Introduction to Directed Algebraic Topology with a view towards modellingConcurrency

Mathematical Structures of Computations - Lyon 2014

Emmanuel Haucourt

CEA-Tech, NanolInnov

The 27th of January
Summary

Different kinds of parallelism

Virtual Machines
  Middle-End Representation
  Execution model

Concurrency
  Generalizing graphs
  Control flow precubical set
  The extended PV language
Distributed computation

- Variable amount of available resources
- Variable population of parallel processes
- e.g. SETI@home, Bitcoin, e-shopping
- Usual requirements: availability, coherence, fault tolerance
Fine grain parallelism

- Constant amount of available resources
- Constant population of parallel processes
- e.g. control-command, graphic rendering
- Usual requirements: deterministic output, nonblocking, as fast as possible
Expressions and values

$\mathcal{V}$: variables $\quad \mathcal{E}$: expressions built on the following operators

<table>
<thead>
<tr>
<th>$\mathcal{V}$</th>
<th>content of $v \in \mathcal{V}$</th>
<th>$x \in \mathbb{R}$</th>
<th>constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\land$</td>
<td>minimum</td>
<td>$\lor$</td>
<td>maximum</td>
</tr>
<tr>
<td>$+$</td>
<td>addition</td>
<td>$-$</td>
<td>subtraction</td>
</tr>
<tr>
<td>$\ast$</td>
<td>multiplication</td>
<td>$/$</td>
<td>division</td>
</tr>
<tr>
<td>$\leq$</td>
<td>less or equal</td>
<td>$\geq$</td>
<td>greater of equal</td>
</tr>
<tr>
<td>$&lt;$</td>
<td>strictly less</td>
<td>$&gt;$</td>
<td>strictly greater</td>
</tr>
<tr>
<td>$\neg$</td>
<td>complement</td>
<td>$=$</td>
<td>equal</td>
</tr>
<tr>
<td>$\bot$</td>
<td></td>
<td></td>
<td>bottom</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nullary</th>
<th>unary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot$, $x \in \mathbb{R}$, $v \in \mathcal{V}$</td>
<td>$\neg$</td>
</tr>
</tbody>
</table>

binary

$\land$, $\lor$, $+$, $-$, $\ast$, $/$, $<$, $>$, $\leq$, $\geq$, $=$
Interpretation of expressions

\([-\cdot] : (\mathcal{V} \rightarrow \mathbb{R}_\bot) \rightarrow \mathcal{E} \rightarrow \mathbb{R}_\bot\)

- distribution: $\delta : \mathcal{V} \rightarrow \mathbb{R}_\bot$
Interpretation of expressions

\[ [-] : (\mathcal{V} \rightarrow \mathbb{R}_\perp) \rightarrow \mathcal{E} \rightarrow \mathbb{R}_\perp \]

- distribution: \( \delta : \mathcal{V} \rightarrow \mathbb{R}_\perp \)
- \( [\mathcal{V}]_\delta = \delta(\mathcal{V}) \)
Interpretation of expressions

\[ [-] : (V \rightarrow \mathbb{R}_\perp) \rightarrow \mathcal{E} \rightarrow \mathbb{R}_\perp \]

- \textbf{distribution: } \delta : V \rightarrow \mathbb{R}_\perp
  
- \[ [v]_\delta = \delta(v) \]

- \text{0 stands for false any value in } \mathbb{R} \setminus \{0\} \text{ stands for true}
Interpretation of expressions

\[ [-] : (\mathcal{V} \rightarrow \mathbb{R}_\bot) \rightarrow \mathcal{E} \rightarrow \mathbb{R}_\bot \]

- distribution: \( \delta : \mathcal{V} \rightarrow \mathbb{R}_\bot \)

- \( [v]_\delta = \delta(v) \)

- 0 stands for false any value in \( \mathbb{R} \setminus \{0\} \) stands for true

- \( [-] : \mathbb{R}_\bot \rightarrow \mathbb{R}_\bot \),
  \( [-](0) = 1 \), and
  \( [-](x) = 0 \) for all \( x \in \mathbb{R} \setminus \{0\} \)
Interpretation of expressions

\[ [-] : (\mathcal{V} \rightarrow \mathbb{R}_\perp) \rightarrow \mathcal{E} \rightarrow \mathbb{R}_\perp \]

- distribution: \( \delta : \mathcal{V} \rightarrow \mathbb{R}_\perp \)
- \( [-]_\delta = \delta(\nu) \)
- 0 stands for false any value in \( \mathbb{R} \setminus \{0\} \) stands for true
- \( [-] : \mathbb{R}_\perp \rightarrow \mathbb{R}_\perp , \)
  \( [-](0) = 1 \), and
  \( [-](x) = 0 \) for all \( x \in \mathbb{R} \setminus \{0\} \)
- \( [-] e = \perp \) for all expression \( e \) in which \( \perp \) occurs
Interpretation of actions

\([-\cdot] : (\mathcal{V} \rightarrow \mathcal{R}_\perp) \rightarrow \mathcal{V} \rightarrow \mathcal{E} \rightarrow (\mathcal{V} \rightarrow \mathcal{R}_\perp)\)

- \(v\): variable, \(e\): expression, \(\delta\): distribution
- \(v := e\) is called an action, \(\mathcal{A}\) set of all the actions
Interpretation of actions

\[[-] : (\mathcal{V} \rightarrow \mathbb{R}_\perp) \rightarrow \mathcal{V} \rightarrow \mathcal{E} \rightarrow (\mathcal{V} \rightarrow \mathbb{R}_\perp)\]

