cdot T(\ell_i, \ell_j),
\]

where

\[
T(\ell_i, \ell_j) = N \otimes E(\ell_i, \ell_j),
\]

with \textbf{N} an operator representing a state update while the second factor realises the transfer of control from label \(\ell_i\) to label \(\ell_j\).
Transfer Operators

\[
\begin{align*}
T(⟨\ell_1, p, \ell_2⟩) &= I \otimes E(\ell_1, \ell_2) & \text{for [skip]}^{\ell_1} \\
T(⟨\ell_1, p, \ell_2⟩) &= U(x ← a) \otimes E(\ell_1, \ell_2) & \text{for [x ← a]}^{\ell_1} \\
T(⟨\ell, p, \ell_t⟩) &= P(b = \text{true}) \otimes E(\ell, \ell_t) & \text{for [b]}^{\ell} \\
T(⟨\ell, p, \ell_f⟩) &= P(b = \text{false}) \otimes E(\ell, \ell_f) & \text{for [b]}^{\ell} \\
T(⟨\ell, p_k, \ell_k⟩) &= I \otimes E(\ell, \ell_k) & \text{for [choose]}^{\ell} \\
T(⟨\ell, p, \ell⟩) &= I \otimes E(\ell, \ell) & \text{for [stop]}^{\ell}
\end{align*}
\]
Projection Operators

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

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

\[
\begin{pmatrix}
d_1 \\
d_2 \\
d_3 \\
d_4 \\
d_5 \\
d_6
\end{pmatrix}^T 
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix} = 
\begin{pmatrix}
0 \\
d_2 \\
d_3 \\
0 \\
d_5 \\
0
\end{pmatrix}^T
\]
Projection Operators

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

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

\[
\begin{pmatrix}
  d_1 \\
  d_2 \\
  d_3 \\
  d_4 \\
  d_5 \\
  d_6
\end{pmatrix}^T \cdot 
\begin{pmatrix}
  0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 & 0 & 0 \\
  0 & 0 & 1 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 1 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix} = 
\begin{pmatrix}
  0 \\
  d_2 \\
  d_3 \\
  0 \\
  d_5 \\
  0
\end{pmatrix}^T
\]
Tests and Filters

Select a certain value \( c \in \text{Value}_k \) for variable \( x_k \):

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

Select a certain classical state \( \sigma \in \text{State} \):

\[
P(\sigma) = \bigotimes_{i=1}^{v} P(\sigma(x_i))
\]

Select states where expression \( e = a | b \) evaluates to \( c \):

\[
P(e = c) = \sum_{\varepsilon(e)\sigma = c} P(\sigma)
\]
Tests and Filters

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

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

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

$$P(\sigma) = \bigotimes_{i=1}^{v} P(\sigma(x_i))$$

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

$$P(e = c) = \sum_{\epsilon(e)\sigma = c} P(\sigma)$$
Tests and Filters

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

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

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

$$P(\sigma) = \bigotimes_{i=1}^{v} P(\sigma(x_i))$$

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

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

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

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

Set value of variable $x_k \in \text{Var}$ to constant $c \in \text{Value}$:

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

Set value of variable $x_k \in \text{Var}$ to value given by $e = a \mid b$:

$$ U(x_k \leftarrow e) = \sum_c P(e = c) U(x_k \leftarrow c) $$
Updates

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

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

Set value of variable $x_k \in \text{Var}$ to constant $c \in \text{Value}$:

$$U(x_k \leftarrow c) = \left( \bigotimes_{i=1}^{k-1} I \right) \otimes U(c) \otimes \left( \bigotimes_{i=k+1}^V I \right)$$

Set value of variable $x_k \in \text{Var}$ to value given by $e = a \mid b$:

$$U(x_k \leftarrow e) = \sum_c P(e = c) U(x_k \leftarrow c)$$
Updates

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

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

Set value of variable $x_k \in \text{Var}$ to constant $c \in \text{Value}$:

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

Set value of variable $x_k \in \text{Var}$ to value given by $e = a \mid b$:

$$U(x_k \leftarrow e) = \sum_c P(e = c) U(x_k \leftarrow c)$$
An Example

\[
\begin{align*}
\text{if } [x == 0]^a \text{ then} & \quad T(P) = P(x = 0) \otimes E(a, b) + \\
\quad [x \leftarrow 0]^b; & \quad + \quad P(x \neq 0) \otimes E(a, c) + \\
\text{else} & \quad + \quad U(x \leftarrow 0) \otimes E(b, d) + \\
\quad [x \leftarrow 1]^c; & \quad + \quad U(x \leftarrow 1) \otimes E(c, d) + \\
\text{fi;} & \quad + \quad I \otimes E(d, d)
\end{align*}
\]

