Towards Efficient Computation of Trace Spaces of Concurrent Programs

Samuel Mimram

CEA, LIST

Goal

When verifying a concurrent program,
there is a priori a large number of possible interleavings to check
(exponential in the number of processes)

Goal

When verifying a concurrent program,
there is a priori a large number of possible interleavings to check
(exponential in the number of processes)

Many executions are equivalent:
we want here to provide a *minimal number of execution traces*
which describe all the possible cases

Goal

When verifying a concurrent program,
there is a priori a large number of possible interleavings to check
(exponential in the number of processes)

Many executions are equivalent:
we want here to provide a *minimal number of execution traces*
which describe all the possible cases

Joint work with M. Raussen, L. Fajstrup, É. Goubault and
E. Haucourt.

Programs generate trace spaces

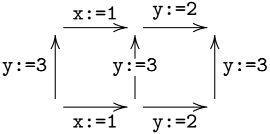
Consider the program

`x:=1;y:=2 | y:=3`

It can be scheduled in three different ways:

`y:=3;x:=1;y:=2` `x:=1;y:=3;y:=2` `x:=1;y:=2;y:=3`

Giving rise to the following graph of traces:



Programs generate trace spaces

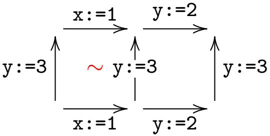
Consider the program

`x:=1;y:=2 | y:=3`

It can be scheduled in three different ways:

`y:=3;x:=1;y:=2`	`x:=1;y:=3;y:=2`	`x:=1;y:=2;y:=3`
$(x, y) = (1, 2)$	$(x, y) = (1, 2)$	$(x, y) = (1, 3)$

Giving rise to the following graph of traces:



homotopy: commutation / filled square

Mutexes

Concurrent access to shared variables should be protected using **mutexes** $a, b, \ldots$:

- P_a: lock the mutex a
- V_a: unlock the mutex a

Mutexes

Concurrent access to shared variables should be protected using **mutexes** $a, b, \ldots$:

- P_a: lock the mutex a
- V_a: unlock the mutex a

$$\texttt{x:=1;y:=2} \quad | \quad \texttt{y:=3}$$

Mutexes

Concurrent access to shared variables should be protected using **mutexes** $a, b, \ldots$:

- P_a: lock the mutex a
- V_a: unlock the mutex a

$$P_b\,;\texttt{x:=1};V_b\,;P_a\,;\texttt{y:=2};V_a \mid P_a\,;\texttt{y:=3};V_a$$

Mutexes

Concurrent access to shared variables should be protected using **mutexes** $a, b, \ldots$:

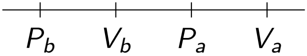
- P_a: lock the mutex a
- V_a: unlock the mutex a

$$P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$$

Geometric semantics

A program will be interpreted as a **directed space**:

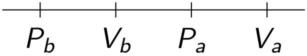
- $P_b.V_b.P_a.V_a$



Geometric semantics

A program will be interpreted as a **directed space**:

- $P_b.V_b.P_a.V_a$

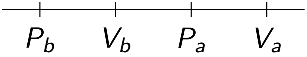


- $P_a.V_a$

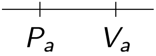
Geometric semantics

A program will be interpreted as a **directed space**:

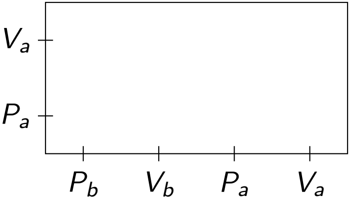
- $P_b.V_b.P_a.V_a$



- $P_a.V_a$



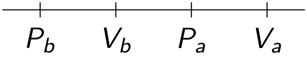
- $P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$



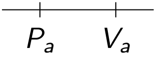
Geometric semantics

A program will be interpreted as a **directed space**:

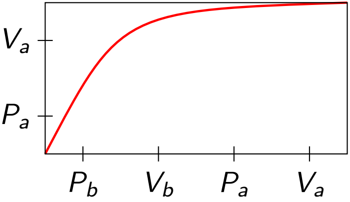
- $P_b.V_b.P_a.V_a$



- $P_a.V_a$



- $P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$

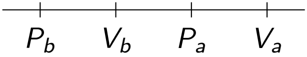


$P_a.P_b.V_a.V_b.P_a.V_a$

Geometric semantics

A program will be interpreted as a **directed space**:

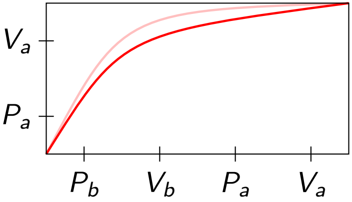
- $P_b.V_b.P_a.V_a$



- $P_a.V_a$

- $P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$

Homotopy

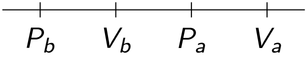


$P_a.P_b.V_a.V_b.P_a.V_a$

Geometric semantics

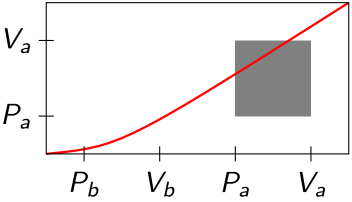
A program will be interpreted as a **directed space**:

- $P_b.V_b.P_a.V_a$



- $P_a.V_a$

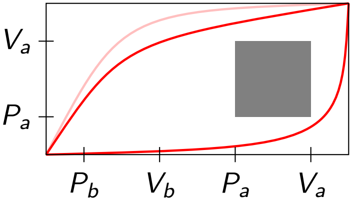
- $P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$



$P_b.V_b.P_a.P_a.V_a.V_a$ Forbidden region

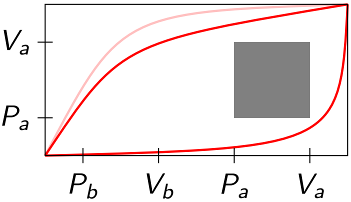
Schedulings

A **scheduling** is the homotopy class of a path.



Schedulings

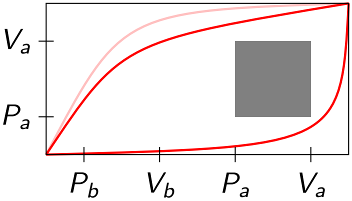
A **scheduling** is the homotopy class of a path.



We want to compute *a path in every scheduling*

Schedulings

A **scheduling** is the homotopy class of a path.