- \(v\): variable, \(e\): expression, \(\delta\): distribution
- \(v := e\) is called an action, \(\mathcal{A}\) set of all the actions
- \([v := e]_\delta\) is the distribution as follows
Interpretation of actions

\[[\cdot] : (\mathcal{V} \rightarrow \mathbb{R}_{\perp}) \rightarrow \mathcal{V} \rightarrow \mathcal{E} \rightarrow (\mathcal{V} \rightarrow \mathbb{R}_{\perp})\]

- \(v\): variable, \(e\): expression, \(\delta\): distribution
- \(v := e\) is called an action, \(A\) set of all the actions
- \([v := e]_\delta\) is the distribution as follows
  \([v := e]_\delta(v) = [e]_\delta\)
Interpretation of actions

- $v$: variable, $e$: expression, $\delta$: distribution
- $v := e$ is called an action, $\mathcal{A}$ set of all the actions
- $[v := e]_{\delta}$ is the distribution as follows
  - $[v := e]_{\delta}(v) = [e]_{\delta}$
  - $[v := e]_{\delta}(v') = \delta(v')$ for $v' \neq v$
Control Flow Graphs

$A$: arrows, $V$: control points, $\mathcal{A}$: actions

$$G : A \xrightarrow{\partial^-} V \quad \text{and} \quad \lambda : A \to \mathcal{A}$$
Control Flow Graphs

A: arrows, V: control points, A: actions

\[
G : A \xrightarrow{\partial^+} V \quad \text{and} \quad \lambda : A \rightarrow A
\]

\[
\Phi : V \rightarrow (E \times A)^* \\
\text{if } \Phi(v) = [(e_1, \alpha_1), \ldots, (e_k, \alpha_k)] \\
\text{then } \partial^{-} \alpha_i = v \text{ for all } v \in V \text{ and all } i \in \{1, \ldots, k\}
\]
Control Flow Graphs

\( A: \) arrows, \( V: \) control points, \( A: \) actions

\[
G : A \xrightarrow{\partial^-} V \quad \text{and} \quad \lambda : A \to A
\]

- \( \Phi : V \to (\mathcal{E} \times A)^* \)
  - if \( \Phi(v) = [(e_1, \alpha_1), \ldots, (e_k, \alpha_k)] \)
    - then \( \partial^- \alpha_i = v \) for all \( v \in V \) and all \( i \in \{1, \ldots, k\} \)
- \( v_0 \in V \) the starting point
Control Flow Graphs

$A$: arrows, $V$: control points, $A$: actions

\[ G : A \xrightarrow{\partial^-} V \quad \text{and} \quad \lambda : A \to A \]

- $\Phi : V \rightarrow (E \times A)^*$
  - if $\Phi(v) = [(e_1, \alpha_1), \ldots, (e_k, \alpha_k)]$
  - then $\partial^- \alpha_i = v$ for all $v \in V$ and all $i \in \{1, \ldots, k\}$

- $v_0 \in V$ the starting point

- $(G, \lambda, \Phi, v_0)$ is the middle-end representation
Sequential Virtual Machine

- $\delta_0$ : initial state (with the starting point $v_0$)
Sequential Virtual Machine

- $\delta_0$: initial state (with the starting point $v_0$)
- $(v_n, \delta_n)$: current state
  - suppose $\Phi(v_n) = [(e_1, \alpha_1), \ldots, (e_k, \alpha_k)]$
Sequential
Virtual Machine

- \( \delta_0 \): initial state (with the starting point \( v_0 \))
- \((v_n, \delta_n)\): current state
  
  suppose \( \Phi(v_n) = [(e_1, \alpha_1), \ldots, (e_k, \alpha_k)] \)
  
  define \( i = \min\{j \in \{1, \ldots, k\} | [e_j]_{\delta_n} \text{ is true}\} \)
Sequential Virtual Machine

- $\delta_0$: initial state (with the starting point $v_0$)
- $(v_n, \delta_n)$: current state
  
  suppose $\Phi(v_n) = [(e_1, \alpha_1), \ldots, (e_k, \alpha_k)]$
  
  define $i = \min\{j \in \{1, \ldots, k\} \mid \llbracket e_j \rrbracket_{\delta_n} \text{ is true}\}$
  
  if $i$ exists then $v_{n+1} = \partial^+ \alpha_i$ and $\delta_{n+1} = \llbracket \lambda(\alpha_i) \rrbracket_{\delta_n}$
Sequential Virtual Machine

- $\delta_0$: initial state (with the starting point $v_0$)
- $(v_n, \delta_n)$: current state

suppose $\Phi(v_n) = [(e_1, \alpha_1), \ldots, (e_k, \alpha_k)]$

define $i = \min\{j \in \{1, \ldots, k\} \mid [e_j] \delta_n \text{ is true}\}$

if $i$ exists then $v_{n+1} = \partial^+ \alpha_i$ and $\delta_{n+1} = [\lambda(\alpha_i)] \delta_n$

otherwise the induction stops
Sequential Virtual Machine

- $\delta_0$ : initial state (with the starting point $v_0$)
- $(v_n, \delta_n)$: current state
  
  suppose $\Phi(v_n) = [(e_1, \alpha_1), \ldots, (e_k, \alpha_k)]$
  
  define $i = \min\{j \in \{1, \ldots, k\} \mid \llbracket e_j \rrbracket_{\delta_n} \text{ is true}\}$
  
  if $i$ exists then $v_{n+1} = \partial^+ \alpha_i$ and $\delta_{n+1} = \llbracket \lambda(\alpha_i) \rrbracket_{\delta_n}$
  
  otherwise the induction stops

- deterministic behavior and output
An example
The Hasse/Syracuse algorithm

```
input x;
while x ≠ 1
do
    if x mod 2 = 0
        then x := x/2
    else x := 3*x + 1
done
```
An example
The Hasse/Syracuse algorithm

```plaintext
input x;
while x ≠ 1
do
  if x mod 2 = 0
    then x := x/2
  else x := 3*x + 1
  done
```
An example
The Hasse/Syracuse algorithm

```plaintext
input x;
while x/=1 do
    if x mod 2 = 0 then x:=x/2
    else x:=3*x+1
done
```

x = 1
An example

The Hasse/Syracuse algorithm

```plaintext
input x;
while x ≠ 1
do
  if x mod 2 = 0
    then x := x/2
  else x := 3*x + 1
  done
```
An example