\]
An Example

\[
\text{if } [x == 0]^a \text{ then } [x \leftarrow 0]^b;
\]

\[
\text{else } [x \leftarrow 1]^c;\fi
\]

\[
[\text{stop}]^d
\]

\[
T(P) = P(x = 0) \otimes E(a, b) +
\]

\[
+ P(x \neq 0) \otimes E(a, c) +
\]

\[
+ U(x \leftarrow 0) \otimes E(b, d) +
\]

\[
+ U(x \leftarrow 1) \otimes E(c, d) +
\]

\[
+ I \otimes E(d, d)
\]
$T(P) = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \otimes E(a, b) + \\
+ \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \otimes E(a, c) + \\
+ \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \otimes E(b, d) + \\
+ \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \otimes E(c, d) + \\
+ (I \otimes E(d, d))$
An Example

\[
T(P) = \left( \left( \begin{array}{cc} 1 & 0 \\ 0 & 0 \end{array} \right) \otimes \left( \begin{array}{cccc} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array} \right) \right) \\
+ \left( \left( \begin{array}{cc} 0 & 0 \\ 0 & 1 \end{array} \right) \otimes \left( \begin{array}{cccc} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array} \right) \right) \\
+ \left( \left( \begin{array}{cc} 1 & 0 \\ 1 & 0 \end{array} \right) \otimes \left( \begin{array}{cccc} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{array} \right) \right) \\
+ \left( \left( \begin{array}{cc} 0 & 1 \\ 0 & 1 \end{array} \right) \otimes \left( \begin{array}{cccc} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right) \right) \\
+ \left( \left( \begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array} \right) \otimes \left( \begin{array}{cccc} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right) \right)
\]
LOS and DTMC

\[
\begin{align*}
\langle x = 0, [x = 0] \rangle & \quad \ldots \\
\langle x = 0, [x := 0] \rangle & \quad \ldots \\
\langle x = 0, [x := 1] \rangle & \quad \ldots \\
\langle x = 0, \text{[stop]} \rangle & \quad \ldots \\
\langle x = 1, [x = 0] \rangle & \quad \ldots \\
\langle x = 1, [x := 0] \rangle & \quad \ldots \\
\langle x = 1, [x := 1] \rangle & \quad \ldots \\
\langle x = 1, \text{[stop]}^d \rangle & \quad \ldots \\
\end{align*}
\]

\[
\begin{pmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
\]
Written in OCaml produces an \texttt{octave} file \texttt{c.m} which specify the LOS matrices $U$, $P$, etc. for a pWhile program \texttt{c.pw}.

We can use the interactive interface of \texttt{octave} and definitions of standard operations in \texttt{LOS.m} to analyse matrices in \texttt{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.
Consider the program $F$ for calculating the factorial of $n$:

```plaintext
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 $F$

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

$$T(F) = U(m \leftarrow 1) \otimes E(1, 2) + P((n > 1)) \otimes E(2, 3) + U(m \leftarrow (m * n)) \otimes E(3, 4) + U(n \leftarrow (n - 1)) \otimes E(4, 2) + P((n <= 1)) \otimes E(2, 5) + I \otimes E(5, 5)$$
Control Flow and LOS for $F$

\[
\text{flow}(F) = \{(1, 1, 2), (2, 1, 3), (3, 1, 4), (4, 1, 2), (2, 1, 5), (5, 1, 5)\}
\]

\[
\text{T}(F) = \text{U}(m \leftarrow 1) \otimes \text{E}(1, 2) + \\
\text{P}((n > 1)) \otimes \text{E}(2, 3) + \\
\text{U}(m \leftarrow (m \ast n)) \otimes \text{E}(3, 4) + \\
\text{U}(n \leftarrow (n - 1)) \otimes \text{E}(4, 2) + \\
\text{P}((n \leq 1)) \otimes \text{E}(2, 5) + \\
\text{I} \otimes \text{E}(5, 5)
\]
Introducing PAI

The matrix $T(F)$ is very big already for small $n$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\dim(T(F))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$45 \times 45$</td>
</tr>
<tr>
<td>3</td>
<td>$140 \times 140$</td>
</tr>
<tr>
<td>4</td>
<td>$625 \times 625$</td>
</tr>
<tr>
<td>5</td>
<td>$3630 \times 3630$</td>
</tr>
<tr>
<td>6</td>
<td>$25235 \times 25235$</td>
</tr>
<tr>
<td>7</td>
<td>$201640 \times 201640$</td>
</tr>
<tr>
<td>8</td>
<td>$1814445 \times 1814445$</td>
</tr>
<tr>
<td>9</td>
<td>$18144050 \times 18144050$</td>
</tr>
</tbody>
</table>

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