A QUICK OVERVIEW
OF
CONCURRENCY THEORY
Handbook of Logic in Computer Science
VOLUME 4

S. ABRAMSKY, DOV M. GABBAY,
and T. S. E. MAIBAUM

OXFORD SCIENCE PUBLICATIONS
PARALLEL AUTOMATA META LANGUAGE
Syntax
Paradigm


\[^1\text{Portable Operating Systems Interface, X is a reference to Unix}\]
Paradigm


- The Dijkstra’s language is a parallel extension of ALGOL60 with P (lock/take), V (unlock/release), and parbegin ... parend

1Portable Operating Systems Interface, X is a reference to Unix
Paradigm


- The Dijkstra’s language is a parallel extension of ALGOL60 with
  P (lock/take), V (unlock/release), and *parbegin ... parend*
- Shared memory (e.g. Parallel RAM - Concurrent Read Exclusive Write)

\(^1\)Portable Operating Systems Interface, X is a reference to Unix
Paradigm


- The Dijkstra’s language is a parallel extension of ALGOL60 with
  P (lock/take), V (unlock/release), and parbegin ... parend
- Shared memory (e.g. Parallel RAM - Concurrent Read Exclusive Write)
- e.g. POSIX\textsuperscript{1} Threads

\textsuperscript{1}Portable Operating Systems Interface, X is a reference to Unix
**Paradigm**


- The Dijkstra’s language is a parallel extension of ALGOL60 with
  P (lock/take), V (unlock/release), and *parbegin* ... *parend*
- Shared memory (e.g. Parallel RAM - Concurrent Read Exclusive Write)
- e.g. POSIX¹ Threads
- Parallel compound can occur anywhere in a program e.g.

  \[
  x:=0 \ ; \ y:=0 \ ; \ (x:=1 \ || \ y:=1)
  \]

¹Portable Operating Systems Interface, X is a reference to Unix
Paradigm


- The Dijkstra’s language is a parallel extension of ALGOL60 with `P` (lock/take), `V` (unlock/release), and `parbegin ... parend`
- Shared memory (e.g. Parallel RAM - Concurrent Read Exclusive Write)
- e.g. POSIX¹ Threads
- Parallel compound can occur anywhere in a program e.g.

\[
x:=0 ; y:=0 ; (x:=1 \parallel y:=1)
\]

- The Carson and Reynolds language is a restriction of Dijkstra’s language:

¹Portable Operating Systems Interface, X is a reference to Unix
Paradigm


- The Dijkstra’s language is a parallel extension of ALGOL60 with
  \( P \) (lock/take), \( V \) (unlock/release), and \texttt{parbegin} ... \texttt{parend}
- Shared memory (e.g. Parallel RAM - Concurrent Read Exclusive Write)
- e.g. POSIX\(^1\) Threads
- Parallel compound can occur anywhere in a program e.g.

\[
x := 0 \; ; \; y := 0 \; ; (x := 1 \; || \; y := 1)
\]

- The Carson and Reynolds language is a restriction of Dijkstra’s language:
  - Operator \( || \) in outermost position: only sequential processes are executed in parallel

\(^1\)Portable Operating Systems Interface, X is a reference to Unix
Parallel Automata Meta Language
Syntax

Paradigm


- The Dijkstra’s language is a parallel extension of ALGOL60 with _P_ (lock/take), _V_ (unlock/release), and _parbegin_ ... _parend_
- Shared memory (e.g. Parallel RAM - Concurrent Read Exclusive Write)
- e.g. POSIX\(^1\) Threads
- Parallel compound can occur anywhere in a program e.g.

\[
x:=0 \; ; \; y:=0 \; ; (x:=1 \; || \; y:=1)
\]

- The Carson and Reynolds language is a restriction of Dijkstra’s language:
  - Operator _||_ in outermost position: only sequential processes are executed in parallel
  - Neither branchings nor loops

---

\(^1\)Portable Operating Systems Interface, X is a reference to Unix
Features

- shared memory abstract machine (PRAM)
- concurrent read exclusive write (CREW)
- Operator `||` in outermost position: only sequential processes are executed in parallel
- Branchings, loops, and synchronisation barriers `W` (wait) are allowed
- no pointer arithmetics
- no function call, only jumps
- no birth nor death of process at runtime
- tokens are owned by processes
- conservative processes
Features

- shared memory abstract machine (PRAM)
- concurrent read exclusive write (CREW)
Features

- shared memory abstract machine (PRAM)
  concurrent read exclusive write (CREW)
- Operator $||$ in outermost position: only sequential processes are executed in parallel
Features

- shared memory abstract machine (PRAM) concurrent read exclusive write (CREW)
- Operator $||$ in outermost position: only sequential processes are executed in parallel
- Branchings, loops, and synchronisation barriers $W$ (wait) are allowed
Features

- shared memory abstract machine (PRAM)
  concurrent read exclusive write (CREW)
- Operator || in outermost position: only sequential processes are executed in parallel
- Branchings, loops, and synchronisation barriers $W$ (wait) are allowed
- no pointer arithmetics
Features

- shared memory abstract machine (PRAM)
  concurrent read exclusive write (CREW)
- Operator | | in outermost position: only sequential processes are executed in parallel
- Branchings, loops, and synchronisation barriers W (wait) are allowed
- no pointer arithmetics
- no function call, only jumps
Features

- shared memory abstract machine (PRAM)
  concurrent read exclusive write (CREW)
- Operator || in outermost position: only sequential processes are executed in parallel
- Branchings, loops, and synchronisation barriers W (wait) are allowed
- no pointer arithmetics
- no function call, only jumps
- no birth nor death of process at runtime
Features

- shared memory abstract machine (PRAM) concurrent read exclusive write (CREW)
- Operator $\parallel$ in outermost position: only sequential processes are executed in parallel
- Branchings, loops, and synchronisation barriers $W$ (wait) are allowed
- no pointer arithmetics
- no function call, only jumps
- no birth nor death of process at runtime
- tokens are owned by processes
Features