The Hasse/Syracuse algorithm

```plaintext
input x;
while x ≠ 1
  do
    if x mod 2 = 0
      then x := x/2
    else x := 3*x + 1
  done
```
An example
The Hasse/Syracuse algorithm

```c
input x;
while x≠1
  do
    if x mod 2 = 0
      then x:=x/2
    else x:=3*x+1  x is odd
  done
```
An example

The Hasse/Syracuse algorithm

```
input x;
while x ≠ 1
  do
    if x mod 2 = 0
      then x := x/2
    else x := 3 * x + 1
  done
```
An example
The Hasse/Syracuse algorithm

input x;
while x ≠ 1
  do
    if x mod 2 = 0
      then x := x/2
    else x := 3*x + 1
  done
An example
The Hasse/Syracuse algorithm

input $x$;
while $x \neq 1$
do
  if $x \mod 2 = 0$
    then $x := x/2$
  else $x := 3 \times x + 1$
done

$\alpha$ stands for input $x$
$\gamma$ stands for $x := x/2$
$\omega$ stands for “exit”
$\delta$ stands for $x := 3 \times x + 1$
An execution trace

Hasse/Syracuse algorithm

input x;
while x ≠ 1 do
  if x mod 2 = 0 then x := x/2
  else x := 3 * x + 1
done
An execution trace
Hasse/Syracuse algorithm

input $x$;  
\[ x = 7 \]
while $x \neq 1$
  do
    if $x \mod 2 = 0$
      then $x := x/2$
    else $x := 3x + 1$
  done
An execution trace

Hasse/Syracuse algorithm

```
input x;
while x ≠ 1
    do
        if x mod 2 = 0
            then x := x/2
        else x := 3*x + 1
    done
```

```
ω
α
γ
δ
```

```
x = 22
```
An execution trace
Hasse/Syracuse algorithm

input x;
while x \neq 1
do
if x \mod 2 = 0
then x := x/2
else x := 3 \times x + 1
done
An execution trace

Hasse/Syracuse algorithm

```
input x;
while x ≠ 1
   do
      if x mod 2 = 0
         then x := x/2
      else x := 3*x + 1
   done
```

```
α δ γ δ
```

```
α

β

γ

δ
```

```
ω
```

```
x = 34
```

```
α δ γ δ
```

```
α

β

γ

δ
```

```
ω
```
An execution trace

Hasse/Syracuse algorithm

input x;
while x≠1
    do
        if x mod 2 = 0
            then x:=x/2
        else x:=3*x+1
    done
An execution trace

Hasse/Syracuse algorithm

```latex
\begin{verbatim}
input x;
while x\neq 1
  do
    if x \mod 2 = 0
      then x:=x/2
    else x:=3*x+1
  done
\end{verbatim}
```
An execution trace
Hasse/Syracuse algorithm

input x;
while x ≠ 1
  do
    if x mod 2 = 0
      then x := x/2
    else x := 3*x + 1
  done

x = 26

\begin{align*}
\alpha & \delta \gamma \delta \gamma \delta \gamma \\
\gamma & \delta \alpha \delta \gamma \delta \gamma \end{align*}
An execution trace

Hasse/Syracuse algorithm

input x;
while x ≠ 1
   do
      if x mod 2 = 0
         then x := x/2
         x = 13
      else x := 3*x + 1
   done
An execution trace

Hasse/Syracuse algorithm

\begin{verbatim}
input x;
while x ≠ 1 
  do 
    if x mod 2 = 0 
      then x := x/2 
    else x := 3 * x + 1 
  done 
\end{verbatim}
An execution trace

Hasse/Syracuse algorithm

```
input x;
while x ≠ 1
   do
      if x mod 2 = 0
         then x := x/2
      else x := 3*x + 1
   done
```

$\text{α δ γ δ γ δ γ γ δ γ}$
An execution trace
Hasse/Syracuse algorithm

input x;
while x ≠ 1
do
if x mod 2 = 0
then x := x/2
else x := 3*x + 1
done
An execution trace

Hasse/Syracuse algorithm

input x;
while x ≠ 1
  do
    if x mod 2 = 0
      then x := x/2
    else x := 3*x + 1
  done
An execution trace
Hasse/Syracuse algorithm

```java
input x;
while x ≠ 1
    do
        if x mod 2 = 0
            then x := x/2
        else x := 3*x+1
    done
```

```
α δ γ δ γ δ γ δ γ δ γ δ γ δ
```

```
ω ∈ {x = 16}
```
An execution trace
Hasse/Syracuse algorithm

input x;
while x \neq 1
  do
    if x \bmod 2 = 0
      then x := x/2
    else x := 3*x + 1
  done
An execution trace
Hasse/Syracuse algorithm

```plaintext
input x;
while x ≠ 1
  do
    if x mod 2 = 0
      then x := x/2
    else x := 3*x + 1
  done
```

```
\[\begin{align*}
\alpha &\rightarrow \delta \\
\gamma &\rightarrow \delta \\
\delta &\rightarrow \gamma \\
\gamma &\rightarrow \delta \\
\end{align*}\]
```
An execution trace