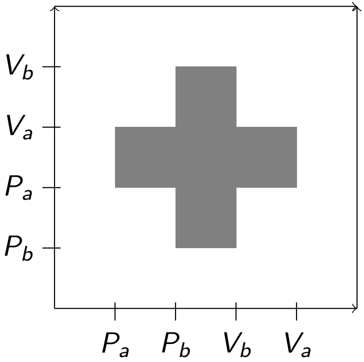


We want to compute *a path in every scheduling*

We do this by testing possible ways to go around forbidden regions:

The Swiss flag

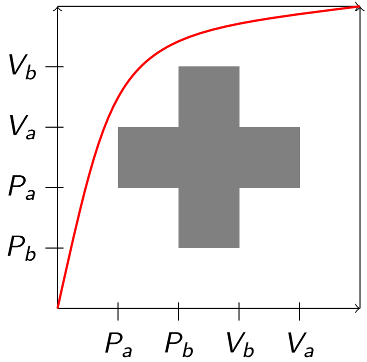
$$P_a.P_b.V_b.V_a \quad | \quad P_b.P_a.V_a.V_b$$



A forbidden region

The Swiss flag

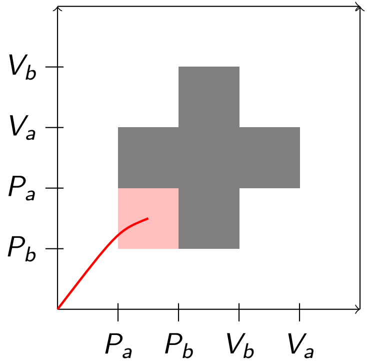
$$P_a.P_b.V_b.V_a \quad | \quad P_b.P_a.V_a.V_b$$



A trace: $P_b.P_a.V_a.P_a.V_b.P_b.V_b.V_a$

The Swiss flag

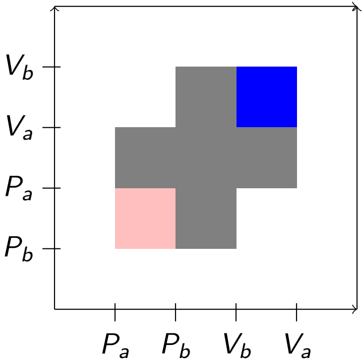
$$P_a.P_b.V_b.V_a \quad | \quad P_b.P_a.V_a.V_b$$



A deadlock: $P_b.P_a$

The Swiss flag

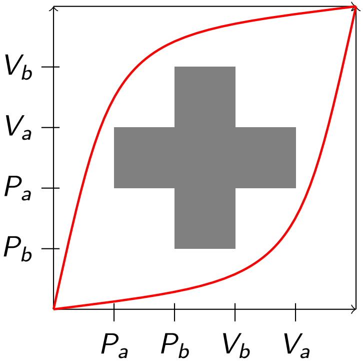
$$P_a.P_b.V_b.V_a \quad | \quad P_b.P_a.V_a.V_b$$



An unreachable region

The Swiss flag

$$P_a.P_b.V_b.V_a \quad | \quad P_b.P_a.V_a.V_b$$



Here we are interested in maximal paths modulo homotopy

Plan

1. Trace semantics of programs
2. Geometric semantics of programs
3. Computation of the trace space

Resources

We suppose fixed a set $\mathcal{R}$ of **resources** *a* with capacity $\kappa_a \in \mathbb{N}$.

The execution of programs are such that

1. a resource *a* cannot be locked (V_a) more than κ_a times
2. a resource *a* cannot be freed if it has not been locked

Example

A mutex is a resource of capacity 1.

Programs

We consider programs of the form:

$$p \quad ::= \quad \mathbf{1} \quad | \quad P_a \quad | \quad V_a \quad | \quad p.p \quad | \quad p|p \quad | \quad p+p \quad | \quad p^*$$

Programs

We consider programs of the form:

$$p \quad ::= \quad \mathbf{1} \quad | \quad P_a \quad | \quad V_a \quad | \quad p.p \quad | \quad p|p$$

We omit non-deterministic choice, loops

Programs

We consider programs of the form:

$$p \quad ::= \quad \mathbf{1} \;\mid\; P_a \;\mid\; V_a \;\mid\; p.p \;\mid\; p|p$$

We omit non-deterministic choice, loops, thread creation an join:

$$\begin{array}{lcll} A & ::= & P_a \mid V_a & \textit{actions} \\ t & ::= & A.t \mid \mathbf{1} & \textit{threads} \\ p & ::= & t|t|\ldots|t & \textit{programs} \end{array}$$

Trace semantics

The trace semantics of a program will be an **asynchronous graph**:

- a graph $G = (V, E)$ labeled by actions
- with an *independence relation* I

$$\begin{tikzcd}
y_1 \arrow[r, "B"] & z \\
x \arrow[u, "A"] \arrow[r, "B"'] & y_2 \arrow[u, "A"']
\end{tikzcd}$$

relating paths of length 2

Trace semantics

The trace semantics of a program will be an **asynchronous graph**:

- a graph $G = (V, E)$ labeled by actions
- with an *independence relation* I

$$\begin{tikzcd}
y_1 \arrow[r, "B"] & z \\
x \arrow[u, "A"] \arrow[r, "B"'] \arrow[ur, phantom, "\sim"] & y_2 \arrow[u, "A"']
\end{tikzcd}$$

relating paths of length 2

Trace semantics

The trace semantics of a program will be an **asynchronous graph**:

- a graph $G = (V, E)$ labeled by actions
- with an *independence relation* I

$$\begin{tikzcd}
y_1 \arrow[r, "B"] & z \\
x \arrow[u, "A"] \arrow[r, "B"'] \arrow[ur, phantom, "\sim"] & y_2 \arrow[u, "A"']
\end{tikzcd}$$

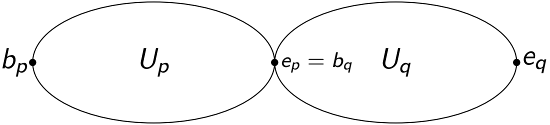
relating paths of length 2

Homotopy is the smallest congruence on paths containing I.

Trace semantics

To every program p we associate (U_p, b_p, e_p) defined by:

- $U_{\mathbf{1}}$: terminal graph
- U_{P_a}: $b_{P_a} \xrightarrow{P_a} e_{P_a}$ $\qquad$ U_{V_a}: $b_{P_a} \xrightarrow{V_a} e_{V_a}$
- $U_{p.q}$:



- $U_{p|q}$ is the "cartesian product" of U_p and U_q:

$$(x, y) \xrightarrow{A} (x', y) \qquad \text{when} \quad x \xrightarrow{A} x' \ \in U_p$$

$$(x, y') \xrightarrow{B} (x, y') \qquad \text{when} \quad y \xrightarrow{B} y' \ \in U_q$$

$$\begin{tikzcd}
(y, x') \arrow[r, "B"] & (y, y') \\
(x, x') \arrow[u, "A"] \arrow[r, "B"'] \arrow[ur, phantom, "\sim"] & (x, y') \arrow[u, "A"']
\end{tikzcd}$$