- shared memory abstract machine (PRAM)
  concurrent read exclusive write (CREW)
- Operator || in outermost position: only sequential processes are executed in parallel
- Branchings, loops, and synchronisation barriers W (wait) are allowed
- no pointer arithmetics
- no function call, only jumps
- no birth nor death of process at runtime
- tokens are owned by processes
- conservative processes
Declarations

A basic block is defined as a (finite) sequence of instructions. A program is a list of declarations, the available declarations are:

- `sem <int> <set of identifiers>`
  e.g. `sem 3 a b c d`

- `sync <int> <set of identifiers>`
  e.g. `sync 3 a b c d`

- `mtx <set of identifiers>`
  e.g. `mtx a b c d`

- `var <identifier> = <constant>`
  e.g. `var x = 0`

- `proc <identifier> = <basic block>`

- `init <multiset of identifiers>`
  e.g. `init a 2b 3c`
Declarations

A basic block is defined as a (finite) sequence of instructions. A program is a list of declarations, the available declarations are:

- `sem <int> <set of identifiers>`
  
  e.g. `sem 3 a b c d`

- `sync <int> <set of identifiers>`
  
  e.g. `sync 3 a b c d`

- `mtx <set of identifiers>`
  
  e.g. `mtx a b c d`

- `var <identifier> = <constant>`
  
  e.g. `var x = 0`

- `proc <identifier> = <basic block>`

- `init <multiset of identifiers>`
  
  e.g. `init a 2b 3c`
A basic block is defined as a (finite) sequence of instructions. A program is a list of declarations, the available declarations are:

- sem <int> <set of identifiers>
  e.g. sem 3 a b c d

- sync <int> <set of identifiers>

- mtx <set of identifiers>

- var <identifier> = <constant>
  e.g. var x = 0

- proc <identifier> = <basic block>

- init <multiset of identifiers>
  e.g. init a 2b 3c
Declarations

A basic block is defined as a (finite) sequence of instructions. A program is a list of declarations, the available declarations are:

- **sem** <int> <set of identifiers>
  
e.g. `sem 3 a b c d`

- **sync** <int> <set of identifiers>
  
e.g. `sync 3 a b c d`
A basic block is defined as a (finite) sequence of instructions. A program is a list of declarations, the available declarations are:

- **sem <int> <set of identifiers>**  
  e.g. `sem 3 a b c d`

- **sync <int> <set of identifiers>**  
  e.g. `sync 3 a b c d`

- **mtx <set of identifiers>**  
  e.g. `mtx a b c d`

- **var <identifier> = <constant>**  
  e.g. `var x = 0`

- **proc <identifier> = <basic block>**

- **init <multiset of identifiers>**  
  e.g. `init a 2b 3c`
Declarations

A basic block is defined as a (finite) sequence of instructions. A program is a list of declarations, the available declarations are:

- **sem** `<int> <set of identifiers>`
  
  e.g. `sem 3 a b c d`

- **sync** `<int> <set of identifiers>`
  
  e.g. `sync 3 a b c d`

- **mtx** `<set of identifiers>`
  
  e.g. `mtx a b c d`

- **var** `<identifier> = <constant>`
  
  e.g. `var x = 0`
A basic block is defined as a (finite) sequence of instructions. A program is a list of declarations, the available declarations are:

- **sem** `<int> <set of identifiers>`
  
e.g. `sem 3 a b c d`

- **sync** `<int> <set of identifiers>`
  
e.g. `sync 3 a b c d`

- **mtx** `<set of identifiers>`
  
e.g. `mtx a b c d`

- **var** `<identifier> = <constant>`
  
e.g. `var x = 0`

- **proc** `<identifier> = <basic block>`
A basic block is defined as a (finite) sequence of instructions. A program is a list of declarations, the available declarations are:

- `sem <int> <set of identifiers>`
  e.g. `sem 3 a b c d`
- `sync <int> <set of identifiers>`
  e.g. `sync 3 a b c d`
- `mtx <set of identifiers>`
  e.g. `mtx a b c d`
- `var <identifier> = <constant>`
  e.g. `var x = 0`
- `proc <identifier> = <basic block>`
- `init <multiset of identifiers>`
  e.g. `init a 2b 3c`
Expressions and values

The set of expressions is inductively built on the set of identifiers and the following set of operators:

- $\land$: content of
- $\lor$: constant
- $\land$: minimum
- $\lor$: maximum
- $+$: addition
- $-$: substraction
- $\times$: multiplication
- $/$: division
- $\leq$: less or equal
- $\geq$: greater or equal
- $<$: strictly less
- $>$: strictly greater
- $=$: equal
- $\neq$: not equal
- $\neg$: complement
- $\%$: modulo
- $\bot$: bottom

Note: The operators $\land$, $\lor$, $\times$, $/$, $<$, $>$, $\leq$, $\geq$, $\neq$, $\%$ are nullary (binary) operators when $v \in V$, $x \in R$, respectively.
Expressions and values

The set of expressions is inductively built on the set of identifiers and the following set of operators
Expressions and values

The set of expressions is inductively built on the set of identifiers and the following set of operators:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Argument_1</th>
<th>Argument_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>content of v ∈ V</td>
<td>x ∈ R</td>
<td>constant</td>
</tr>
<tr>
<td>∧</td>
<td>minimum</td>
<td>v</td>
<td>maximum</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
<td>−</td>
<td>substraction</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td>≤</td>
<td>less or equal</td>
<td>≥</td>
<td>greater of equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>strictly less</td>
<td>&gt;</td>
<td>strictly greater</td>
</tr>
<tr>
<td>=</td>
<td>equal</td>
<td>≠</td>
<td>not equal</td>
</tr>
<tr>
<td>¬</td>
<td>complement</td>
<td>%</td>
<td>modulo</td>
</tr>
<tr>
<td>⊥</td>
<td>nullary</td>
<td>unary</td>
<td></td>
</tr>
<tr>
<td></td>
<td>¬</td>
<td>binary</td>
<td></td>
</tr>
<tr>
<td></td>
<td>∧, v, +, −, *, /, &lt;, &gt;, ≤, ≥, =, ≠, %</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Non branching instructions

- `identifier := expression`
  The expression is evaluated then the result is stored in the identifier.

- `P(identifier)`
  Takes an occurrence of the resource `identifier` (there are `arity` available tokens), stops the process otherwise.