Hasse/Syracuse algorithm

```
input x;
while x ≠ 1
    do
        if x mod 2 = 0
            then x := x/2
        else x := 3*x + 1
    done
```
An execution trace
Hasse/Syracuse algorithm

```
input x;
while x≠1
  do
    if x mod 2 = 0
      then x:=x/2
    else x:=3*x+1
  done
```
An execution trace

Hasse/Syracuse algorithm

input x;
while x ≠ 1
  do
    if x mod 2 = 0
      then x := x/2
      else x := 3*x + 1
    done
Execution traces of a program as paths over its control flow graph

- Any execution trace induces a path
- Some paths do not come from an execution trace
Execution traces of a program
as paths over its control flow graph

- Any execution trace induces a path
- Some paths do not come from an execution trace

Therefore the collection of all paths provides a (strict) overapproximation of the collection of execution traces
Execution traces of a program
as paths over its control flow graph

- Any execution trace induces a path
- Some paths do not come from an execution trace

Therefore the collection of all paths provides a (strict) overapproximation of the collection of execution traces

The (infinite) collection of paths is entirely determined by the (finite) control flow graph
The overall idea of Static Analysis

The model of a program should be the finite representation of an overapproximation of the collection of all its execution traces.
The parallel composition operator
Enabling several actions to be performed at the same time
The parallel composition operator
Enabling several actions to be performed at the same time

- Middle-end: $d$-sequence of control flow graphs
The parallel composition operator
Enabling several actions to be performed at the same time

- Middle-end: $d$-sequence of control flow graphs
- Shared memory: all variables can be seen by all processes
The parallel composition operator
Enabling several actions to be performed at the same time

- Middle-end: \( d \)-sequence of control flow graphs
- Shared memory: all variables can be seen by all processes
- State: a \( d \)-uple of control points with a single distribution
The parallel composition operator
Enabling several actions to be performed at the same time

- Middle-end: $d$-sequence of control flow graphs
- Shared memory: all variables can be seen by all processes
- State: a $d$-uple of control points with a single distribution
- The virtual machine has to be adapted accordingly
Interleaving

Virtual Machine

- global clock: 1 tick / 1 process / 1 step performed
Interleaving

Virtual Machine

- global clock: 1 tick / 1 process / 1 step performed
- global choice $p \in \{1, \ldots, d\}^\mathbb{N}$
  process $p(k)$ activated at the $k^{th}$ tick of the clock
Interleaving
Virtual Machine

- global clock: 1 tick / 1 process / 1 step performed
- global choice $p \in \{1, \ldots, d\}^\mathbb{N}$
  process $p(k)$ activated at the $k^{th}$ tick of the clock
- neither behavior nor output is deterministic e.g.

\[
x := 0 \mid x := 1
\]
Precubical sets
higher dimensional graphs

•

dimension 0
Precubical sets
higher dimensional graphs

dimension 1
Precubical sets
higher dimensional graphs

\[ \partial^-_0 \cdot \longrightarrow \cdot \partial^+_0 \]

dimension 1
Precubical sets

higher dimensional graphs

dimension 2
Precubical sets
higher dimensional graphs

dimension 2
Precubical sets
another approach
dimension 2

Parallelisms

Virtual Machines
Middle-End
Dynamics

Concurrency
Generalizing graphs
Control flow
PV language
Precubical sets

another approach

dimension 1
Precubical sets
another approach

dimension 0
Precubical sets
The $\square^+$ category formally

- $\{\text{Objects of } \square^+\} = \mathbb{N}$
Precubical sets

The $\square^+$ category formally

- $\{\text{Objects of } \square^+\} = \mathbb{N}$
- $\square^+[n, m] =$

  $\{\text{words of length } m \text{ on } \{0, 1, x\} \text{ with } n \text{ occurrences of } x\}$

  empty when $n > m$; singleton when $n = m$
Precubical sets
The $\square^+$ category formally

- $\{\text{Objects of } \square^+\} = \mathbb{N}$
- $\square^+[n, m] =$
  $\{\text{words of length } m \text{ on } \{0, 1, x\} \text{ with } n \text{ occurences of } x\}$
  empty when $n > m$; singleton when $n = m$
- $\text{id}_n = x^n$
Precubical sets

The $\square^+$ category formally

- $\{\text{Objects of } \square^+\} = \mathbb{N}$
- $\square^+[n, m] = \{\text{words of length } m \text{ on } \{0, 1, x\} \text{ with } n \text{ occurences of } x\}$
  empty when $n > m$; singleton when $n = m$
- $\text{id}_n = x^n$
- $\partial_i \cong (x \cdots x_0 x \cdots x)$ and $\partial_i^+ \cong (x \cdots x_1 x \cdots x)$
Precubical sets
The □⁺ category formally

- \{\text{Objects of } □⁺\} = \mathbb{N}
- \square^+[n, m] =
  \{\text{words of length } m \text{ on } \{0, 1, x\} \text{ with } n \text{ occurances of } x\}
  \text{empty when } n > m; \text{ singleton when } n = m
- \text{id}_n = x^n
- \partial_i \equiv (x \cdots x \underbrace{0}_{i^{th}} x \cdots x) \text{ and } \partial_i^+ \equiv (x \cdots x \underbrace{1}_{i^{th}} x \cdots x)
- \text{if } w : a \to b \text{ and } w' : b \to c \text{ then } w'w \text{ is obtained}
  \text{replacing the } i^{th} \text{ occurrence of } x \text{ in } w'
  \text{by the } i^{th} \text{ letter of } w.
Precubical sets

The $\square^+$ category pictured
Precubical sets

The □⁺ category pictured
Precubical sets

The □+ category pictured

Diagram:

0 → 1 → 2

0x → 1x → x1 → 2

0 → 1 → 2

0x → 1x → x1
Precubical sets

The $\square^+$ category pictured
Precubical sets

The $\square^+$ category pictured
Precubical sets
as presheaves over $\Box^+$
Tensor product
of precubical sets