Trace semantics

Example:

$$P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$$

$$\begin{tikzcd}
{} \arrow[r, "P_b"] & {} \arrow[r, "V_b"] & {} \arrow[r, "P_a"] & {} \arrow[r, "V_a"] & e_p \\
{} \arrow[u, "V_a"] \arrow[r, "P_b"] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "V_a"] \arrow[r, "V_b"] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "V_a"] \arrow[r, "P_a"] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "V_a"] \arrow[r, "V_a"] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "V_a"'] \\
b_p \arrow[u, "P_a"] \arrow[r, "P_b"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a"] \arrow[r, "V_b"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a"] \arrow[r, "P_a"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a"] \arrow[r, "V_a"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a"']
\end{tikzcd}$$

Trace semantics

Example:

$$P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$$

$$\begin{tikzcd}
{} \arrow[r, "P_b"] & {} \arrow[r, "V_b"] & {} \arrow[r, "P_a"] & {} \arrow[r, "V_a"] & e_p \\
{} \arrow[u, "V_a"] \arrow[r, "P_b" description] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "V_a" description] \arrow[r, "V_b" description] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "V_a" description] \arrow[r, "P_a" description] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "V_a" description] \arrow[r, "V_a" description] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "V_a"'] \\
b_p \arrow[u, "P_a"] \arrow[r, "P_b"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a" description] \arrow[r, "V_b"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a" description] \arrow[r, "P_a"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a" description] \arrow[r, "V_a"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a"']
\end{tikzcd}$$

The **resource function** r_a associates to every vertex x:

number of releases of a - number locks of a

Trace semantics

Example:

$$P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$$

$$\begin{tikzcd}
{} \arrow[r, "P_b"] & {} \arrow[r, "V_b"] & {} \arrow[r, "P_a"] & {} \arrow[r, "V_a"] & e_p \\
{} \arrow[u, "V_a"] \arrow[r, "P_b"] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "V_a"] \arrow[r, "V_b"] \arrow[ur, phantom, "\sim"] & \color{red}{x} \arrow[u, "V_a"] \arrow[r, "P_a"] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "V_a"] \arrow[r, "V_a"] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "V_a"'] \\
b_p \arrow[u, "P_a"] \arrow[r, "P_b"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a"] \arrow[r, "V_b"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a"] \arrow[r, "P_a"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a"] \arrow[r, "V_a"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a"']
\end{tikzcd}$$

The **resource function** r_a associates to every vertex x:

number of releases of a - number locks of a

Ex: $r_a(x) = -1$, $r_b(x) = 0$

Trace semantics

Example:

$$P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$$

$$\begin{tikzcd}
{} \arrow[r, "P_b"] & {} \arrow[r, "V_b"] & {} \arrow[r, "P_a"] & {} \arrow[r, "V_a"] & e_p \\
{} \arrow[u, "V_a"] \arrow[r, "P_b"] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "V_a"] \arrow[r, "V_b"] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "V_a"] \arrow[r, "P_a"] \arrow[ur, phantom, "\sim"] & \color{red}{y} \arrow[u, "V_a"] \arrow[r, "V_a"] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "V_a"'] \\
b_p \arrow[u, "P_a"] \arrow[r, "P_b"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a"] \arrow[r, "V_b"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a"] \arrow[r, "P_a"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a"] \arrow[r, "V_a"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a"']
\end{tikzcd}$$

The **resource function** r_a associates to every vertex x:

number of releases of a - number locks of a

Ex: $r_a(y) = -2$, $r_b(y) = 0$

Trace semantics

Example:

$$P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$$

$$\begin{tikzcd}
{} \arrow[r, "P_b"] & {} \arrow[r, "V_b"] & {} \arrow[r, "P_a"] & {} \arrow[r, "V_a"] & e_p \\
{} \arrow[u, "V_a"] \arrow[r, "P_b"] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "V_a"] \arrow[r, "V_b"] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "V_a"] \arrow[r, "P_a"] \arrow[ur, phantom, "\sim"] & \color{red}{y} \arrow[u, "V_a"] \arrow[r, "V_a"] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "V_a"] \\
b_p \arrow[u, "P_a"] \arrow[r, "P_b"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a"] \arrow[r, "V_b"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a"] \arrow[r, "P_a"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a"] \arrow[r, "V_a"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a"]
\end{tikzcd}$$

The **resource function** r_a associates to every vertex x:

number of releases of a - number locks of a

Ex: $r_a(y) = -2 < -1 = \kappa_a$

Trace semantics

Trace semantics T_p:
U_p where we remove vertices x which do not satisfy

$$0 \leqslant r_a(x) + \kappa_a \leqslant \kappa_a$$

Example:

$$P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$$

$$\begin{tikzcd}
{} \arrow[r, "P_b"] & {} \arrow[r, "V_b"] & {} \arrow[r, "P_a"] & {} \arrow[r, "V_a"] & {} \\
{} \arrow[u, "V_a"] \arrow[r, "P_b"] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "V_a"] \arrow[r, "V_b"] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "V_a"] & & {} \arrow[u, "V_a"'] \\
{} \arrow[u, "P_a"] \arrow[r, "P_b"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a"] \arrow[r, "V_b"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "P_a"] \arrow[r, "P_a"'] & {} \arrow[r, "V_a"'] & {} \arrow[u, "P_a"']
\end{tikzcd}$$

Geometric semantics

The trace semantics is difficult to use to build intuitions. . .

In a similar way, one can define a **geometric semantics** where programs are interpreted by *directed spaces*.

Geometric semantics

A **path** in a topological space X is a continuous map $I = [0, 1] \to X$.

Definition

A **d-space** (X, dX) consists of

- a topological space X
- a set dX of paths in X, called *directed paths*, such that
  - *constant paths*: every constant path is directed,
  - *reparametrization*: dX is closed under precomposition with increasing maps $I \to I$, which are called *reparametrizations*,
  - *concatenation*: dX is closed under concatenation.

Geometric semantics

A **path** in a topological space X is a continuous map $I = [0,1] \to X$.

Definition

A **d-space** (X, dX) consists of

- a topological space X
- a set dX of paths in X, called *directed paths*, such that
  - *constant paths*: every constant path is directed,
  - *reparametrization*: dX is closed under precomposition with increasing maps $I \to I$, which are called *reparametrizations*,
  - *concatenation*: dX is closed under concatenation.

Example

$(X, \leqslant)$ space with a partial order, $dX = \{\text{increasing maps } I \to X\}$

$\vec{I}$: d-space induced by $[0,1]$

Geometric semantics

A **path** in a topological space X is a continuous map $I = [0, 1] \to X$.

