Introduction to Directed Algebraic Topology with a view towards modelling Concurrency I

Mathematical Structures of Computations - Lyon 2014

Emmanuel Haucourt

CEA-Tech, NanoInnov

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

v	content of $v \in \mathcal{V}$	$x \in \mathbb{R}$	constant
$\wedge$	minimum	$\vee$	maximum
$+$	addition	$-$	substraction
$*$	multiplication	$/$	division
$\leqslant$	less or equal	$\geqslant$	greater of equal
$<$	strictly less	$>$	strictly greater
$\neg$	complement	$=$	equal
$\perp$	bottom		

nullary	unary
$\perp$, $x \in \mathbb{R}$, $v \in \mathcal{V}$	$\neg$
binary	
$\wedge$, $\vee$, $+$, $-$, $*$, $/$, $<$, $>$, $\leqslant$, $\geqslant$, $=$	

Interpretation of expressions

$[\![\_]\!] : (\mathcal{V} \to \mathbb{R}_\perp) \to \mathcal{E} \to \mathbb{R}_\perp$

- distribution: $\delta : \mathcal{V} \to \mathbb{R}_\perp$

Interpretation of expressions

$[\![\_]\!] : (\mathcal{V} \to \mathbb{R}_\perp) \to \mathcal{E} \to \mathbb{R}_\perp$

- distribution: $\delta : \mathcal{V} \to \mathbb{R}_\perp$
- $[\![v]\!]_\delta = \delta(v)$

Interpretation of expressions

$[\![\_]\!] : (\mathcal{V} \to \mathbb{R}_\perp) \to \mathcal{E} \to \mathbb{R}_\perp$

- distribution: $\delta : \mathcal{V} \to \mathbb{R}_\perp$
- $[\![v]\!]_\delta = \delta(v)$
- 0 stands for false any value in $\mathbb{R} \setminus \{0\}$ stands for true

Interpretation of expressions

$[\![\_]\!] : (\mathcal{V} \to \mathbb{R}_\perp) \to \mathcal{E} \to \mathbb{R}_\perp$

- distribution: $\delta : \mathcal{V} \to \mathbb{R}_\perp$
- $[\![v]\!]_\delta = \delta(v)$
- 0 stands for false any value in $\mathbb{R} \setminus \{0\}$ stands for true
- $[\![\neg]\!] : \mathbb{R}_\perp \to \mathbb{R}_\perp$,
  $[\![\neg]\!](0) = 1$, and
  $[\![\neg]\!](x) = 0$ for all $x \in \mathbb{R} \setminus \{0\}$

Interpretation of expressions

$[\![\_]\!] : (\mathcal{V} \to \mathbb{R}_\perp) \to \mathcal{E} \to \mathbb{R}_\perp$

- distribution: $\delta : \mathcal{V} \to \mathbb{R}_\perp$
- $[\![v]\!]_\delta = \delta(v)$
- 0 stands for false any value in $\mathbb{R} \setminus \{0\}$ stands for true
- $[\![\neg]\!] : \mathbb{R}_\perp \to \mathbb{R}_\perp$,
  $[\![\neg]\!](0) = 1$, and
  $[\![\neg]\!](x) = 0$ for all $x \in \mathbb{R} \setminus \{0\}$
- $[\![e]\!] = \perp$ for all expression e in which $\perp$ occurs

Interpretation of actions

$[\![\_]\!] : (\mathcal{V} \to \mathbb{R}_\perp) \to \mathcal{V} \to \mathcal{E} \to (\mathcal{V} \to \mathbb{R}_\perp)$

- v: variable, e: expression, δ: distribution
- $v := e$ is called an action, $\mathcal{A}$ set of all the actions

Interpretation of actions

$[\![\_]\!] : (\mathcal{V} \to \mathbb{R}_\perp) \to \mathcal{V} \to \mathcal{E} \to (\mathcal{V} \to \mathbb{R}_\perp)$

- v: variable, e: expression, δ: 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

$[\![\_]\!] : (\mathcal{V} \to \mathbb{R}_\perp) \to \mathcal{V} \to \mathcal{E} \to (\mathcal{V} \to \mathbb{R}_\perp)$

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

Interpretation of actions

$[\![\_]\!] : (\mathcal{V} \to \mathbb{R}_\perp) \to \mathcal{V} \to \mathcal{E} \to (\mathcal{V} \to \mathbb{R}_\perp)$

- v: variable, e: expression, δ: 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

$$G : A \underset{\partial^+}{\overset{\partial^-}{\rightrightarrows}} V \quad \text{and} \quad \lambda : A \to \mathcal{A}$$

Control Flow Graphs

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

$$G : A \underset{\partial^+}{\overset{\partial^-}{\rightrightarrows}} V \quad \text{and} \quad \lambda : A \to \mathcal{A}$$

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

Control Flow Graphs

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

$$G : A \overset{\partial^-}{\underset{\partial^+}{\rightrightarrows}} V \quad \text{and} \quad \lambda : A \to \mathcal{A}$$

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

Control Flow Graphs

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

$$G : A \underset{\partial^+}{\overset{\partial^-}{\rightrightarrows}} V \quad \text{and} \quad \lambda : A \to \mathcal{A}$$

- $\Phi : \mathcal{V} \to (\mathcal{E} \times A)^*$
  if $\Phi(v) = [(e_1, \alpha_1), \ldots, (e_k, \alpha_k)]$
  then $\partial^- \alpha_i = v$ for all $v \in \mathcal{V}$ and all $i \in \{1, \ldots, k\}$
- $v_0 \in V$ the starting point
- (G, λ, Φ, v_0) is the middle-end representation

- δ_0 : initial state (with the starting point v_0)

Sequential

Virtual Machine

- δ_0 : initial state (with the starting point v_0)
- (v_n, δ_n): current state
  suppose $\Phi(v_n) = [(e_1, \alpha_1), \ldots, (e_k, \alpha_k)]$

Sequential

Virtual Machine

- δ_0 : initial state (with the starting point v_0)
- (v_n, δ_n): current state
  suppose $\Phi(v_n) = [(e_1, \alpha_1), \ldots, (e_k, \alpha_k)]$
  define $i = \min\{j \in \{1, \ldots, k\} \mid [\![e_j]\!]_{\delta_n} \text{ is true}\}$