Given precubical sets $K$ and $K'$ of dimension $p$ and $q$, the set of $n$-cubes for $0 \leq n \leq p + q$

$$(K \otimes K')_n = \bigsqcup_{i+j=n} K_i \times K_j$$

For $x \otimes y \in K_i \times K_j'$ with $i + j = n$ the $k^{th}$ face map, with $0 \leq k < n$, is given by

$$\partial_k^\pm (x \otimes y) = \begin{cases} 
\partial_k^\pm (x) \otimes y & \text{if } 0 \leq k < i \\
x \otimes \partial_k^\pm y & \text{if } i \leq k < n
\end{cases}$$
Example of tensor product of precubical sets
Example of tensor product of precubical sets
Example of tensor product of precubical sets
Example of tensor product of precubical sets
Example of tensor product of precubical sets
Example of tensor product of precubical sets
True concurrency - discrete version

Virtual Machine

- get rid of the global clock
True concurrency - discrete version

Virtual Machine

- get rid of the global clock
- an execution step from \(((v_1, \ldots, v_d), \delta)\) becomes a multiset $M$ on \{1, \ldots, d\}
True concurrency - discrete version

Virtual Machine

- get rid of the global clock
- an execution step from $((v_1, \ldots, v_d), \delta)$ becomes a multiset $M$ on $\{1, \ldots, d\}$
- need a total order on multisets to provide a global choice
True concurrency - discrete version

Virtual Machine

- get rid of the global clock
- an execution step from $((v_1, \ldots, v_d), \delta)$ becomes a multiset $M$ on $\{1, \ldots, d\}$
- need a total order on multisets to provide a global choice
- interleaving model only allows $M$ such that $|M| = 1$
True concurrency - discrete version

Virtual Machine

- performing $\mathcal{M}$ only makes sense under the sheaf condition: for all finite sequences $s$ of length $\ell$
  
  with elements in $\{1, \ldots, d\}$ and satisfying
  
  $\#\{i \mid s_i = k\} \leq \mathcal{M}(k)$ for all $k \in \{1, \ldots, d\}$,

  the intermediate state of the interleaving execution at step $\ell$ from the initial state $(v_1, \ldots, v_d, \delta)$ and according to the global choice $s$, only depends on the multiset

  $k \mapsto \#\{i \mid s_i = k\}$. 
True concurrency - discrete version
Control flow from tensor product of control flow graphs

(process p) x:=0 ; x:=2 |
(process q) x:=1 ; x:=2
True concurrency - discrete version
Control flow from tensor product of control flow graphs

\[(\text{process } p) \ x:=0 \ ; \ x:=2 \mid \ (\text{process } q) \ x:=1 \ ; \ x:=2\]

\[
\begin{array}{c}
\xrightarrow{\uparrow} \quad \xrightarrow{\uparrow} \\
\xrightarrow{\uparrow} \quad \xrightarrow{\uparrow} \\
\xrightarrow{\uparrow} \quad \xrightarrow{\uparrow}
\end{array}
\]

\[
\begin{array}{c}
x:=2 \\
x:=1 \\
x:=0 \\
x:=2
\end{array}
\]
True concurrency - discrete version

Control flow from tensor product of control flow graphs

\[(process \ p) \ x := 0 \ ; \ x := 2 \mid \ (process \ q) \ x := 1 \ ; \ x := 2\]
True concurrency - discrete version

Control flow from tensor product of control flow graphs

\[(\text{process p}) \ x := 0 \ ; \ x := 2 \mid (\text{process q}) \ x := 1 \ ; \ x := 2\]

- \( p + q \subseteq 2p + 2q \)
- \( 2p + 2q \) is compatible yet \( p + q \) is not
True concurrency - discrete version

Virtual Machine

\[(\text{process } p) \ x := y ; \ x := 2 | \]
\[(\text{process } q) \ x := z ; \ x := 2 \]
True concurrency - discrete version

Virtual Machine

(process p) x := y ; x := 2 |
(process q) x := z ; x := 2

- filling square may depend on the current distribution
True concurrency - discrete version

Virtual Machine

\((\text{process } p) \ x:=y \ ; \ x:=2 | \ (\text{process } q) \ x:=z \ ; \ x:=2\)

- filling square may depend on the current distribution
- solution: actions with disjoint sets of occurring variables
The control flow precubical set
Middle-end representation taking race conditions into account

- $G_1 \otimes \cdots \otimes G_d$ tensor product of the control flow graphs
The control flow precubical set
Middle-end representation taking race conditions into account

- $G_1 \otimes \cdots \otimes G_d$ tensor product of the control flow graphs
- Labelling all cubes of dimension $1 \leq k \leq d$ by
  $$\lambda(\alpha_1 \otimes \cdots \otimes \alpha_k) = \lambda_1(\alpha_1), \cdots, \lambda_k(\alpha_k) \text{ for } k \leq d$$
The control flow precubical set
Middle-end representation taking race conditions into account

- $G_1 \otimes \cdots \otimes G_d$ tensor product of the control flow graphs
- Labelling all cubes of dimension $1 \leq k \leq d$ by
  \[ \lambda(\alpha_1 \otimes \cdots \otimes \alpha_k) = \lambda_1(\alpha_1), \ldots, \lambda_k(\alpha_k) \] for $k \leq d$
- remove all cubes $\alpha_1 \otimes \cdots \otimes \alpha_k$ s.t. there are
  $1 \leq i < j \leq k$ whose actions $\lambda_i(\alpha_i)$ and $\lambda_j(\alpha_j)$
  share some variable
Dynamics
Dynamics

$$A = 2p + 5q$$
True concurrency - discrete version

Virtual Machine

- the true concurrency virtual machine is thus well-defined
True concurrency - discrete version
Virtual Machine