Definition

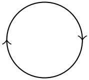
A **d-space** (X, dX) consists of

- a topological space X
- a set dX of paths in X, called *directed paths*, such that
  - *constant paths*: every constant path is directed,
  - *reparametrization*: dX is closed under precomposition with increasing maps $I \to I$, which are called *reparametrizations*,
  - *concatenation*: dX is closed under concatenation.

Example

$S^1 = \{\mathrm{e}^{\mathrm{i}\,\theta}\} 0 \leqslant \theta < 2\pi$

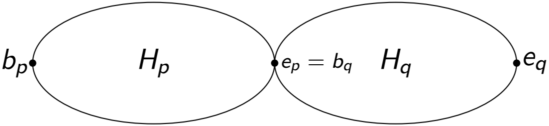
dS^1: $p(t) = \mathrm{e}^{\mathrm{i}\,f(t)}$ for some increasing function $f : I \to \mathbb{R}$



Geometric semantics

To each program p we associate a d-space (H_p, b_p, e_p):

- $H_{\mathbf{1}}$: $\bullet$
- $H_{P_a} = \vec{I}$ $\qquad$ $H_{V_a} = \vec{I}$
- $H_{p.q}$:

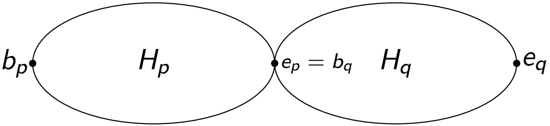


- $H_{p|q}$: $H_p \times H_q$, $b_{p|q} = (b_p, b_q)$, $e_{p|q} = (e_p, e_q)$

Geometric semantics

To each program p we associate a d-space (H_p, b_p, e_p):

- $H_\mathbf{1}$: $\bullet$
- $H_{P_a} = \vec{I}$ $\qquad H_{V_a} = \vec{I}$
- $H_{p.q}$:



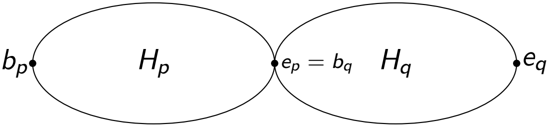
- $H_{p|q}$: $H_p \times H_q$, $b_{p|q} = (b_p, b_q)$, $e_{p|q} = (e_p, e_q)$

Resource function: $r_a(x) \in \mathbb{N}$ for each $a \in \mathcal{R}$ and point x

Geometric semantics

To each program p we associate a d-space (H_p, b_p, e_p):

- $H_{\mathbf{1}}$: $\bullet$
- $H_{P_a} = \vec{I}$ $\qquad$ $H_{V_a} = \vec{I}$
- $H_{p.q}$:



- $H_{p|q}$: $H_p \times H_q$, $b_{p|q} = (b_p, b_q)$, $e_{p|q} = (e_p, e_q)$

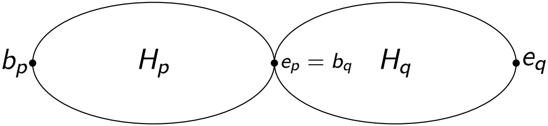
Resource function: $r_a(x) \in \mathbb{N}$ for each $a \in \mathcal{R}$ and point x

Forbidden region:
$F_p = \{x \in H_p \;/\; \exists a \in \mathcal{R}, \quad r_a(x) + \kappa_a < 0 \quad \text{or} \quad r_a(x) > 0\}$

Geometric semantics

To each program p we associate a d-space (H_p, b_p, e_p):

- $H_{\mathbf{1}}$: $\bullet$
- $H_{P_a} = \vec{I}$ $\qquad$ $H_{V_a} = \vec{I}$
- $H_{p.q}$:



- $H_{p|q}$: $H_p \times H_q$, $b_{p|q} = (b_p, b_q)$, $e_{p|q} = (e_p, e_q)$

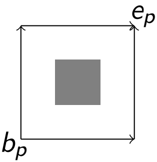
Resource function: $r_a(x) \in \mathbb{N}$ for each $a \in \mathcal{R}$ and point x

Forbidden region:
$F_p = \{x \in H_p \ / \ \exists a \in \mathcal{R}, \quad r_a(x) + \kappa_a < 0 \quad \text{or} \quad r_a(x) > 0\}$

Geometric semantics: $G_p = H_p \setminus F_p$

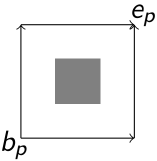
Examples of geometric semantics

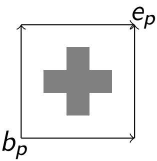
$P_a.V_a|P_a.V_a$



Examples of geometric semantics

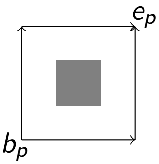
$P_a.V_a|P_a.V_a$ $P_a.P_b.V_b.V_a|P_b.P_a.V_a.V_b$



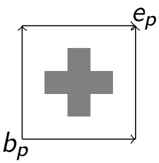


Examples of geometric semantics

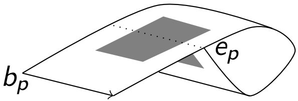
$P_a.V_a|P_a.V_a$



$P_a.P_b.V_b.V_a|P_b.P_a.V_a.V_b$

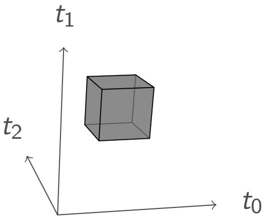


$P_a.(V_a.P_a)^*|P_a.V_a$

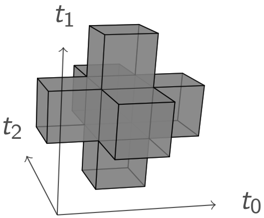


Examples of geometric semantics

$P_a.V_a|P_a.V_a|P_a.V_a$
$(\kappa_a = 2)$



$P_a.V_a|P_a.V_a|P_a.V_a$
$(\kappa_a = 1)$



Geometric realization

The two semantics are "essentially the same": the geometric semantics is the **geometric realization** of a *cubical set*

$$G_p \quad = \quad \int^{n \in \square} T_p(n) \cdot \vec{I}^n$$

Proposition

Given a program p, with T_p as trace semantics and G_p as geometric semantics,

- *every path $\pi : b \to e$ in T_p induces a path $\overline{\pi} : b \to e$ in G_p,*
- *$\pi \sim \rho$ in T_p implies $\overline{\pi} \sim \overline{\rho}$ in G_p*
- *every path ρ of G_p is homotopic to a path $\overline{\pi}$ (π path in G_p)*

Computing the trace space

Goal

Given a program p, we describe an algorithm to compute a trace in each equivalence class of traces $\pi : b_p \to e_p$ up to homotopy in G_p.

The proposition before ensures that it is the same to compute this in the trace semantics or in the geometric semantics.

The algorithm

Suppose given a program

$$p \quad = \quad p_0|p_1|\dots|p_{n-1}$$

with n *threads*.