Sequential

Virtual Machine

- δ_0 : initial state (with the starting point v_0)
- (v_n, δ_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 $\delta_{n+1} = [\![\lambda(\alpha_i)]\!]_{\delta_n}$

Sequential

Virtual Machine

- δ_0 : initial state (with the starting point v_0)
- (v_n, δ_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

- δ_0 : initial state (with the starting point v_0)
- (v_n, δ_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
- 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

```
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
```

$x = 1$

An example

The Hasse/Syracuse algorithm

input x;

while x≠1

$x \neq 1$

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
```

x is even

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
```

x is odd

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
```

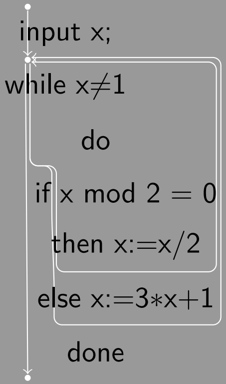
An example

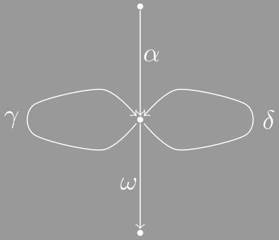
The Hasse/Syracuse algorithm

```
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





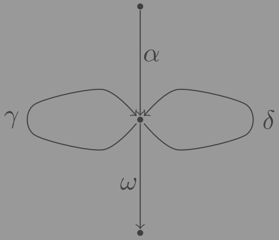
α stands for `input x` γ stands for `x:=x/2`

ω stands for "`exit`" δ stands for `x:=3*x+1`

An execution trace

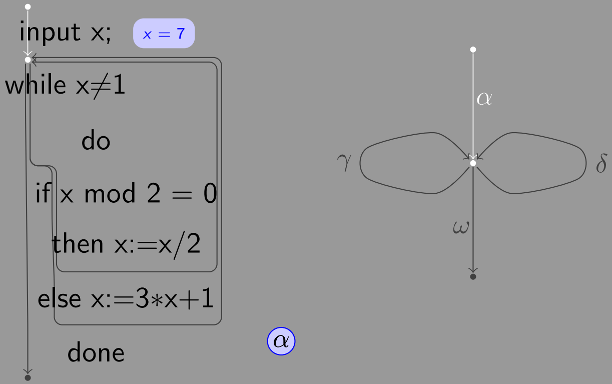
Hasse/Syracuse algorithm

```
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

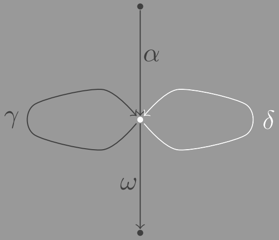


An execution trace

Hasse/Syracuse algorithm

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

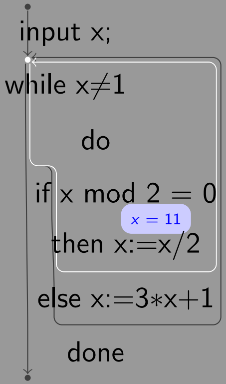
$x = 22$

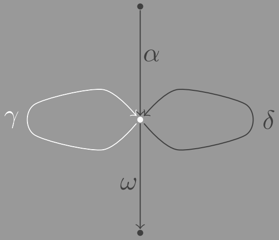


$\alpha\delta$

An execution trace

Hasse/Syracuse algorithm





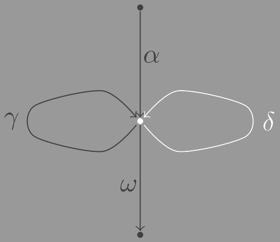
$\alpha\ \delta\ \gamma$

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
```

$x = 34$



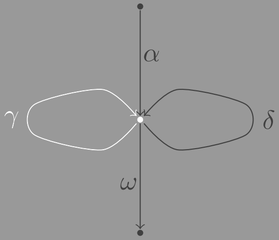
$\alpha\ \delta\ \gamma\ \delta$

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
```

$x = 17$



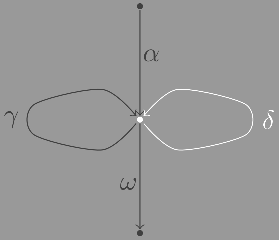
$\alpha\ \delta\ \gamma\ \delta\ \gamma$

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
```

$x = 52$



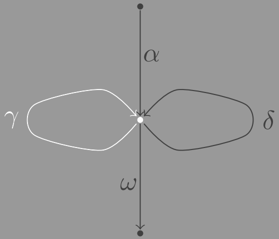
$\alpha\ \delta\ \gamma\ \delta\ \gamma\ \delta$

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
```

$x = 26$



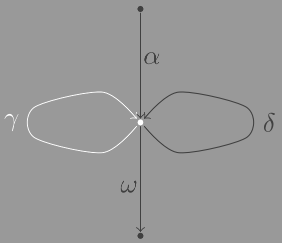
$\alpha\ \delta\ \gamma\ \delta\ \gamma\ \delta\ \gamma$

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

$x = 13$



$\alpha\ \delta\ \gamma\ \delta\ \gamma\ \delta\ \gamma\ \gamma$

An execution trace

Hasse/Syracuse algorithm

input x;

while x≠1

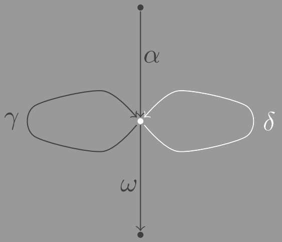
do

if x mod 2 = 0

then x:=x/2

else x:=3∗x+1

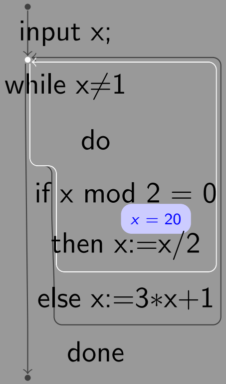
done $x = 40$

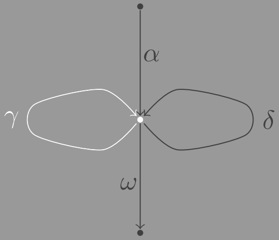


$\alpha\ \delta\ \gamma\ \delta\ \gamma\ \delta\ \gamma\ \gamma\ \delta$

An execution trace

Hasse/Syracuse algorithm





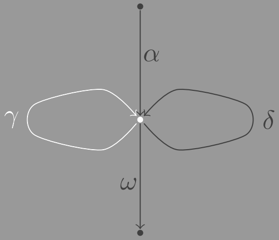
$\alpha\ \delta\ \gamma\ \delta\ \gamma\ \delta\ \gamma\ \gamma\ \delta\ \gamma$

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
```