- the true concurrency virtual machine is thus well-defined
- language extension paradigm:
  parallelize as much as possible
True concurrency - discrete version

Virtual Machine

- the true concurrency virtual machine is thus well-defined
- language extension paradigm:
  parallelize as much as possible
- a weak form of synchronization remains...
  ...continuous models are not far
True concurrency - discrete version

Virtual Machine

- the true concurrency virtual machine is thus well-defined
- language extension paradigm:
  parallelize as much as possible
- a weak form of synchronization remains...
  ...continuous models are not far
- how do we deal with nondeterminacy?
The PV language

Dijkstra 68 - Input language for ALCOOL in an extended form

- **Sem**: set of semaphores with arity in $\mathbb{N} \setminus \{0, 1\}$
The PV language

Dijkstra 68 - Input language for ALCOOL in an extended form

- *Sem*: set of semaphores with arity in $\mathbb{N} \setminus \{0, 1\}$
- *Mtx*: set of mutex, an alias for a semaphore of arity 2
The PV language

Dijkstra 68 - Input language for ALCOOL in an extended form

- $\text{Sem}$: set of semaphores with arity in $\mathbb{N} \setminus \{0, 1\}$
- $\text{Mtx}$: set of mutex, an alias for a semaphore of arity 2
- A semaphore $x$ of arity $n$ is a resource offering $n - 1$ tokens, each process can hold one token or more

- $\text{Wait}$: set of synchronization barriers with arity in $\mathbb{N} \setminus \{0, 1\}$
- Instruction $W(x)$ blocks the execution of the process until $n$ (arity of $x$) processes are blocked by $x$ then all the execution are resumed at the same time
The PV language

Dijkstra 68 - Input language for ALCOOL in an extended form

- **Sem**: set of semaphores with arity in \( \mathbb{N} \setminus \{0, 1\} \)
- **Mtx**: set of mutex, an alias for a semaphore of arity 2
- A semaphore \( x \) of arity \( n \) is a resource offering \( n - 1 \) tokens, each process can hold one token or more
- a process acquire a token executing the instruction \( P(x) \) and release it executing the instruction \( V(x) \)
The PV language
Dijkstra 68 - Input language for ALCOOL in an extended form

- *Sem*: set of semaphores with arity in $\mathbb{N} \setminus \{0, 1\}
- *Mtx*: set of mutex, an alias for a semaphore of arity 2
- A semaphore $x$ of arity $n$ is a resource offering $n - 1$ tokens,
  each process can hold one token or more
- a process acquire a token executing the instruction $P(x)$
  and release it executing the instruction $V(x)$
- A mutex can be held by only one process at the time
The PV language

Dijkstra 68 - Input language for ALCOOL in an extended form

- $Sem$: set of semaphores with arity in $\mathbb{N} \setminus \{0, 1\}$
- $Mtx$: set of mutex, an alias for a semaphore of arity 2
- A semaphore $x$ of arity $n$ is a resource offering $n - 1$ tokens,
each process can hold one token or more
- a process acquire a token executing the instruction $P(x)$
  and release it executing the instruction $V(x)$
- A mutex can be held by only one process at the time
- Trying to perform $P(x)$ though $x$ is not available
  blocks the execution unless $x$ is a mutex
  already held by the process
The PV language

Dijkstra 68 - Input language for ALCOOL in an extended form

- \(Sem\): set of semaphores with arity in \(\mathbb{N} \setminus \{0, 1\}\)
- \(Mtx\): set of mutex, an alias for a semaphore of arity 2
- A semaphore \(x\) of arity \(n\) is a resource offering \(n – 1\) tokens,
  each process can hold one token or more
- a process acquire a token executing the instruction \(P(x)\)
  and release it executing the instruction \(V(x)\)
- A mutex can be held by only one process at the time
- Trying to perform \(P(x)\) though \(x\) is not available
  blocks the execution unless \(x\) is a mutex
  already held by the process
- the instruction \(V(x)\) is not blocking
The PV language

Dijkstra 68 - Input language for ALCOOL in an extended form

- **Sem**: set of semaphores with arity in $\mathbb{N} \setminus \{0, 1\}$
- **Mtx**: set of mutex, an alias for a semaphore of arity 2
- A semaphore $x$ of arity $n$ is a resource offering $n - 1$ tokens, each process can hold one token or more
- a process acquire a token executing the instruction $P(x)$ and release it executing the instruction $V(x)$
- A mutex can be held by only one process at the time
- Trying to perform $P(x)$ though $x$ is not available blocks the execution unless $x$ is a mutex already held by the process
- the instruction $V(x)$ is not blocking
- **Wait**: set of synchronization barriers with arity in $\mathbb{N} \setminus \{0, 1\}$
The PV language

Dijkstra 68 - Input language for ALCOOL in an extended form

- **Sem**: set of semaphores with arity in $\mathbb{N} \setminus \{0, 1\}$
- **Mtx**: set of mutex, an alias for a semaphore of arity 2
- A semaphore $x$ of arity $n$ is a resource offering $n - 1$ tokens, each process can hold one token or more
- a process acquire a token executing the instruction $P(x)$ and release it executing the instruction $V(x)$
- A mutex can be held by only one process at the time
- Trying to perform $P(x)$ though $x$ is not available blocks the execution unless $x$ is a mutex already held by the process
- the instruction $V(x)$ is not blocking
- **Wait**: set of synchronization barriers with arity in $\mathbb{N} \setminus \{0, 1\}$
- Instruction $W(x)$ blocks the execution of the process until $n$ (arity of $x$) processes are blocked by $x$ then all the execution are resumed at the same time
Extending the middle-end representation