The algorithm

Suppose given a program

$$p \quad = \quad p_0 | p_1 | \ldots | p_{n-1}$$

with n *threads*.

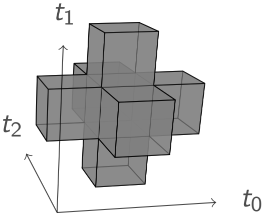
Under mild assumptions, the geometric semantics is of the form

$$G_p \quad = \quad \vec{I}^n \setminus \bigcup_{i=0}^{l-1} R^i$$

where

$$R^i = \prod_{j=0}^{n-1}]x^i_j, y^i_j[$$

are l *open rectangles*.



The algorithm

Under mild assumptions, the geometric semantics is of the form

$$G_p \quad = \quad \vec{I}^n \setminus \bigcup_{i=0}^{l-1} R^i$$

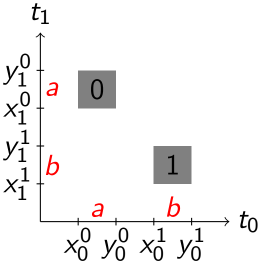
where

$$R^i = \prod_{j=0}^{n-1}]x_j^i, y_j^i[$$

are l *open rectangles*.

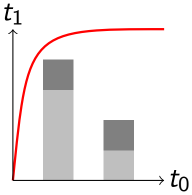
Example

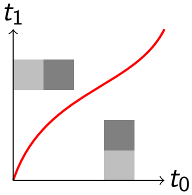
$$P_a.V_a.P_b.V_b|P_b.V_b.P_a.V_a$$

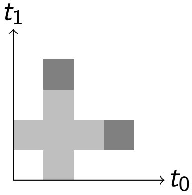


The algorithm

The main idea of the algorithm is to extend the forbidden cubes downwards in various directions and look whether there is a path from b to e in the resulting space.



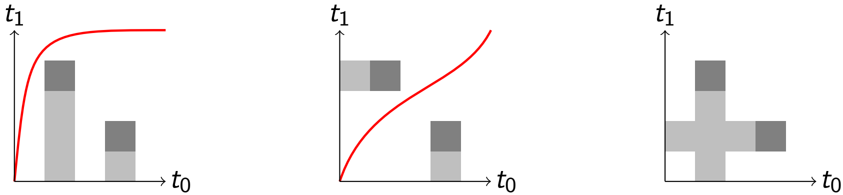




By combining those information, we will be able to compute traces modulo homotopy.

The algorithm

The main idea of the algorithm is to extend the forbidden cubes downwards in various directions and look whether there is a path from b to e in the resulting space.



By combining those information, we will be able to compute traces modulo homotopy.

The directions in which to extend the holes will be coded by boolean matrices M.

The index poset

$\mathcal{M}_{l,n}$: boolean matrices with l rows and n columns.

The index poset

$\mathcal{M}_{l,n}$: boolean matrices with l rows and n columns.

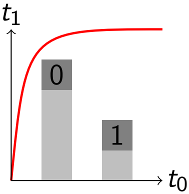
X_M:

space obtained by *extending*
for every (i,j) such that $M(i,j)=1$
the forbidden cube i downwards
in every direction other than j

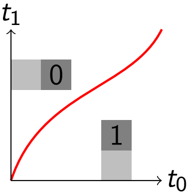
The index poset

$\mathcal{M}_{l,n}$: boolean matrices with l rows and n columns.

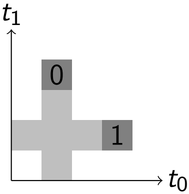
X_M: space obtained by *extending*
for every (i,j) such that $M(i,j)=1$
the forbidden cube i downwards
in every direction other than j



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



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

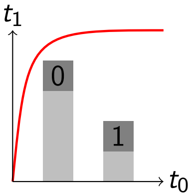


$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

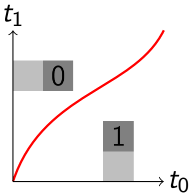
The index poset

$\mathcal{M}_{l,n}$: boolean matrices with l rows and n columns.

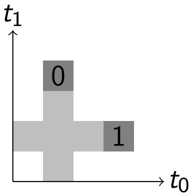
X_M: space obtained by *extending*
for every (i,j) such that $M(i,j)=1$
the forbidden cube i downwards
in every direction other than j



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



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



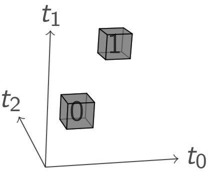
$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$\Psi : \mathcal{M}_{l,n} \to \{0,1\}$:

- $\Psi(M)=0$ if there is a path $b \to e$: M is alive
- $\Psi(M)=1$ if there is no path $b \to e$: M is dead

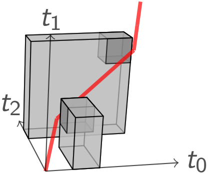
The index poset

$$P_a.V_a.P_b.V_b \quad \Big| \quad P_a.V_a.P_b.V_b \quad \Big| \quad P_a.V_a.P_b.V_b$$



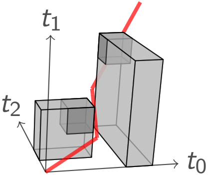
$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

alive



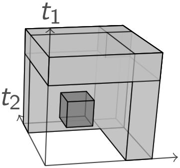
$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

alive



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

alive



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

dead

The index poset

- $\mathcal{M}_{l,n}$ is equipped with the pointwise ordering
- Ψ is increasing: more $1 \Rightarrow$ more obstructions
- $\mathcal{M}^R_{l,n}$: matrices with non-null rows
- $\mathcal{M}^C_{l,n}$: matrices with unit column vectors

The index poset

- $\mathcal{M}_{l,n}$ is equipped with the pointwise ordering
- Ψ is increasing: more 1 ⇒ more obstructions
- $\mathcal{M}^R_{l,n}$: matrices with non-null rows
- $\mathcal{M}^C_{l,n}$: matrices with unit column vectors

Definition

The **index poset** $\mathcal{C}(X) = \{M \in \mathcal{M}^R_{l,n} \;/\; \Psi(M) = 0\}$ (the alive matrices).

The index poset

- $\mathcal{M}_{l,n}$ is equipped with the pointwise ordering
- Ψ is increasing: more 1 $\Rightarrow$ more obstructions
- $\mathcal{M}_{l,n}^R$: matrices with non-null rows
- $\mathcal{M}_{l,n}^C$: matrices with unit column vectors

Definition

The **index poset** $\mathcal{C}(X) = \{M \in \mathcal{M}_{l,n}^R \ / \ \Psi(M) = 0\}$
(the alive matrices).

Definition

The **dead poset** $D(X) = \{M \in \mathcal{M}_{l,n}^C \ / \ \Psi(M) = 1\}$.