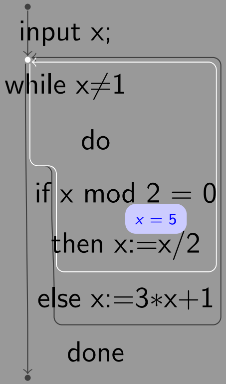
$x = 10$

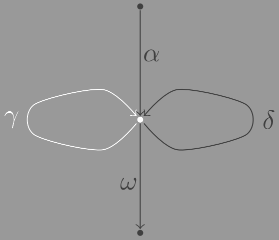


$\alpha\ \delta\ \gamma\ \delta\ \gamma\ \delta\ \gamma\ \gamma\ \delta\ \gamma\ \gamma$

An execution trace

Hasse/Syracuse algorithm





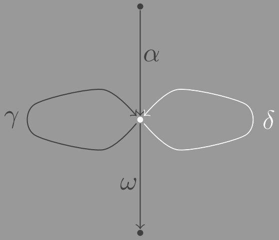
$\alpha\ \delta\ \gamma\ \delta\ \gamma\ \delta\ \gamma\ \gamma\ \delta\ \gamma\ \gamma\ \gamma$

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
```

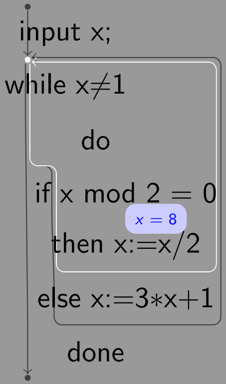
$x = 16$

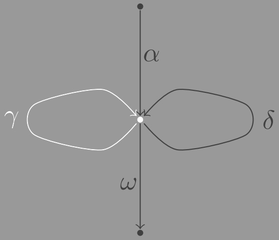


$\alpha\ \delta\ \gamma\ \delta\ \gamma\ \delta\ \gamma\ \gamma\ \delta\ \gamma\ \gamma\ \gamma\ \delta$

An execution trace

Hasse/Syracuse algorithm

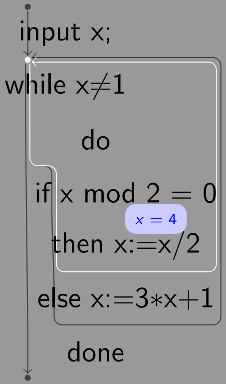


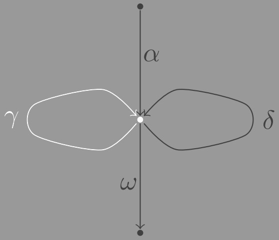


$\alpha\ \delta\ \gamma\ \delta\ \gamma\ \delta\ \gamma\ \gamma\ \delta\ \gamma\ \gamma\ \gamma\ \delta\ \gamma$

An execution trace

Hasse/Syracuse algorithm





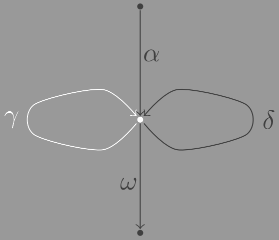
$\alpha\ \delta\ \gamma\ \delta\ \gamma\ \delta\ \gamma\ \gamma\ \delta\ \gamma\ \gamma\ \gamma\ \delta\ \gamma\ \gamma$

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
```

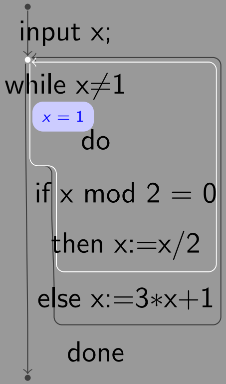
$x = 2$

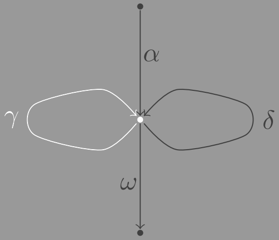


$\alpha\ \delta\ \gamma\ \delta\ \gamma\ \delta\ \gamma\ \gamma\ \delta\ \gamma\ \gamma\ \gamma\ \delta\ \gamma\ \gamma\ \gamma$

An execution trace

Hasse/Syracuse algorithm



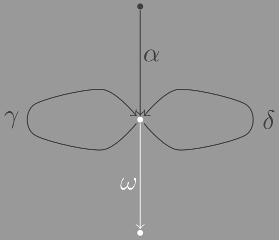


$\alpha\ \delta\ \gamma\ \delta\ \gamma\ \delta\ \gamma\ \gamma\ \delta\ \gamma\ \gamma\ \gamma\ \delta\ \gamma\ \gamma\ \gamma\ \gamma$

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
```



$\alpha\ \delta\ \gamma\ \delta\ \gamma\ \delta\ \gamma\ \gamma\ \delta\ \gamma\ \gamma\ \gamma\ \delta\ \gamma\ \gamma\ \gamma\ \gamma\ \omega$

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 | x:=1
```

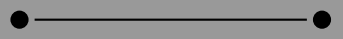
Precubical sets

higher dimensional graphs

dimension 0

Precubical sets

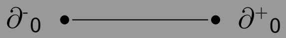
higher dimensional graphs



dimension 1

Precubical sets

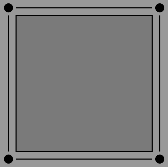
higher dimensional graphs



dimension 1

Precubical sets

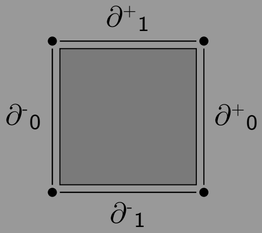
higher dimensional graphs



dimension 2

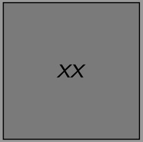
Precubical sets

higher dimensional graphs



dimension 2

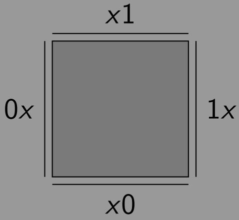
Precubical sets
another approach



dimension 2

Precubical sets

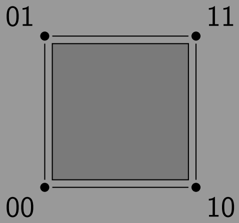
another approach



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

- $\{\text{Objects of } \square^+\} = \mathbb{N}$
- $\square^+[n, m] =$
  $\{\text{words of length } m \text{ on } \{0, 1, x\} \text{ with } n \text{ occurences of } x\}$
  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 $n = m$
- $\mathrm{id}_n = x^n$