- `V(identifier)`
  Release an occurrence of the resource `identifier` (if such an occurrence is held by the process), ignored otherwise.

- `W(identifier)`
  Stops the execution of the process until `arity + 1` of them are stopped by the barrier `identifier`.

- `J(identifier)`
  The execution of the process is stopped and the one of a copy of `identifier` starts. There is no return mechanism.

- `(L)`
  Enclose a list of instructions between parenthesis to make it a single instruction.
Non branching instructions

- identifier := expression  the expression is evaluated then the result is stored in the identifier
Non branching instructions

- **identifier**:=**expression** the expression is evaluated then the result is stored in the identifier
- **P(identifier)** takes an occurrence of the resource **identifier** (there are **arity** available tokens), stops the process otherwise

- **L** enclose a list of instructions between parenthesis to make it a single instruction
Non branching instructions

- **identifier**:=**expression** the expression is evaluated then the result is stored in the identifier
- **P(identifier)** takes an occurrence of the resource **identifier** (there are **arity** available tokens), stops the process otherwise
- **V(identifier)** release an occurrence of the resource **identifier** (if such an occurrence is held by the process), ignored otherwise
Non branching instructions

- \texttt{identifier} := \texttt{expression} the expression is evaluated then the result is stored in the identifier

- \texttt{P(identifier)} takes an occurrence of the resource \texttt{identifier} (there are \textit{arity} available tokens), stops the process otherwise

- \texttt{V(identifier)} release an occurrence of the resource \texttt{identifier} (if such an occurrence is held by the process), ignored otherwise

- \texttt{W(identifier)} stops the execution of the process until \textit{arity} + 1 of them are stopped by the barrier \texttt{identifier}
Non branching instructions

- **identifier**:=**expression** the expression is evaluated then the result is stored in the identifier
- **P(identifier)** takes an occurrence of the resource **identifier** (there are **arity** available tokens), stops the process otherwise
- **V(identifier)** release an occurrence of the resource **identifier** (if such an occurrence is held by the process), ignored otherwise
- **W(identifier)** stops the execution of the process until **arity** + 1 of them are stopped by the barrier **identifier**
- **J(identifier)** the execution of the process is stopped and the one of a copy of **identifier** starts. There is no return mechanism.
Non branching instructions

- \text{identifier}: = \text{expression} \text{ the expression is evaluated then the result is stored in the identifier}
- \text{P(identifier)} \text{ takes an occurrence of the resource } \text{identifier} \text{ (there are } \text{arity} \text{ available tokens), stops the process otherwise}
- \text{V(identifier)} \text{ release an occurrence of the resource } \text{identifier} \text{ (if such an occurrence is held by the process), ignored otherwise}
- \text{W(identifier)} \text{ stops the execution of the process until } \text{arity} + 1 \text{ of them are stopped by the barrier } \text{identifier}
- \text{J(identifier)} \text{ the execution of the process is stopped and the one of a copy of } \text{identifier} \text{ starts. There is no return mechanism.}
- (L) \text{ enclose a list of instructions between parenthesis to make it a single instruction}
Branching

The branching is provided by a kind of "match case like" instruction:

\[
(L_1)^+ [e_1]^+ (L_2)^+ [e_2]^+ \cdots (L_n)^+ [e_n]^+(L_{n+1})
\]

- Each \(L_k\) is a basic block
- Each \(e_k\) is an expression
- The triggered branch is \(L_k\) with \(k\) being the first index such that \(e_k\) evaluates to some nonzero value
- If all the expressions evaluate to zero, then \(L_{n+1}\) is triggered.
The branching is provided by a kind of “match case like” instruction

\[(L_1) + [e_1] + (L_2) + [e_2] + \cdots + (L_n) + [e_n] + (L_{n+1})\]
Branching

The branching is provided by a kind of “match case like” instruction

\[(L_1)+[e_1]+(L_2)+[e_2]+\cdots+(L_n)+[e_n]+(L_{n+1})\]

- Each $L_k$ is a basic block
Branching

The branching is provided by a kind of “match case like” instruction

\[(L_1) + [e_1] + (L_2) + [e_2] + \cdots + (L_n) + [e_n] + (L_{n+1})\]

- Each \( L_k \) is a basic block
- Each \( e_k \) is an expression
Branching

The branching is provided by a kind of “match case like” instruction

\[(L_1)[e_1] + (L_2)[e_2] + \cdots + (L_n)[e_n] + (L_{n+1})\]

- Each \(L_k\) is a basic block
- Each \(e_k\) is an expression
- The triggered branch is \(L_k\) with \(k\) being the first index such that \(e_k\) evaluate to some nonzero value
The branching is provided by a kind of “match case like” instruction

\[(L_1) + [e_1] + (L_2) + [e_2] + \cdots + (L_n) + [e_n] + (L_{n+1})\]

- Each \(L_k\) is a basic block
- Each \(e_k\) is an expression
- The triggered branch is \(L_k\) with \(k\) being the first index such that \(e_k\) evaluate to some nonzero value
- If all the expressions evaluate to zero, then \(L_{n+1}\) is triggered.
Describing a process
Describing a process

The body of a process is just a (possibly empty) sequence of instructions, i.e. a basic block, separated by semicolons. E.g. the Hasse/Syracuse algorithm with input value 7:
Describing a process

The body of a process is just a (possibly empty) sequence of instructions, i.e. a basic block, separated by semicolons e.g. the Hasse/Syracuse algorithm with input value 7

\`\text{proc}\ \text{p} = x:=7; J(q)\`
Describing a process

The body of a process is just a (possibly empty) sequence of instructions, i.e. a basic block, separated by semicolons e.g. the Hasse/Syracuse algorithm with input value 7

```plaintext
proc p = x:=7; J(q)

proc q = J(r) + [x<>1] + ()
```

Describing a process

The body of a process is just a (possibly empty) sequence of instructions, i.e. a basic block, separated by semicolons e.g. the Hasse/Syracuse algorithm with input value 7

```plaintext
proc p = x:=7; J(q)

proc q = J(r)+[x<>1]+() 

proc r = (x:=x/2)+[x%2=0]+(x:=3*x+1) ; J(q)
```
Parallel Automata Meta Language

Describing a process

The body of a process is just a (possibly empty) sequence of instructions, i.e. a basic block, separated by semicolons.