Potential function along a path

- $\mathcal{R} = \{\text{semaphores and mutex}\}$
Extending the middle-end representation

Potential function along a path

- \( \mathcal{R} = \{\text{semaphores and mutex}\} \)
- distribution: \( \delta : \mathcal{V} \cup \mathcal{R} \to \mathbb{N} \)
Extending the middle-end representation

Potential function along a path

\[ R = \{ \text{semaphores and mutex} \} \]

- distribution: \( \delta : V \cup R \to \mathbb{N} \)

\[
\begin{align*}
[P(a)]_\delta(x) &= \begin{cases} 
\delta(x) & \text{if } x \neq a \\
\delta(a) + 1 & \text{if } x = a 
\end{cases}
\end{align*}
\]
Extending the middle-end representation

Potential function along a path

- $\mathcal{R} = \{\text{semaphores and mutex}\}$
- distribution: $\delta : \mathcal{V} \cup \mathcal{R} \to \mathbb{N}$

\[
\begin{align*}
\llbracket P(a) \rrbracket_\delta(x) &= \begin{cases} 
\delta(x) & \text{if } x \neq a \\
\delta(a) + 1 & \text{if } x = a
\end{cases} \\
\llbracket V(a) \rrbracket_\delta(x) &= \begin{cases} 
\delta(x) & \text{if } x \neq a \\
\max\{0, \delta(a) - 1\} & \text{if } x = a
\end{cases}
\end{align*}
\]
Extending the middle-end representation

Potential function along a path

- $\mathcal{R} = \{\text{semaphores and mutex}\}$
- distribution: $\delta: \mathcal{V} \cup \mathcal{R} \rightarrow \mathbb{N}$

$$\llbracket P(a) \rrbracket_\delta(x) = \begin{cases} 
\delta(x) & \text{if } x \neq a \\
\delta(a) + 1 & \text{if } x = a 
\end{cases}$$

$$\llbracket V(a) \rrbracket_\delta(x) = \begin{cases} 
\delta(x) & \text{if } x \neq a \\
\max\{0, \delta(a) - 1\} & \text{if } x = a 
\end{cases}$$

$$\llbracket W(a) \rrbracket = \text{ignored}$$
Extending the middle-end representation

Conservative process

- $\gamma = \gamma_1, \ldots, \gamma_n$ a path on a cfg, then by definition
  $$[\gamma] \cdot \delta = [\lambda(\gamma_n)] \cdots [\lambda(\gamma_1)] \cdot \delta$$

  is the action of the path $\gamma$ on the distribution $\delta$
Extending the middle-end representation

Conservative process

- $\gamma = \gamma_1, \ldots, \gamma_n$ a path on a cfg, then by definition
  $\mathcal{[\gamma]} \cdot \delta = \mathcal{[\lambda(\gamma_n)] \cdots [\lambda(\gamma_1)] \cdot \delta}$
  is the action of the path $\gamma$ on the distribution $\delta$
- A process is conservative when for all paths $\gamma, \gamma'$ on its
cfg, all $x \in \mathcal{R}$ and all distributions $\delta$

  $\partial^- \gamma = \partial^- \gamma'$ and $\partial^+ \gamma = \partial^+ \gamma' \Rightarrow \mathcal{[\gamma]} \cdot \delta(x) = \mathcal{[\gamma']} \cdot \delta(x)$
Being conservative is decidable

- approximation: a mapping from $V$ to $2^NR$
Being conservative is decidable

- approximation: a mapping from $V$ to $2^\mathbb{N}$
- $s \subseteq s'$ means $s(v) \subseteq s'(v)$ for all $v \in V$
Being conservative
is decidable

- approximation: a mapping from \( V \) to \( 2^{\mathbb{N}^{\mathbb{R}}} \)
- \( s \subseteq s' \) means \( s(v) \subseteq s'(v) \) for all \( v \in V \)
- \( \{s_0, \ldots, s_n\} \) inductively defined as follows:
Being conservative is decidable

- approximation: a mapping from $V$ to $2^{\mathbb{N}^R}$
- $s \subseteq s'$ means $s(v) \subseteq s'(v)$ for all $v \in V$
- $\{s_0, \ldots, s_n\}$ inductively defined as follows:
  The initial term $s_0$ is defined by $s_0(v_0) = \{\delta_0\}$, and $s_0(v) = \emptyset$ for $v \neq v_0$. 
Being conservative is decidable

- approximation: a mapping from $V$ to $2^{\mathbb{N}^R}$
- $s \subseteq s'$ means $s(v) \subseteq s'(v)$ for all $v \in V$
- $\{s_0, \ldots, s_n\}$ inductively defined as follows:
  The initial term $s_0$ is defined by $s_0(v_0) = \{\delta_0\}$, and $s_0(v) = \emptyset$ for $v \neq v_0$.
  Assuming $s_n$ is built, $s_{n+1}$ is defined for all $v \in V$ by

$$s_{n+1}(v) = s_n(v) \cup \bigcup_{f \in A; \partial^+ f = v; \lambda(f) \in \{P,V\}} f \cdot s_n(\partial^- f)$$
Being conservative induces a potential function

- The induction stops at the $n^{th}$ step when either of the following property is satisfied:
Being conservative induces a potential function

- The induction stops at the $n^{th}$ step when either of the following property is satisfied:
  $s_n = s_{n-1}$: 'true', or
  there exists some $v \in V$ such that $\#s_n(v) \geq 2$: 'false'
Being conservative induces a potential function