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 $n = m$
- $\mathrm{id}_n = x^n$
- $\partial_i^- \cong (x \cdots x \underbrace{0}_{i^{th}} x \cdots x)$ and $\partial_i^+ \cong (x \cdots x \underbrace{1}_{i^{th}} x \cdots x)$

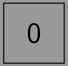
Precubical sets

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 $n = m$
- $\mathrm{id}_n = x^n$
- $\partial_i^- \cong (x \cdots x \underbrace{0}_{i^{th}} x \cdots x)$ and $\partial_i^+ \cong (x \cdots x \underbrace{1}_{i^{th}} x \cdots x)$
- if $w : a \to b$ and $w' : b \to c$ then $w'w$ is obtained
  replacing the i^{th} occurrence of x in w'
  by the i^{th} letter of w.

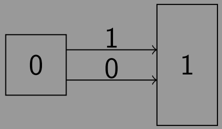
Precubical sets

The $\square^+$ category pictured



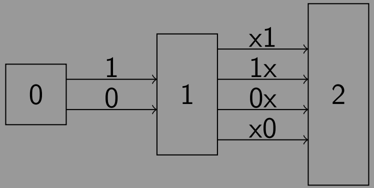
Precubical sets

The $\square^+$ category pictured



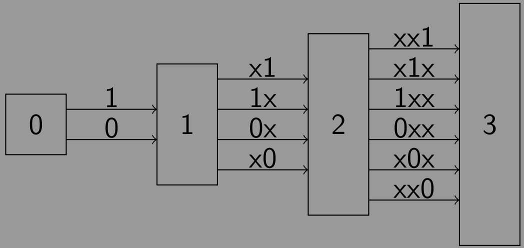
Precubical sets

The $\square^+$ category pictured



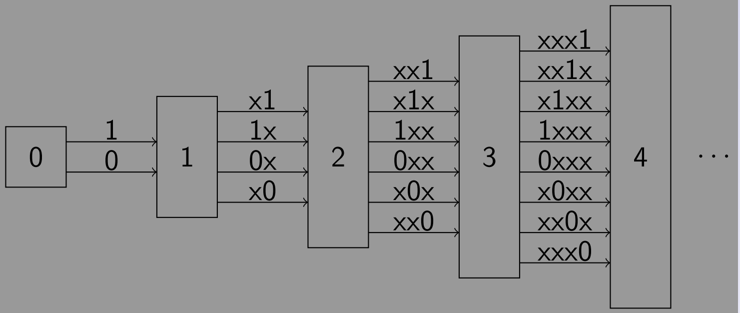
Precubical sets

The $\square^+$ category pictured



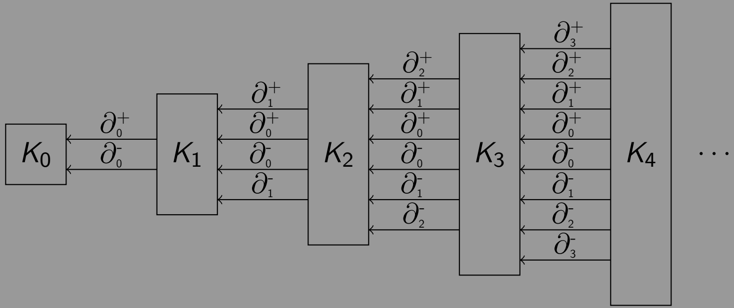
Precubical sets

The $\square^+$ category pictured



Precubical sets

as presheaves over $\square^+$



Tensor product

of precubical sets

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

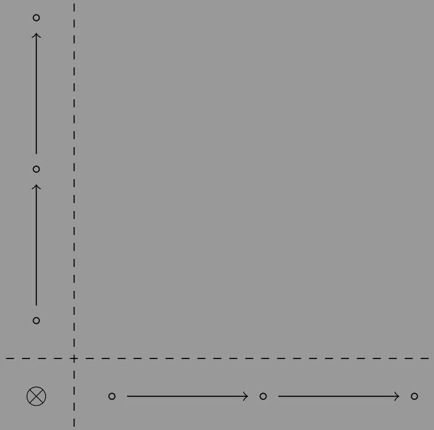
$$(K \otimes K')_n = \bigsqcup_{i+j=n} K_i \times K_j$$

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

$$\partial^{\pm}_k(x \otimes y) = \begin{cases} \partial^{\pm}_k(x) \otimes y & \text{if } 0 \leqslant k < i \\ x \otimes \partial^{\pm}_{k-p} y) & \text{if } i \leqslant k < n \end{cases}$$