E.g. the Hasse/Syracuse algorithm with input value 7

```
proc p = x:=7;J(q)
proc q = J(r)+[x<>1]+() 
proc r = (x:=x/2)+[x%2=0]+(x:=3*x+1) ; J(q)
init p
```
Describing a process

The body of a process is just a (possibly empty) sequence of instructions, i.e. a basic block, separated by semicolons e.g. the Hasse/Syracuse algorithm with input value 7

```plaintext
proc p = x:=7; J(q)
proc q = J(r)+[x<>1]+()
proc r = (x:=x/2)+[x%2=0]+(x:=3*x+1) ; J(q)
init p
```

Due to the branchings, basic blocks are actually trees.
Control Flow Graphs
Control flow graphs and flowcharts

Control flow analysis, *F. E. Allen*, 1970
Assigning meanings to programs, *R. W. Floyd*, 1967
Control flow graphs and flowcharts

Control flow analysis, *F. E. Allen*, 1970
Assigning meanings to programs, *R. W. Floyd*, 1967

- Compilers and static analyzers internal representation of programs.
Control flow graphs and flowcharts

Control flow analysis, *F. E. Allen*, 1970
Assigning meanings to programs, *R. W. Floyd*, 1967

- Compilers and static analyzers internal representation of programs.
- No theoretical definition yet control flow graphs must be finite for practical reasons.
Control flow graphs and flowcharts

Control flow analysis, *F. E. Allen*, 1970
Assigning meanings to programs, *R. W. Floyd*, 1967

- **Compilers** and static analyzers internal representation of programs.
- No theoretical definition yet control flow graphs must be finite for practical reasons.
- At the core of many softwares dealing with source code
e.g. GCC (cf. “basic blocks”), LLVM, Frama-C.
Control flow graphs and flowcharts

Control flow analysis, *F. E. Allen*, 1970
Assigning meanings to programs, *R. W. Floyd*, 1967

- **Compilers and static analyzers** internal representation of programs.
- No theoretical definition yet **control flow graphs must be finite** for practical reasons.
- At the core of many softwares dealing with source code e.g. GCC (cf. “basic blocks”), LLVM, Frama-C.
- No such structure exist for parallel programs.
Generators

\[
x := f \\
\]

\[
\begin{align*}
&x := f \\
&b_1 \\
&a_1 \\
&b_1 \\
&\phi? \\
&\text{Yes} \\
&b_1 \\
&\text{No} \\
&b_2
\end{align*}
\]

\[
\begin{align*}
&a_1 \\
&b_1 \\
&a_2 \\
&b_1
\end{align*}
\]
The Hasse-Syracuse algorithm in PAML

\[
\begin{align*}
\text{var } x &= 7 \\
\text{proc } p &= () + [x=1] + J(q) \\
\text{proc } q &= (x:=x/2) + [x\%2=0] + (x:=3x+1) ; J(p) \\
\text{init } p
\end{align*}
\]
Building the control flow graph
of the Hasse-Syracuse algorithm
Building the control flow graph
of the Hasse-Syracuse algorithm
Building the control flow graph of the Hasse-Syracuse algorithm

entry point of the basic block of $p$

entry point of the basic block of $q$

$x = 1$

$x := 3 * x + 1$

$x = x / 2$

$x \% 2 = 0$
Building the control flow graph
of the Hasse-Syracuse algorithm
Building the control flow graph of the Hasse-Syracuse algorithm
Building the control flow graph
of the Hasse-Syracuse algorithm

\[ x = 1 \]

entry point of the
basic block of \( p \)

entry point of the
basic block of \( q \)

\[ x := 3 \times x + 1 \]
Building the control flow graph
of the Hasse-Syracuse algorithm
Reducing the Control Flow Graph of the Hasse-Syracuse algorithm

\[ x := x/2 \]

\[ x := 3 \times x + 1 \]

\[ x \mod 2 = 0 \]

\[ x = 1 \]

The current value of \( x \) is 7.
Reducing the Control Flow Graph
of the Hasse-Syracuse algorithm
Reducing the Control Flow Graph
of the Hasse-Syracuse algorithm

entry point

x := x / 2

x := 3 * x + 1

x = 1

x = 1

x % 2 = 0

x % 2 = 0

the current value of x is 7

12 / 49
An execution trace on a control flow graph
of the Hasse-Syracuse algorithm
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of $x$ is 7
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of $x$ is 7
An execution trace on a control flow graph
of the Hasse-Syracuse algorithm

the current value of $x$ is 22
An execution trace on a control flow graph of the Hasse-Syracuse algorithm.

The current value of $x$ is 22.
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

\[
\begin{align*}
x &= x/2 \\
x &= 3 \cdot x + 1 \\
x \% 2 &= 0 \\
x &= 1
\end{align*}
\]

the current value of \( x \) is 22
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

The current value of $x$ is 22
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

The current value of $x$ is 11
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

The current value of $x$ is 11
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

The current value of $x$ is 11
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of x is 11
An execution trace on a control flow graph
of the Hasse-Syracuse algorithm

the current value of \( x \) is 34
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

The current value of $x$ is 34
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of $x$ is 34
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

The current value of $x$ is 34
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of $x$ is 17
An execution trace on a control flow graph
of the Hasse-Syracuse algorithm

the current value of $x$ is 17
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of $x$ is 17
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

The current value of $x$ is 17
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of $x$ is 52
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

The current value of $x$ is 52
An execution trace on a control flow graph
of the Hasse-Syracuse algorithm

the current value of $x$ is 52
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

The current value of \( x \) is 52
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

The current value of x is 26
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of \( x \) is 26
An execution trace on a control flow graph
of the Hasse-Syracuse algorithm

x := x/2
x := 3*x + 1
x mod 2 = 0
x = 1

the current value of x is 26
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of $x$ is 26
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of $x$ is 13
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

The current value of $x$ is 13
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of x is 13
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of \( x \) is 13
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of $x$ is 40
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of $x$ is 40
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of $x$ is 40
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