The index poset

- $\mathcal{M}_{l,n}$ is equipped with the pointwise ordering
- Ψ is increasing: more 1 $\Rightarrow$ more obstructions
- $\mathcal{M}_{l,n}^R$: matrices with non-null rows
- $\mathcal{M}_{l,n}^C$: matrices with unit column vectors

Definition

The **index poset** $\mathcal{C}(X) = \{M \in \mathcal{M}_{l,n}^R \ / \ \Psi(M) = 0\}$ (the alive matrices).

Definition

The **dead poset** $D(X) = \{M \in \mathcal{M}_{l,n}^C \ / \ \Psi(M) = 1\}$.

$D(X) \quad \rightsquigarrow \quad \mathcal{C}(X) \quad \rightsquigarrow$ homotopy classes of traces

The dead poset

Proposition

A matrix $M \in \mathcal{M}_{l,n}^{C}$ *is in* $D(X)$ *iff it satisfies*

$$\forall (i,j) \in [0:l[\times [0:n[, \qquad M(i,j) = 1 \quad \Rightarrow \quad x_j^i < \min_{i' \in R(M)} y_j^{i'}$$

where $R(M)$*: indexes of non-null rows of* M*.*

The dead poset

Proposition

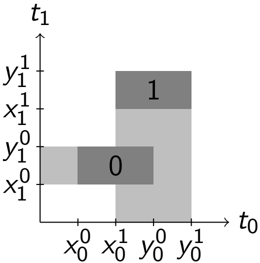
A matrix $M \in \mathcal{M}_{l,n}^{C}$ is in $D(X)$ iff it satisfies

$$\forall (i,j) \in [0:l[\times[0:n[, \qquad M(i,j)=1 \quad \Rightarrow \quad x_j^i < \min_{i' \in R(M)} y_j^{i'}$$

where $R(M)$: indexes of non-null rows of M.

Example

M is dead:



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

$$\begin{aligned} x_1^0 &= 1 < 2 = \min(y_1^0, y_1^1) \\ x_0^1 &= 2 < 3 = \min(y_0^0, y_0^1) \end{aligned}$$

The index poset

Proposition

A matrix M is in $\mathcal{C}(X)$ iff for every $N \in D(X)$, $N \not\leq M$.

The index poset

Proposition

A matrix M is in $\mathcal{C}(X)$ iff for every $N \in D(X)$, $N \nleq M$.

Remark

$N \nleq M$: there exists (i,j) s.t. $N(i,j) = 1$ and $M(i,j) = 0$.

Remark

Since $\mathcal{C}(X)$ is downward closed it will be enough to compute the set $\mathcal{C}_{\max}(X)$ of maximal alive matrices.

Remark

The index poset contains all the geometrical information!

Connected components

$M \wedge N$: pointwise min of M and N

Definition

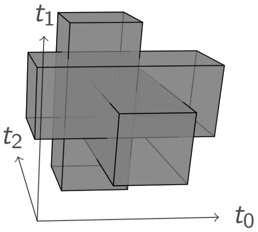
Two matrices M and N are **connected** when $M \wedge N$ does not contain any null row.

Proposition

The connected components of $\mathcal{C}(X)$ are in bijection with homotopy classes of traces $b \to e$ in X.

Dining philosophers

n processes p_k in parallel:



$$p_k = P_{a_k}.P_{a_{k+1}}.V_{a_k}.V_{a_{k+1}}$$

n	sched.	ALCOOL (s)	ALCOOL (MB)	SPIN (s)	SPIN (MB)
8	254	0.1	0.8	0.3	12
9	510	0.8	1.4	1.5	41
10	1022	5	4	8	179
11	2046	32	9	42	816
12	4094	227	26	313	3508
13	8190	1681	58	∞	∞
14	16382	13105	143	∞	∞

How do we extend this methodology to program with loops?

Loops

Given a thread p, we write p^* for its looping: `while(...){`p`}`.

Loops

Given a thread p, we write p^* for its looping: `while(...){`p`}`.

Given a program p with n threads:

$$p \quad = \quad p_1 | p_2 | \ldots | p_n$$

we write p^* for

$$p^* \quad = \quad p_1^* | p_2^* | \ldots | p_n^*$$

Loops

Given a thread p, we write p^* for its looping: `while(...){`p`}`.

Given a program p with n threads:

$$p \quad = \quad p_1 | p_2 | \ldots | p_n$$

we write p^* for

$$p^* \quad = \quad p_1^* | p_2^* | \ldots | p_n^*$$

Notice that the geometric semantics X_{p^*} can be deduced from the semantics of p by glueing copies of X_p in every direction:

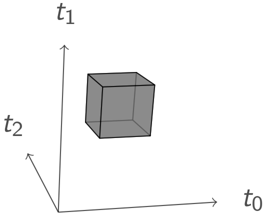
$$p_i^* \quad = \quad p_i . p_i . p_i \ldots$$

Deloopings

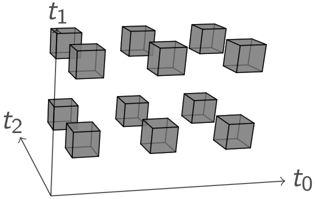
Notice that the geometric semantics X_{p^*} can be deduced from the semantics of p by glueing copies of X_p in every direction.

Example

Consider the program $p = q|q|q$ with $q = P_a.V_a$ (and a of arity 3):



X_p



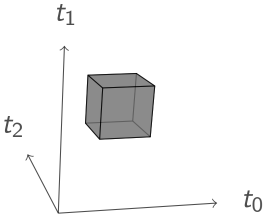
X_{p^*}

Deloopings

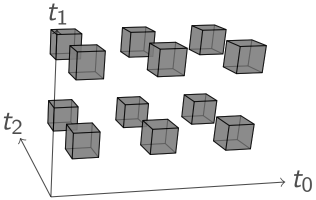
Notice that the geometric semantics X_{p^*} can be deduced from the semantics of p by glueing copies of X_p in every direction.