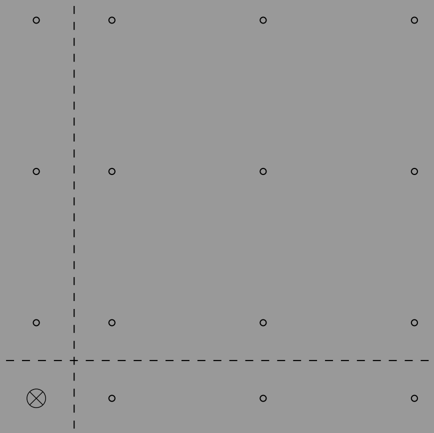
Example of tensor product

of precubical sets



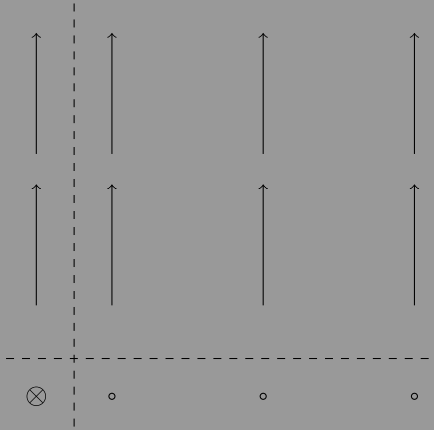
Example of tensor product

of precubical sets



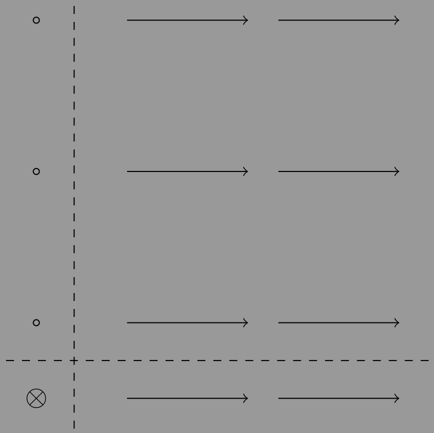
Example of tensor product

of precubical sets

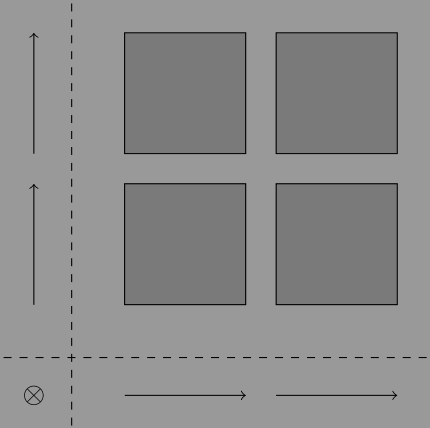


Example of tensor product

of precubical sets

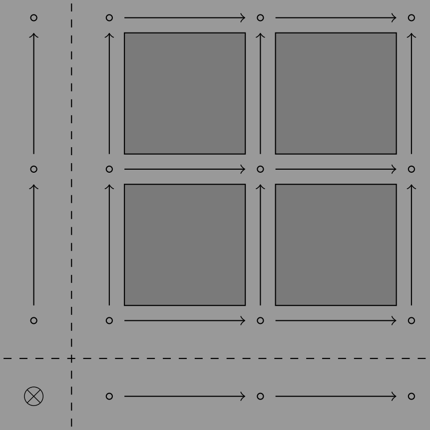


Example of tensor product
of precubical sets



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 M only makes sense under the sheaf condition:
  for all finite sequences s of length ℓ
  with elements in $\{1, \ldots, d\}$ and satisfying
  $\#\{i \mid s_i = k\} \leqslant M(k)$ for all $k \in \{1, \ldots, d\}$,
  the intermediate state of the interleaving execution at step
  ℓ from the inital 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

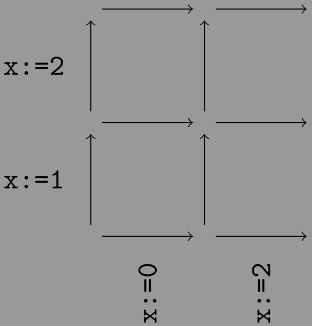
Control flow from tensor product of control flow graphs

$(\mathit{process}\ \ p)$ `x:=0 ; x:=2|`
$(\mathit{process}\ \ q)$ `x:=1 ; x:=2`

True concurrency - discrete version

Control flow from tensor product of control flow graphs

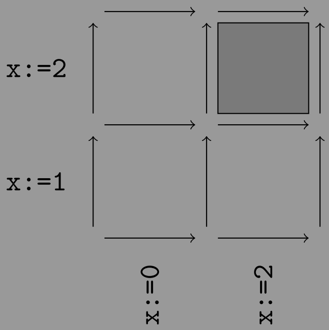
$$(\textit{process}\ \ p)\ \texttt{x:=0 ; x:=2|}$$
$$(\textit{process}\ \ q)\ \texttt{x:=1 ; x:=2}$$



True concurrency - discrete version

Control flow from tensor product of control flow graphs

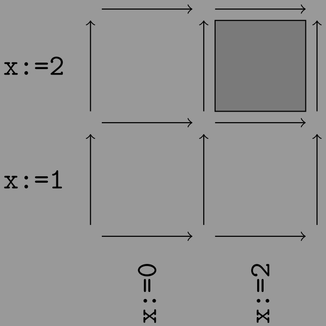
$(\textit{process}\ \ p)$ `x:=0 ; x:=2|`
$(\textit{process}\ \ q)$ `x:=1 ; x:=2`



True concurrency - discrete version

Control flow from tensor product of control flow graphs

$(\textit{process}\ \ p)$ `x:=0 ; x:=2|`
$(\textit{process}\ \ q)$ `x:=1 ; x:=2`



- $p + q \subseteq 2p + 2q$
- $2p + 2q$ is compatible yet $p + q$ is not

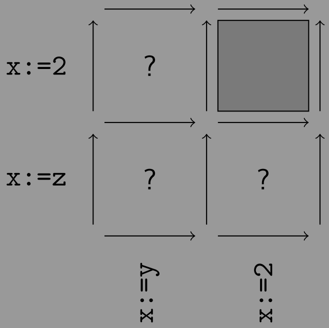
True concurrency - discrete version

Virtual Machine

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

True concurrency - discrete version
Virtual Machine

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

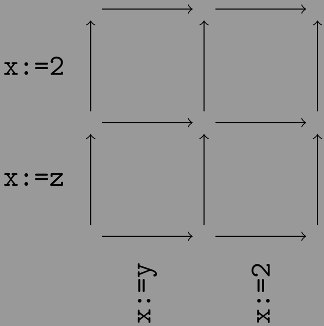


- filling square may depend on the current distribution

True concurrency - discrete version

Virtual Machine

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



- filling square may depend on the current distribution
- solution: actions with disjoint sets of occuring 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

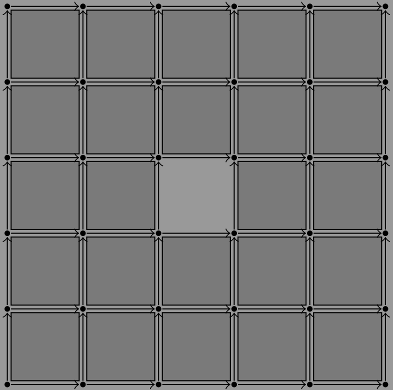
- $G_1 \otimes \cdots \otimes G_d$ tensor product of the control flow graphs
- Labelling all cubes of dimenson $1 \leqslant k \leqslant d$ by
  $\lambda(\alpha_1 \otimes \cdots \otimes \alpha_k) = \lambda_1(\alpha_1), \cdots, \lambda_k(\alpha_k)$ for $k \leqslant d$