The current value of $x$ is 40
An execution trace on a control flow graph

of the Hasse-Syracuse algorithm

the current value of $x$ is 20
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of \( x \) is 20
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of $x$ is 20
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

The current value of $x$ is 20
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

The current value of x is 10
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

The current value of $x$ is 10
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of $x$ is 10
An execution trace on a control flow graph
of the Hasse-Syracuse algorithm

the current value of $x$ is 10
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of $x$ is 5
An execution trace on a control flow graph
of the Hasse-Syracuse algorithm

the current value of $x$ is 5
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of \( x \) is 5
An execution trace on a control flow graph
of the Hasse-Syracuse algorithm

The current value of $x$ is 5
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

The current value of $x$ is 16
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

The current value of $x$ is 16
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

The current value of $x$ is 16

entry point

$x = 1$

$x = x/2$

$x = 3 \times x + 1$

$x \mod 2 = 0$
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

The current value of $x$ is 16
An execution trace on a control flow graph

of the Hasse-Syracuse algorithm

the current value of $x$ is 8
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of $x$ is 8
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of $x$ is 8
An execution trace on a control flow graph of the Hasse-Syracuse algorithm.

The current value of $x$ is 8
An execution trace on a control flow graph
of the Hasse-Syracuse algorithm

the current value of $x$ is 4
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

- The current value of $x$ is 7
- The current value of $x$ is 4

The current value of $x$ is 4
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

The current value of $x$ is 4
An execution trace on a control flow graph
of the Hasse-Syracuse algorithm

the current value of $x$ is 4
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

- **entry point**
- $x := x/2$
- $x := 3 \times x + 1$
- $x \% 2 = 0$
- $x = 1$

The current value of $x$ is 2.
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of $x$ is 2
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of \( x \) is 2
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of \( x \) is 2
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of $x$ is 1
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of \( x \) is 1
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of $x$ is 1
An execution trace on a control flow graph
of the Hasse-Syracuse algorithm

the current value of $x$ is 1
An execution trace on a control flow graph of the Hasse-Syracuse algorithm

the current value of $x$ is 1
Execution traces as paths over a control flow graph
Execution traces as paths over a control flow graph

- Any execution trace induces a path
Execution traces as paths over a control flow graph

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

- Any execution trace induces a path
- Some paths do not come from an execution trace
- Therefore the collection of path provides a (strict) overapproximation of the collection of execution traces
- Any execution trace induces a path
- Some paths do not come from an execution trace
- Therefore the collection of path 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 overall idea of static analysis

Any model of a program should contain a finite representation of an overapproximation of the collection of all its execution traces.
The overall idea of static analysis

Any model of a program should contain a finite representation of an overapproximation of the collection of all its execution traces.

One of the goal of the course it to provide such a structure for a large class of PAML programs.
Restrictions from the PAML syntax

- There is neither birth nor death of processes at runtime
- The arity of resources cannot be changed at runtime
- There is no pointer arithmetics
Restrictions from the PAML syntax

By construction the PAML language enforces the following restrictions
Restrictions from the PAML syntax

By construction the PAML language enforces the following restrictions
- There is neither birth nor death of processes at runtime
Restrictions from the PAML syntax

By construction the PAML language enforces the following restrictions

- There is neither birth nor death of processes at runtime
- The arity of resources cannot by changed at runtime
By construction the PAML language enforces the following restrictions
- There is neither birth nor death of processes at runtime
- The arity of resources cannot by changed at runtime
- There is no pointer arithmetics
Abstract Machine
Abstract expressions
Abstract expressions

- The set of variables of a program is $\mathcal{X}$. 
Abstract expressions

- The set of variables of a program is $\mathcal{X}$.
- A valuation or memory state is a mapping $\nu : \mathcal{X} \rightarrow \mathbb{R}_{\bot} = \mathbb{R} \cup \{\bot\}$. 
Abstract expressions

- The set of variables of a program is $\mathcal{X}$.
- A valuation or memory state is a mapping $\nu : \mathcal{X} \rightarrow \mathbb{R}_\bot = \mathbb{R} \cup \{\bot\}$.
- An expression is a mapping $\varepsilon : \{\text{valuations}\} \rightarrow \mathbb{R}$ with a finite set $\mathcal{F}(\varepsilon) \subseteq \mathcal{X}$ such that if the valuations $\nu$ and $\nu'$ match on $\mathcal{F}(\varepsilon)$ then $\varepsilon(\nu) = \varepsilon(\nu')$. 
Abstract expressions

- The set of variables of a program is $\mathcal{X}$.
- A valuation or memory state is a mapping $\nu : \mathcal{X} \rightarrow \mathbb{R}_\bot = \mathbb{R} \cup \{\bot\}$.
- An expression is a mapping $\varepsilon : \{\text{valuations}\} \rightarrow \mathbb{R}$ with a finite set $\mathcal{F}(\varepsilon) \subseteq \mathcal{X}$ such that if the valuations $\nu$ and $\nu'$ match on $\mathcal{F}(\varepsilon)$ then $\varepsilon(\nu) = \varepsilon(\nu')$.
- The set of expressions occurring in the program is denoted by $\mathcal{E}$. 
Interpretation of expressions

only depends on the current memory state
Interpretation of expressions
only depends on the current memory state

- $[x]_\nu = \nu(x)$ for all $x \in \mathcal{X}$
Interpretation of expressions
only depends on the current memory state

- $[x]_\nu = \nu(x)$ for all $x \in \mathcal{X}$
- Any value in $\mathbb{R} \setminus \{0\}$ stands for true while 0 stands for false
Interpretation of expressions
only depends on the current memory state

- \([x]_\nu = \nu(x)\) for all \(x \in \mathcal{X}\)
- Any value in \(\mathbb{R} \setminus \{0\}\) stands for true while 0 stands for false
- \([-] : \mathbb{R}_\perp \to \mathbb{R}_\perp\)
Parallel Automata Meta Language