Example

Consider the program $p = q|q|q$ with $q = P_a.V_a$ (and a of arity 3):



X_p



$X_{p^{(3,2,2)}}$

Finite deloopings:

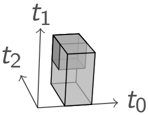
$$X_{p^{(3,2,2)}} \quad = \quad (Y \oplus_1 Y) \oplus_2 (Y \oplus_1 Y) \qquad \text{with} \qquad Y = X_p \oplus_0 X_p \oplus_0 X_p$$

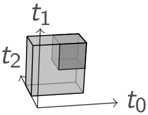
Schedulings

Similarly, given schedulings

$$M = \begin{pmatrix}1 & 0 & 0\end{pmatrix} \qquad \text{and} \qquad N = \begin{pmatrix}0 & 0 & 1\end{pmatrix}$$

of the previous program p



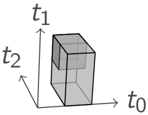


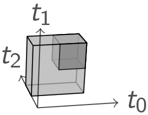
Schedulings

Similarly, given schedulings

$$M = \begin{pmatrix}1 & 0 & 0\end{pmatrix} \qquad \text{and} \qquad N = \begin{pmatrix}0 & 0 & 1\end{pmatrix}$$

of the previous program p



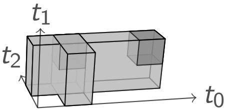


we write

$$M \oplus_0 N = \begin{pmatrix}1 & 0 & 0\\ 0 & 0 & 1\end{pmatrix}$$

for the following scheduling of $X_p^{(2,1,1)} = X_p \oplus_0 X_p$

$X_{M\oplus_0 N} =$

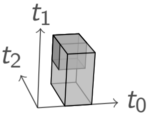


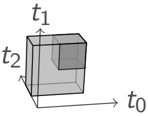
Schedulings

Similarly, given schedulings

$$M = \begin{pmatrix}1 & 0 & 0\end{pmatrix} \qquad \text{and} \qquad N = \begin{pmatrix}0 & 0 & 1\end{pmatrix}$$

of the previous program p



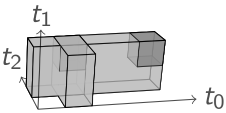


we write

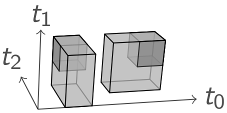
$$M \oplus_0 N = \begin{pmatrix}1 & 0 & 0\\ 0 & 0 & 1\end{pmatrix}$$

for the following scheduling of $X_p^{(2,1,1)} = X_p \oplus_0 X_p$

$X_{M\oplus_0 N} =$



$\neq$

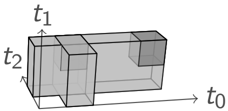


$= X_M \oplus_0 X_N$

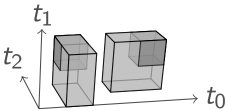
Shadows

In fact, scheduling drop "*shadows*" on previous schedulings

$X_{M\oplus_0 N} =$



$\neq$

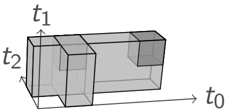


$= X_M \oplus_0 X_N$

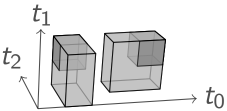
Shadows

In fact, scheduling drop "*shadows*" on previous schedulings

$X_{M\oplus_0 N} =$



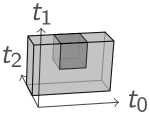
$\neq$



$= X_M \oplus_0 X_N$

Write $X_{M|_j}$ for the **shadow** projected by scheduling M in direction j:

$X_{N|_0} \quad =$



so that

$$X_{M\oplus_j N} \quad = \quad (X_M \cap X_{N|_j}) \otimes_j X_N$$

Alive matrices for programs with loops

Every scheduling M of a delooping of X_p is composed by glueing *submatrices* $(M_{i_1,\dots,i_n})$.

Alive matrices for programs with loops

Every scheduling M of a delooping of X_p is composed by glueing *submatrices* $(M_{i_1,\ldots,i_n})$.

If X_M contains a deadlock then some subspace $X_{(M_{i_1,\ldots,i_n})}$ contains a deadlock:

Lemma

If a matrix M is alive then all its submatrices are alive.

Alive matrices for programs with loops

Every scheduling M of a delooping of X_p is composed by glueing *submatrices* $(M_{i_1,\ldots,i_n})$.

If X_M contains a deadlock then some subspace $X_{(M_{i_1,\ldots,i_n})}$ contains a deadlock:

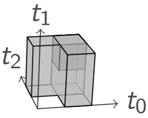
Lemma

If a matrix M is alive then all its submatrices are alive.

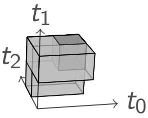
`The converse is not true!`

Shadows can create deadlocks

The following matrices P and Q coding the schedulings



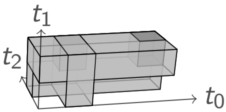
X_P



X_Q

of p are alive, however the matrix $P \oplus_0 Q$ is dead:

$X_{P \oplus_0 Q} \quad =$



The shadow automaton

We construct an automaton which describes all the schedulings possible in the future (which won't create deadlocks by their shadow): given a scheduling M and a direction j, it describes all the matrices N such that $M \oplus_j N$ is alive.

The shadow automaton

Definition

The **shadow automaton** of a program p is a non-deterministic automaton whose

- states are shadows
- transitions $N \xrightarrow{j,M} N'$ are labeled by a direction j (with $0 \leqslant j < n$) and a scheduling M

defined as the smallest automaton

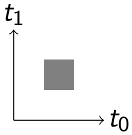
- containing the empty scheduling $\emptyset$
- and such that for every state N', for every direction j and for every scheduling M such that the scheduling $M \cup N'$ is alive, and M is maximal with this property, there is a transition $N \xrightarrow{j,M} N'$ with $N = (M \cup N')|_j$.

All the states of the automaton are both initial and final.

The shadow automaton

For instance consider the program $p = P_a.V_a|P_a.V_a$

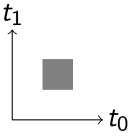
$X_p \quad =$



The shadow automaton

For instance consider the program $p = P_a.V_a|P_a.V_a$

$X_p =$

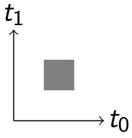


There are two maximal schedulings

The shadow automaton

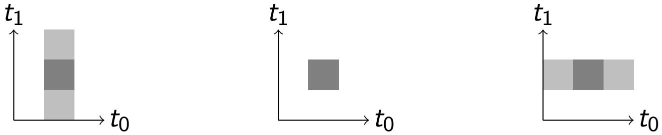
For instance consider the program $p = P_a.V_a|P_a.V_a$

$X_p =$



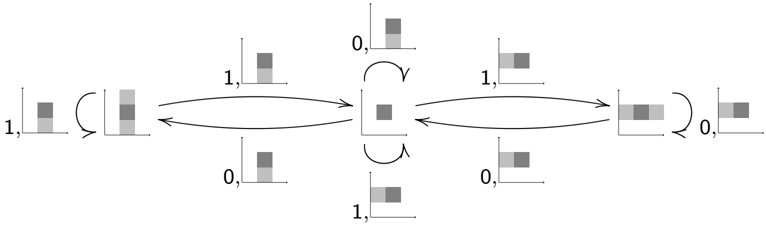
There are two maximal schedulings

which can drop three possible shadows



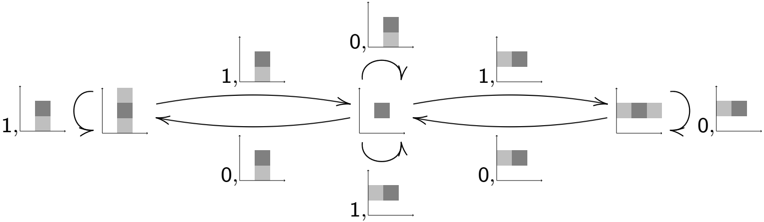
The shadow automaton

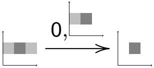
The shadow automaton of p is

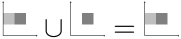


The shadow automaton

The shadow automaton of p is



For instance, the transition  is computed as follows:

- consider the shadow $M =$ 
- compute its shadow in direction 0: 

The shadow automaton

Theorem

Given a program p to any total path in a delooping of p is represented by a path in the shadow automaton of p such that

- *every path in the automaton comes from a total path in* $X_{p^?}$
- *if two total paths in* $X_{p^?}$ *correspond to the same path in the automaton then they are homotopic*

Paths in the shadow automaton describe homotopy classes in deloopings of p.