The control flow precubical set

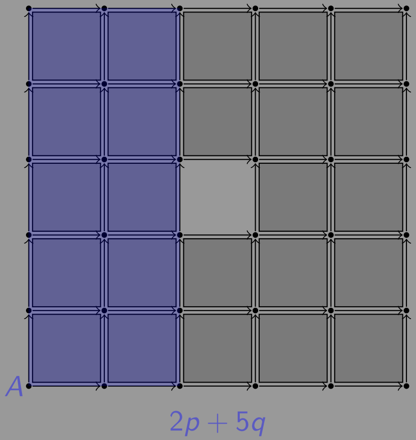
Middle-end representation taking race conditions into account

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

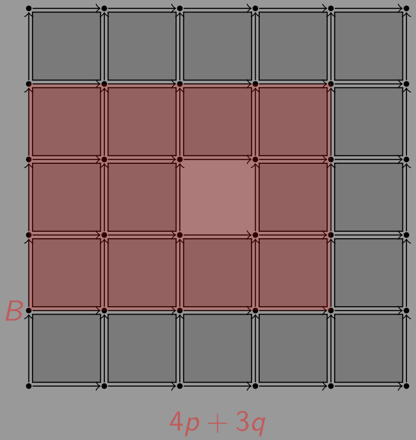
Dynamics



Dynamics



Dynamics



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

- *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 $\mathtt{x}$ of arity n is a resource offering $n-1$ tokens, each process can hold one token or more

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 bareers 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 bareers 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

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

Extending the middle-end representation

Potential function along a path

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

$$[\![\mathtt{P(a)}]\!]_\delta(x) = \begin{cases} \delta(x) & \text{if } x \neq \mathtt{a} \\ \delta(\mathtt{a}) + 1 & \text{if } x = \mathtt{a} \end{cases}$$

Extending the middle-end representation

Potential function along a path

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

$$[\![\mathtt{P(a)}]\!]_\delta(x) = \begin{cases} \delta(x) & \text{if } x \neq \mathtt{a} \\ \delta(\mathtt{a}) + 1 & \text{if } x = \mathtt{a} \end{cases}$$

$$[\![\mathtt{V(a)}]\!]_\delta(x) = \begin{cases} \delta(x) & \text{if } x \neq \mathtt{a} \\ \max\{0, \delta(\mathtt{a}) - 1\} & \text{if } x = \mathtt{a} \end{cases}$$

Extending the middle-end representation

Potential function along a path

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

$$[\![\mathtt{P(a)}]\!]_\delta(x) = \begin{cases} \delta(x) & \text{if } x \neq \mathtt{a} \\ \delta(\mathtt{a}) + 1 & \text{if } x = \mathtt{a} \end{cases}$$

$$[\![\mathtt{V(a)}]\!]_\delta(x) = \begin{cases} \delta(x) & \text{if } x \neq \mathtt{a} \\ \max\{0, \delta(\mathtt{a}) - 1\} & \text{if } x = \mathtt{a} \end{cases}$$

$$[\![\mathtt{W(a)}]\!] = \text{ignored}$$

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 γ on the distribution δ

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 γ on the distribution δ
- A process is conservative when for all paths γ, γ' on its cfg, all $x \in \mathcal{R}$ and all distributions δ

$$\partial^- \gamma = \partial^- \gamma' \text{ and } \partial^+ \gamma = \partial^+ \gamma' \;\Rightarrow\; [\![\gamma]\!] \cdot \delta(x) = [\![\gamma']\!] \cdot \delta(x)$$

Being conservative
is decidable

- approximation: a mapping from V to $2^{\mathbb{N}^{\mathcal{R}}}$

Being conservative

is decidable

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

Being conservative
is decidable

- approximation: a mapping from V to $2^{\mathbb{N}^{\mathcal{R}}}$
- $s \subseteq s'$ 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 $2^{\mathbb{N}^{\mathcal{R}}}$
- $s \subseteq s'$ 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 $2^{\mathbb{N}^{\mathcal{R}}}$
- $s \subseteq s'$ 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

$$s_{n+1}(v) = s_n(v) \cup \bigcup_{f \in A;\ \partial^+ f = v;\ \lambda(f) \in \{\mathtt{P},\mathtt{V}\}} f \cdot s_n(\partial^- f)$$

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 $\#s_n(v) \geqslant 2$: '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 $\#s_n(v) \geqslant 2$: '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 $F : K_0 \times \mathcal{R} \to \mathbb{N}$ 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
  $F(v,x) \geqslant \text{arity}(x)$ 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
  $F(v,x) \geqslant \text{arity}(x)$ for some semaphore or mutex x
- replace each n-cubes c whose edges carrying `W(x)`
  for some synchronization bareer `x` of arity n
  by an arrow $low(c) \to up(c)$

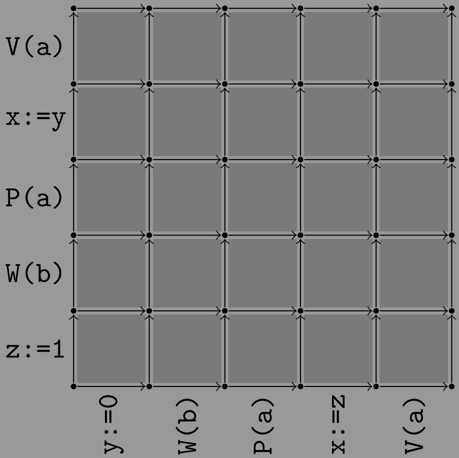
The control flow precubical set

taking P, V, and W into account

- Remove from K all v such that
  $F(v,x) \geqslant \mathrm{arity}(x)$ for some semaphore or mutex x
- replace each n-cubes c whose edges carrying `W(x)`
  for some synchronization bareer `x` of arity n
  by an arrow $low(c) \to up(c)$
- remove all arrows carrying `W(x)`
  for some synchronization bareer `x`

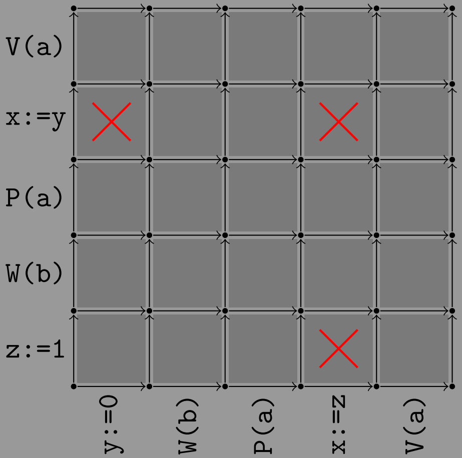
Control Flow Precubical Set: an example

y:=0.W(b).P(a).x:=z.V(a)|z:=0.W(b).P(a).x:=y.V(a)