Interpretation of expressions
only depends on the current memory state

- $[x]_\nu = \nu(x)$ for all $x \in X$
- Any value in $\mathbb{R} \setminus \{0\}$ stands for true while 0 stands for false
- $[\neg] : \mathbb{R}_\bot \to \mathbb{R}_\bot$
  $\neg(0) = 1$
Interpretation of expressions
only depends on the current memory state

- \([x]_\nu = \nu(x)\) for all \(x \in X\)
- Any value in \(\mathbb{R} \setminus \{0\}\) stands for \text{true} while \(0\) stands for \text{false}
- \([\neg] : \mathbb{R}_\bot \to \mathbb{R}_\bot\),
  \([\neg](0) = 1\),
  \([\neg](\bot) = \bot\), and
Interpretation of expressions
only depends on the current memory state

- $\llbracket x \rrbracket_\nu = \nu(x)$ for all $x \in \mathcal{X}$
- Any value in $\mathbb{R} \setminus \{0\}$ stands for true while 0 stands for false
- $\llbracket \neg \rrbracket : \mathbb{R}_\bot \rightarrow \mathbb{R}_\bot$, $\llbracket \neg \rrbracket(0) = 1$, $\llbracket \neg \rrbracket(\bot) = \bot$, and $\llbracket \neg \rrbracket(x) = 0$ for all $x \in \mathbb{R} \setminus \{0\}$
Interpretation of expressions
only depends on the current memory state

- $\llbracket x \rrbracket_\nu = \nu(x)$ for all $x \in \mathcal{X}$
- Any value in $\mathbb{R} \setminus \{0\}$ stands for true while 0 stands for false
- $\llbracket \neg \rrbracket : \mathbb{R}_\perp \rightarrow \mathbb{R}_\perp$,
  $\llbracket \neg \rrbracket (0) = 1$,
  $\llbracket \neg \rrbracket (\perp) = \perp$, and
  $\llbracket \neg \rrbracket (x) = 0$ for all $x \in \mathbb{R} \setminus \{0\}$
- $\llbracket e \rrbracket = \perp$ for all expression $e$ in which $\perp$ occurs
Interpretation of expressions
only depends on the current memory state

- $[[x]]_\nu = \nu(x)$ for all $x \in X$
- Any value in $\mathbb{R} \setminus \{0\}$ stands for true while 0 stands for false
- $[[\neg]] : \mathbb{R}_\bot \to \mathbb{R}_\bot$, 
  $[[\neg]](0) = 1$, 
  $[[\neg]](\bot) = \bot$, and 
  $[[\neg]](x) = 0$ for all $x \in \mathbb{R} \setminus \{0\}$
- $[[e]] = \bot$ for all expression $e$ in which $\bot$ occurs
- the other operators are interpreted as expected
Abstract instructions
Abstract instructions

The sets of semaphores, and barriers of a program are respectively $S$ and $B$. 
Abstract instructions

The sets of semaphores, and barriers of a program are respectively $S$ and $B$.

- An assignment is an element of $\mathcal{X} \times \mathcal{E}$ yet we write $x := \epsilon$ instead of $(x, \epsilon)$. By extension $\mathcal{F}(x := \epsilon) = \mathcal{F}(\epsilon)$.
Abstract instructions

The sets of semaphores, and barriers of a program are respectively \( S \) and \( B \).
- An assignment is an element of \( X \times \mathcal{E} \) yet we write \( x := \varepsilon \) instead of \((x, \varepsilon)\). By extension \( \mathcal{F}(x := \varepsilon) = \mathcal{F}(\varepsilon) \).
- Given a graph

\[
G : A \xrightarrow{\partial^-} V \xleftarrow{\partial^+}
\]

a conditional branching at vertex \( v \in V \) is a mapping

\[
\beta : \{\text{valuations}\} \rightarrow \{ a \in A \mid \partial^- a = v \}
\]

together with a subset \( \mathcal{F}(\beta) \subseteq X \) such that if the valuations \( \nu \) and \( \nu' \) match on \( \mathcal{F}(\beta) \) then \( \beta(\nu) = \beta(\nu') \).
Abstract instructions

The sets of semaphores, and barriers of a program are respectively $S$ and $B$.

- An assignment is an element of $X \times E$ yet we write $x := \varepsilon$ instead of $(x, \varepsilon)$. By extension $F(x := \varepsilon) = F(\varepsilon)$.

- Given a graph

\[
G : A \xrightarrow{\partial^-} V \quad \xrightarrow{\partial^+} V
\]

A conditional branching at vertex $v \in V$ is a mapping

\[
\beta : \{\text{valuations}\} \rightarrow \{a \in A \mid \partial^- a = v\}
\]

together with a subset $F(\beta) \subseteq X$ such that if the valuations $\nu$ and $\nu'$ match on $F(\beta)$ then $\beta(\nu) = \beta(\nu')$.

- The synchronisation primitives $P(s)$, $V(s)$, and $W(b)$ for $s \in S$ and $b \in B$
Abstract processes as control flow graphs
Abstract processes as control flow graphs

\[ G : A \xrightarrow{\partial} V \quad \text{and} \quad \lambda : V \rightarrow \{\text{instructions}\} \]
Abstract processes as control flow graphs

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

- An entry point \( v_0 \in V \) such that \( \lambda(v_0) = \text{Skip} \).
Abstract processes as control flow graphs

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

- An entry point \( v_0 \in V \) such that \( \lambda(v_0) = \text{Skip} \).
- If \( \lambda(v) \neq \text{Skip} \), then \( v \) has at least one outgoing arrow.
Abstract processes as control flow graphs

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

- An entry point \( v_0 \in V \) such that \( \lambda(v_0) = \text{Skip} \).
- If \( \lambda(v) \neq \text{Skip} \), then \( v \) has at least one outgoing arrow.
- If \( \lambda(v) \) is not a branching, then \( v \) has at most one outgoing arrow.
Abstract processes as control flow graphs

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

- An entry point \( v_0 \in V \) such that \( \lambda(v_0) = \text{Skip} \).
- If \( \lambda(v) \neq \text{Skip} \), then \( v \) has at least one outgoing arrow.
- If \( \lambda(v) \) is not a branching, then \( v \) has at most one outgoing arrow.