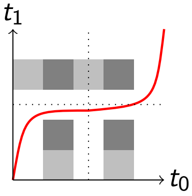
The shadow automaton

Theorem

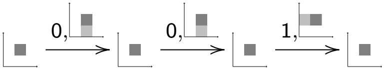
Given a program p to any total path in a delooping of p is represented by a path in the shadow automaton of p such that

- *every path in the automaton comes from a total path in $X_{p^?}$*
- *if two total paths in $X_{p^?}$ correspond to the same path in the automaton then they are homotopic*

Paths in the shadow automaton describe homotopy classes in deloopings of p.



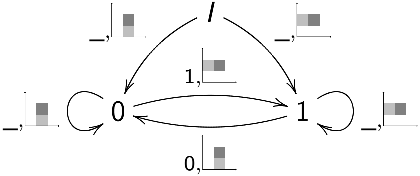
is represented by



Reducing the size of the automaton

The shadow automaton is too big:

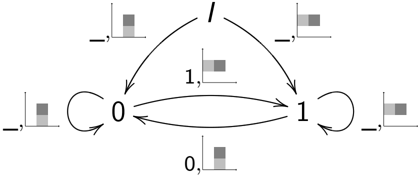
- we can determinize it:



Reducing the size of the automaton

The shadow automaton is too big:

- we can determinize it:



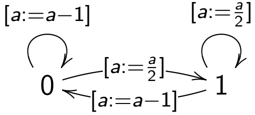
- two distinct paths in the automaton can represent the same homotopy class of paths: we can quotient paths under connexity.

An application to static analysis

The program

$$p^* \quad = \quad \Big(P_a.a := a - 1.V_a\Big)^*\Big|\Big(P_a.\Big(a := \frac{a}{2}\Big).V_a\Big)^*$$

induces the automaton

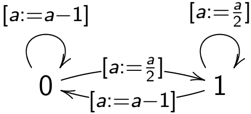


An application to static analysis

The program

$$p^* \quad = \quad \Big(P_a.a := a - 1.V_a\Big)^* \Big| \Big(P_a.\Big(a := \frac{a}{2}\Big).V_a\Big)^*$$

induces the automaton



and thus the set of equations

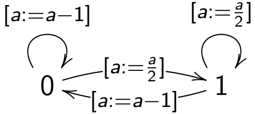
$$\begin{cases} A_0 & = & I \cup (A_0 - 1) \cup (A_1 - 1) \\ A_1 & = & I \cup \frac{A_1}{2} \cup \frac{A_0}{2} \end{cases}$$

An application to static analysis

The program

$$p^* \quad = \quad \Big(P_a.a := a - 1.V_a\Big)^* \Big| \Big(P_a.\Big(a := \frac{a}{2}\Big).V_a\Big)^*$$

induces the automaton



and thus the set of equations

$$\begin{cases} A_0 &= I \cup (A_0 - 1) \cup (A_1 - 1) \\ A_1 &= I \cup \frac{A_1}{2} \cup \frac{A_0}{2} \end{cases}$$

which admits a least fixed point

$$A_0^\infty \quad = \quad A_1^\infty \quad = \quad]-\infty, 1]$$

An example: the two-phase protocol

We consider n programs locking l resources:

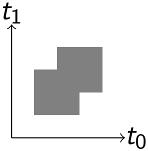
$$p_{n,l} = q|q|\ldots|q \qquad \text{with} \qquad q = P_{a_1}.\ldots.P_{a_l}.V_{a_1}.\ldots.V_{a_l}$$

An example: the two-phase protocol

We consider n programs locking l resources:

$$p_{n,l} = q|q|\ldots|q \qquad \text{with} \qquad q = P_{a_1}.\ldots.P_{a_l}.V_{a_1}.\ldots.V_{a_l}$$

For instance, $p_{2,2} = q|q$ is

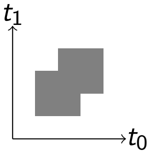


An example: the two-phase protocol

We consider n programs locking l resources:

$$p_{n,l} = q|q|\dots|q \qquad \text{with} \qquad q = P_{a_1}.\dots.P_{a_l}.V_{a_1}.\dots.V_{a_l}$$

For instance, $p_{2,2} = q|q$ is



We get the following results compared to spin:

n	l	s	t	s'	t'	s''	t''	s_{SPIN}	t_{SPIN}
2	1	3	8	3	10	1	1	58	65
2	2	3	8	3	10	1	1	112	129
2	3	3	8	3	10	1	1	180	209
3	1	19	90	4	24	1	1	171	218
3	2	19	90	4	24	1	1	441	602
3	3	19	90	4	24	1	1	817	1128

About geometric models

Was the use of the geometric model necessary?

About geometric models

Was the use of the geometric model necessary?

$\Rightarrow$ No: we could have formulated it directly on the trace space

About geometric models

Was the use of the geometric model necessary?

$\Rightarrow$ No: we could have formulated it directly on the trace space

Was the geometric model useful?

About geometric models

Was the use of the geometric model necessary?

⇒ No: we could have formulated it directly on the trace space

Was the geometric model useful?

⇒ Yes: it would have been very hard to think of the algorithm without “seeing” the spaces

About geometric models

Was the use of the geometric model necessary?

$\Rightarrow$ No: we could have formulated it directly on the trace space

Was the geometric model useful?

$\Rightarrow$ Yes: it would have been very hard to think of the algorithm without "seeing" the spaces

$\Rightarrow$ Yes: computers are much better at manipulating booleans than complex algebraic structures

Future works

We compute **one execution trace in each homotopy class**.

Future works

We compute **one execution trace in each homotopy class**.

What remains to do:

- use these trace to do static analysis
  (e.g. abstract interpretation)
- speed improvements
- implementation improvements (e.g. GPU)
- lots of work remain to be done on the theoretical side