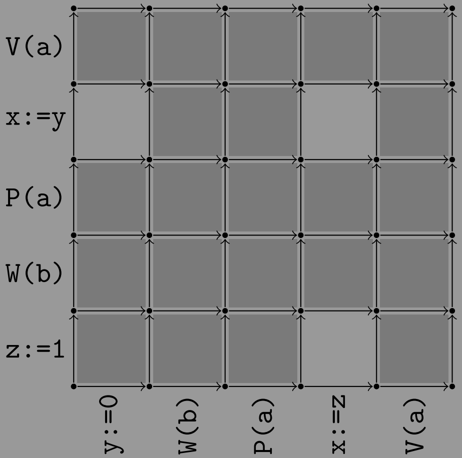
Control Flow Precubical Set: an example

`y:=0.W(b).P(a).x:=z.V(a)|z:=0.W(b).P(a).x:=y.V(a)`



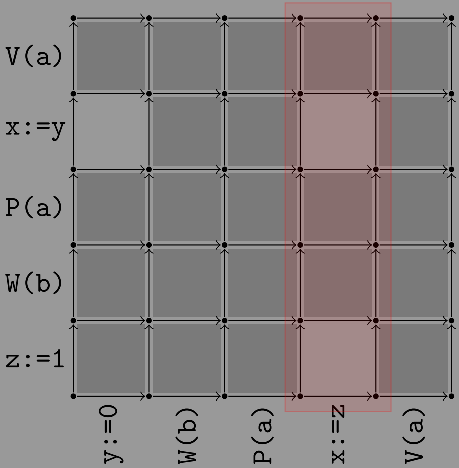
Control Flow Precubical Set: an example

`y:=0.W(b).P(a).x:=z.V(a)|z:=0.W(b).P(a).x:=y.V(a)`



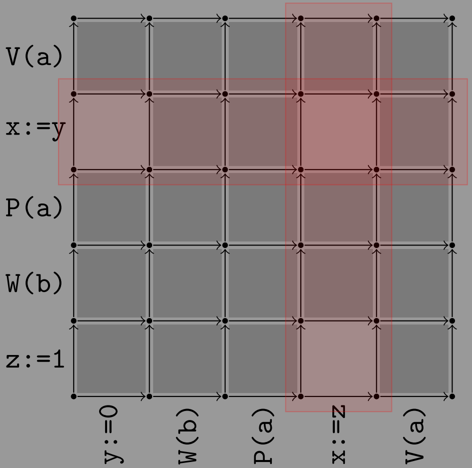
Control Flow Precubical Set: an example

`y:=0.W(b).P(a).x:=z.V(a)|z:=0.W(b).P(a).x:=y.V(a)`



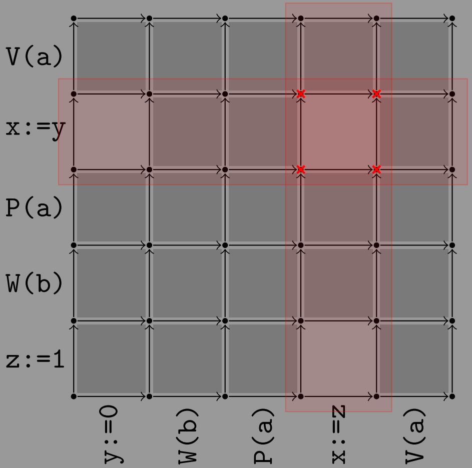
Control Flow Precubical Set: an example

`y:=0.W(b).P(a).x:=z.V(a)|z:=0.W(b).P(a).x:=y.V(a)`



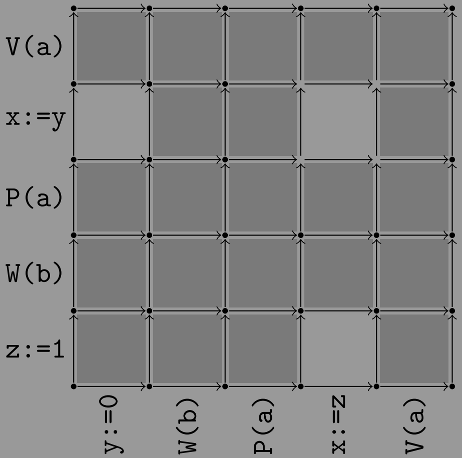
Control Flow Precubical Set: an example

`y:=0.W(b).P(a).x:=z.V(a)|z:=0.W(b).P(a).x:=y.V(a)`



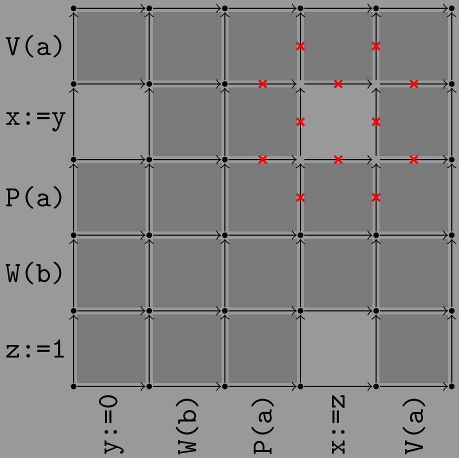
Control Flow Precubical Set: an example

y:=0.W(b).P(a).x:=z.V(a)|z:=0.W(b).P(a).x:=y.V(a)



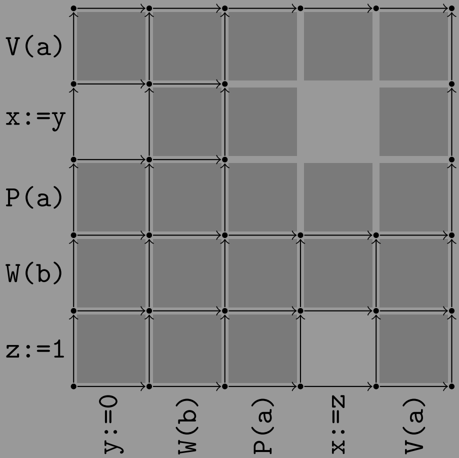
Control Flow Precubical Set: an example

y:=0.W(b).P(a).x:=z.V(a)|z:=0.W(b).P(a).x:=y.V(a)