- The induction stops at the $n^{th}$ step when either of the following property is satisfied:
  $s_n = s_{n-1}$: ‘true’, or
  there exists some $v \in V$ such that $\#s_n(v) \geq 2$: ‘false’
- in the first case we have the potential function
  $F : V \times \mathcal{R} \to \mathbb{N}$ defined by
  $F(v, x) = \delta(x)$ where $s_n(v) = \{\delta\}$
  note that if $s_n(v) = \emptyset$ then $v$ is unreachable
The potential function of a PV program $P_1 \cdots | P_d$

- assume each $P_k$ is conservative and $F_k$ the associated potential function
The potential function
of a PV program $P_1 | \cdots | P_d$

- assume each $P_k$ is conservative
  and $F_k$ the associated potential function
- let $K_0 = V_1 \times \cdots \times V_d$ the 0-dimensional cubes of the
  control flow precubical set $K$ obtained
  by ignoring instructions $P$, $V$, and $W$
The potential function
of a PV program $P_1|\cdots|P_d$

- assume each $P_k$ is conservative
  and $F_k$ the associated potential function
- let $K_0 = V_1 \times \cdots \times V_d$ the 0-dimensional cubes of the control flow precubical set $K$ obtained by ignoring instructions $P$, $V$, and $W$
- The potential function $F : K_0 \times R \rightarrow \mathbb{N}$ is

$$F(v_1, \ldots, v_d, x) = \sum_{k=1}^{d} F_k(v_k, x)$$
The control flow precubical set taking $P$, $V$, and $W$ into account

- Remove from $K$ all $v$ such that $F(v, x) \geq \text{arity}(x)$ for some semaphore or mutex $x$
The control flow precubical set
taking $P$, $V$, and $W$ into account

- Remove from $K$ all $v$ such that $F(v, x) \geq \text{arity}(x)$ for some semaphore or mutex $x$
- replace each $n$-cubes $c$ whose edges carrying $W(x)$ for some synchronization barrier $x$ of arity $n$
  by an arrow $\text{low}(c) \rightarrow \text{up}(c)$
The control flow precubical set
taking $P$, $V$, and $W$ into account

- Remove from $K$ all $v$ such that $F(v, x) \geq \text{arity}(x)$ for some semaphore or mutex $x$
- replace each $n$-cubes $c$ whose edges carrying $W(x)$ for some synchronization barrier $x$ of arity $n$
  by an arrow $\text{low}(c) \rightarrow \text{up}(c)$
- remove all arrows carrying $W(x)$ for some synchronization barrier $x$
Control Flow Precubical Set: an example

\[ y := 0 \cdot W(b) \cdot P(a) \cdot x := z \cdot V(a) | z := 0 \cdot W(b) \cdot P(a) \cdot x := y \cdot V(a) \]
Control Flow Precubical Set: an example

\[ y := 0.W(b).P(a).x := z.V(a) \mid z := 0.W(b).P(a).x := y.V(a) \]
Control Flow Precubical Set: an example

\[ y := 0 \cdot W(b) \cdot P(a) \cdot x := z \cdot V(a) | z := 0 \cdot W(b) \cdot P(a) \cdot x := y \cdot V(a) \]
Control Flow Precubical Set: an example

\[
y := 0 . W(b) . P(a) . x := z . V(a) | z := 0 . W(b) . P(a) . x := y . V(a)
\]
Control Flow Precubical Set: an example

\[y := 0 . W(b) . P(a) . x := z . V(a) | z := 0 . W(b) . P(a) . x := y . V(a)\]
Control Flow Precubical Set: an example

\[ y := 0 \cdot W(b) \cdot P(a) \cdot x := z \cdot V(a) | z := 0 \cdot W(b) \cdot P(a) \cdot x := y \cdot V(a) \]
Control Flow Precubical Set: an example

\[ y := 0 \cdot W(b) \cdot P(a) \cdot x := z \cdot V(a) \mid z := 0 \cdot W(b) \cdot P(a) \cdot x := y \cdot V(a) \]
Control Flow Precubical Set: an example

\[ y := 0. W(b). P(a). x := z. V(a) | z := 0. W(b). P(a). x := y. V(a) \]
Control Flow Precubical Set: an example

\[ y := 0 . W(b) . P(a) . x := z . V(a) | z := 0 . W(b) . P(a) . x := y . V(a) \]
Control Flow Precubical Set: an example

\[ y := 0 \cdot W(b) \cdot P(a) \cdot x := z \cdot V(a) | z := 0 \cdot W(b) \cdot P(a) \cdot x := y \cdot V(a) \]
Control Flow Precubical Set: an example

\[ y := 0 . W(b) . P(a) . x := z . V(a) \mid z := 0 . W(b) . P(a) . x := y . V(a) \]
Control Flow Precubical Set: an example

\[ y := 0 . W(b) . P(a) . x := z . V(a) | z := 0 . W(b) . P(a) . x := y . V(a) \]
Control Flow Precubical Set: an example

\[
y := 0 . W(b) . P(a) . x := z . V(a) | z := 0 . W(b) . P(a) . x := y . V(a)
\]
Control Flow Precubical Set: an example

\( y := 0 \cdot W(b) \cdot P(a) \cdot x := z \cdot V(a) | z := 0 \cdot W(b) \cdot P(a) \cdot x := y \cdot V(a) \)
Control Flow Precubical Set: an example

\[ y:=0.W(b).P(a).x:=z.V(a) | z:=0.W(b).P(a).x:=y.V(a) \]
Control Flow Precubical Set: an example

\[ y := 0 . W(b) . P(a) . x := z . V(a) | z := 0 . W(b) . P(a) . x := y . V(a) \]
Control Flow Precubical Set: an example

\[ y := 0 \cdot W(b) \cdot P(a) \cdot x := z \cdot V(a) \mid z := 0 \cdot W(b) \cdot P(a) \cdot x := y \cdot V(a) \]