The arrows are interpreted as **intermediate positions** of the instruction pointer so a **point** on a control flow graph is either a vertex or an arrow.
Abstract program
Abstract program

- The initial valuation $\nu : \mathcal{X} \rightarrow \mathbb{R}$ which provides the values of the variables at the beginning of each execution of the program.
Abstract program

- The initial valuation $\nu : X \rightarrow \mathbb{R}$ which provides the values of the variables at the beginning of each execution of the program.
- The arity map $\alpha : S \sqcup B \rightarrow \mathbb{N} \cup \{\infty\}$.
Abstract program

- The initial valuation $\nu : \mathcal{X} \to \mathbb{R}$ which provides the values of the variables at the beginning of each execution of the program.

- The arity map $\alpha : \mathcal{S} \sqcup \mathcal{B} \to \mathbb{N} \cup \{\infty\}$.

- The tuple $(G_1, \ldots, G_n)$ of processes which are launched simultaneously at the beginning of each execution of the program.
Points and multi-instructions

*Higher Dimensional Transition Systems*, G. L. Cattani and V. Sassone, 1996
Points and multi-instructions

*Higher Dimensional Transition Systems*, G. L. Cattani and V. Sassone, 1996

- A point of \((G_1, \ldots, G_n)\) is an \(n\)-tuple \(p\) whose \(i^{th}\) component, namely \(p_i\), is a point of \(G_i\).
Points and multi-instructions

*Higher Dimensional Transition Systems*, G. L. Cattani and V. Sassone, 1996

- A point of \((G_1, \ldots, G_n)\) is an \(n\)-tuple \(p\) whose \(i^{th}\) component, namely \(p_i\), is a point of \(G_i\).
- A multi-instruction is a partial map \(\mu : \{1, \ldots, n\} \to \{\text{instructions}\} \).
The internal states of the abstract machine

A state is a mapping \( \sigma \) defined over the disjoint union \( X \sqcup S \) such that:

- for all \( x \in X \), \( \sigma(x) \in R \perp \), and
- for all \( s \in S \), \( \sigma(s) \) is a multiset over \( \{1, \ldots, n\} \).
The internal states of the abstract machine

A state is a mapping \( \sigma \) defined over the disjoint union \( \mathcal{X} \sqcup \mathcal{S} \) such that:

- for all \( x \in \mathcal{X} \), \( \sigma(x) \in R \), and
- for all \( s \in \mathcal{S} \), \( \sigma(s) \) is a multiset over \( \{1, \ldots, n\} \).
The internal states of the abstract machine

A state is a mapping $\sigma$ defined over the disjoint union $\mathcal{X} \sqcup \mathcal{S}$ such that:
- for all $x \in \mathcal{X}$, $\sigma(x) \in \mathbb{R}_{\perp}$, and
The internal states of the abstract machine

A state is a mapping $\sigma$ defined over the disjoint union $X \sqcup S$ such that:
- for all $x \in X$, $\sigma(x) \in \mathbb{R}_\perp$, and
- for all $s \in S$, $\sigma(s)$ is a multiset over $\{1, \ldots, n\}$.
Admissible multi-instructions

- write-write: $x := \varepsilon$ vs $x := \varepsilon'$
- read-write: $x := \varepsilon$ vs an instruction in which $x$ is free

A multi-instruction $\mu$ is said to be admissible at state $\sigma$ when:

- for $i, j \in \text{dom}(\mu)$ with $i \neq j$, $\mu(i)$ and $\mu(j)$ do not conflict,
- for all $s \in S$, $0 \leq \varphi(s) \leq \alpha(s)$ where $\varphi(s) = |\sigma(s)| + \text{card}\{i \in \text{dom}(\mu) | \mu(i) = P(s)\} - \text{card}\{i \in \text{dom}(\mu) | \mu(i) = V(s)\}$,
- for all $b \in B$, $\text{card}\{i \in \text{dom}(\mu) | \mu(i) = W(b)\} \notin \{1, \ldots, \alpha(b)\}$.
Admissible multi-instructions

The possible conflicts are:
Admissible multi-instructions

The possible conflicts are:

- write-write: \(x := \varepsilon\) vs \(x := \varepsilon'\)
Admissible multi-instructions

The possible conflicts are:

- write-write : $x := \epsilon$ vs $x := \epsilon'$
- read-write : $x := \epsilon$ vs an instruction in which $x$ is free
Admissible multi-instructions

The possible conflicts are:
- write-write: \( x := \varepsilon \) vs \( x := \varepsilon' \)
- read-write: \( x := \varepsilon \) vs an instruction in which \( x \) is free

A multi-instruction \( \mu \) is said to be admissible at state \( \sigma \) when:
Admissible multi-instructions

The possible conflicts are:

- write-write: \( x := \varepsilon \) vs \( x := \varepsilon' \)
- read-write: \( x := \varepsilon \) vs an instruction in which \( x \) is free

A multi-instruction \( \mu \) is said to be admissible at state \( \sigma \) when:

- for \( i, j \in \text{dom}(\mu) \) with \( i \neq j \), \( \mu(i) \) and \( \mu(j) \) do not conflict,
Admissible multi-instructions

The possible conflicts are:
- write-write: \( x := \varepsilon \) vs \( x := \varepsilon' \)
- read-write: \( x := \varepsilon \) vs an instruction in which \( x \) is free

A multi-instruction \( \mu \) is said to be admissible at state \( \sigma \) when:
- for \( i, j \in \text{dom}(\mu) \) with \( i \neq j \), \( \mu(i) \) and \( \mu(j) \) do not conflict,
- for all \( s \in S \), \( 0 \leq \phi(s) \leq \alpha(s) \) where

\[
\phi(s) = |\sigma(s)| + \text{card}\{i \in \text{dom}(\mu) \mid \mu(i) = P(s)\} - \text{card}\{i \in \text{dom}(\mu) \mid \mu(i) = V(s)\}
\]
Admissible multi-instructions

The possible conflicts are:
- write-write: $x := \varepsilon$ vs $x := \varepsilon'$
- read-write: $x := \varepsilon$ vs an instruction in which $x$ is free