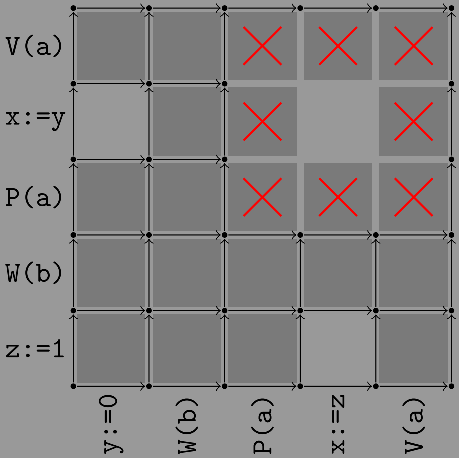
Control Flow Precubical Set: an example

y:=0.W(b).P(a).x:=z.V(a)|z:=0.W(b).P(a).x:=y.V(a)



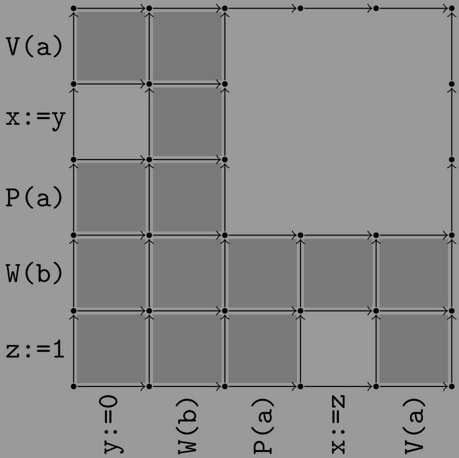
Control Flow Precubical Set: an example

`y:=0.W(b).P(a).x:=z.V(a)|z:=0.W(b).P(a).x:=y.V(a)`



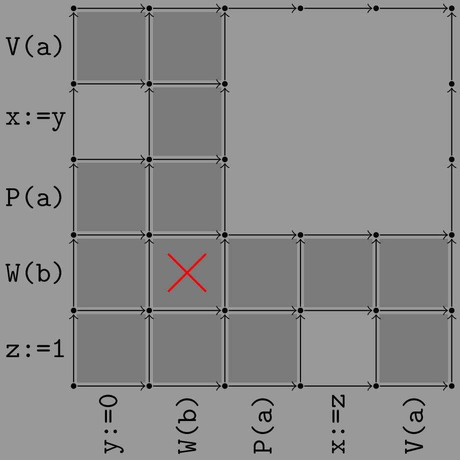
Control Flow Precubical Set: an example

y:=0.W(b).P(a).x:=z.V(a)|z:=0.W(b).P(a).x:=y.V(a)



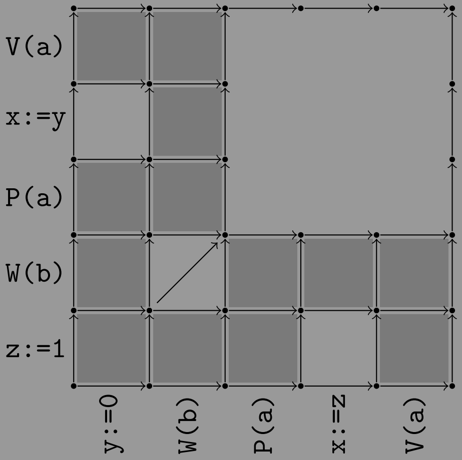
Control Flow Precubical Set: an example

`y:=0.W(b).P(a).x:=z.V(a)|z:=0.W(b).P(a).x:=y.V(a)`



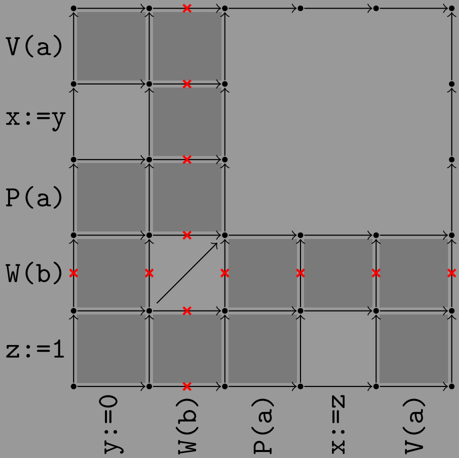
Control Flow Precubical Set: an example

y:=0.W(b).P(a).x:=z.V(a)|z:=0.W(b).P(a).x:=y.V(a)



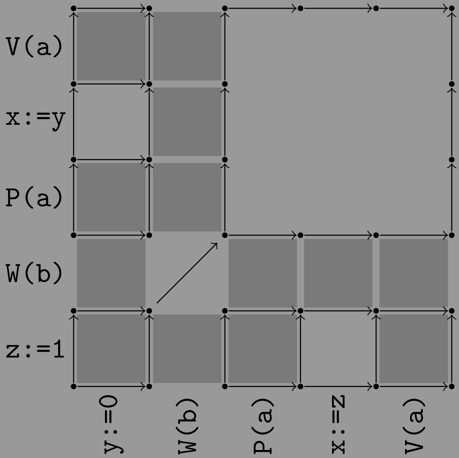
Control Flow Precubical Set: an example

y:=0.W(b).P(a).x:=z.V(a)|z:=0.W(b).P(a).x:=y.V(a)



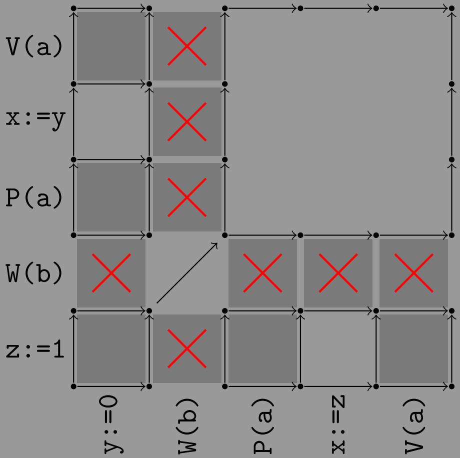
Control Flow Precubical Set: an example

`y:=0.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.W(b).P(a).x:=z.V(a)|z:=0.W(b).P(a).x:=y.V(a)