A multi-instruction $\mu$ is said to be admissible at state $\sigma$ when:
- for $i, j \in \text{dom}(\mu)$ with $i \neq j$, $\mu(i)$ and $\mu(j)$ do not conflict,
- for all $s \in S$, $0 \leq \phi(s) \leq \alpha(s)$ where

$$
\phi(s) = |\sigma(s)| + \text{card}\{i \in \text{dom}(\mu) \mid \mu(i) = P(s)\} - \text{card}\{i \in \text{dom}(\mu) \mid \mu(i) = V(s)\}
$$

- for all $b \in B$, $\text{card}\{i \in \text{dom}(\mu) \mid \mu(i) = W(b)\} \not\subseteq \{1, \ldots, \alpha(b)\}$
Action of a multi-instruction on a state

Assuming that $\mu$ is admissible at $\sigma$
Action of a multi-instruction on a state
Assuming that $\mu$ is admissible at $\sigma$

The state $\sigma \cdot \mu$ is defined as follows.
Action of a multi-instruction on a state

Assuming that $\mu$ is admissible at $\sigma$

The state $\sigma \cdot \mu$ is defined as follows.

- For every $x \in X$, if there exists $i \in \{1, \ldots, n\}$ s.t. $\mu(i)$ is $x := \varepsilon$, then one has

$$
(\sigma \cdot \mu)(x) = \varepsilon(\sigma|X)
$$

- For all $s \in S$ the multiset $(\sigma \cdot \mu)(s)$, seen as a mapping from $\{1, \ldots, n\}$ to $\mathbb{N}$, is given by

$$
i \rightarrow
\begin{cases}
\sigma(s)(i) + 1 & \text{if } i \in \text{dom}(\mu) \text{ and } \mu(i) = P(s) \\
\sigma(s)(i) - 1 & \text{if } i \in \text{dom}(\mu) \text{ and } \mu(i) = V(s) \\
\sigma(s)(i) & \text{in all other cases}
\end{cases}
$$
Action of a multi-instruction on a state

Assuming that $\mu$ is admissible at $\sigma$

The state $\sigma \cdot \mu$ is defined as follows.
- For every $x \in \mathcal{X}$, if there exists $i \in \{1, \ldots, n\}$ s.t. $\mu(i)$ is $x := \varepsilon$, then one has

$$\sigma \cdot \mu(x) = \varepsilon(\sigma|\mathcal{X})$$

Otherwise one has $(\sigma \cdot \mu)(x) = \sigma(x)$. 
Action of a multi-instruction on a state

Assuming that $\mu$ is admissible at $\sigma$

The state $\sigma \cdot \mu$ is defined as follows.

- For every $x \in \mathcal{X}$, if there exists $i \in \{1, \ldots, n\}$ s.t. $\mu(i)$ is $x := \varepsilon$, then one has

$$ (\sigma \cdot \mu)(x) = \varepsilon(\sigma \mid \mathcal{X}) $$

Otherwise one has $(\sigma \cdot \mu)(x) = \sigma(x)$.

- For all $s \in S$ the multiset $(\sigma \cdot \mu)(s)$, seen as a mapping from $\{1, \ldots, n\}$ to $\mathbb{N}$, is given by

$$ i \mapsto \begin{cases} 
\sigma(s)(i) + 1 & \text{if } i \in \text{dom}(\mu) \text{ and } \mu(i) = P(s) \\
\sigma(s)(i) - 1 & \text{if } i \in \text{dom}(\mu) \text{ and } \mu(i) = V(s) \\
\sigma(s)(i) & \text{in all other cases}
\end{cases} $$
Action of a multi-instruction on a state

Assuming that $\mu$ is admissible at $\sigma$

The state $\sigma \cdot \mu$ is defined as follows.

- For every $x \in \mathcal{X}$, if there exists $i \in \{1, \ldots, n\}$ s.t. $\mu(i)$ is $x := \varepsilon$, then one has
  
  $$ (\sigma \cdot \mu)(x) = \varepsilon(\sigma|\mathcal{X}) $$

  Otherwise one has $(\sigma \cdot \mu)(x) = \sigma(x)$.

- For all $s \in S$ the multiset $(\sigma \cdot \mu)(s)$, seen as a mapping from $\{1, \ldots, n\}$ to $\mathbb{N}$, is given by

  $$ i \mapsto \begin{cases} 
  \sigma(s)(i) + 1 & \text{if } i \in \text{dom}(\mu) \text{ and } \mu(i) = P(s) \\
  \sigma(s)(i) - 1 & \text{if } i \in \text{dom}(\mu) \text{ and } \mu(i) = V(s) \\
  \sigma(s)(i) & \text{in all other cases}
  \end{cases} $$

A sequence $\mu_0, \ldots, \mu_{q-1}$ of multi-instructions is said to be admissible at state $\sigma$ when for all $k \in \{0, \ldots, q-1\}$ the multi-instruction $\mu_k$ is admissible at state $\sigma \cdot \mu_0 \cdots \mu_k$. 
Directed paths and sequences of multi-instructions
Directed paths and sequences of multi-instructions

A directed path $\gamma$ on $(G_1, \ldots, G_n)$ is a sequence $(\gamma(k))_{k \in \{0, \ldots, q\}}$ of points such that for all $k \in \{0, \ldots, q - 1\}$ we have
Directed paths and sequences of multi-instructions

A directed path $\gamma$ on $(G_1, \ldots, G_n)$ is a sequence $(\gamma(k))_{k \in \{0, \ldots, q\}}$ of points such that for all $k \in \{0, \ldots, q - 1\}$ we have

- $\gamma_i(k) = \gamma_i(k + 1)$ or $\gamma_i(k) = \partial \gamma_i(k + 1)$ for all $i \in \{1, \ldots, n\}$, or
Directed paths and sequences of multi-instructions

A directed path $\gamma$ on $(G_1, \ldots, G_n)$ is a sequence $(\gamma(k))_{k \in \{0, \ldots, q\}}$ of points such that for all $k \in \{0, \ldots, q - 1\}$ we have

- $\gamma_i(k) = \gamma_i(k + 1)$ or $\gamma_i(k) = \partial^- \gamma_i(k + 1)$ for all $i \in \{1, \ldots, n\}$, or
- $\gamma_i(k) = \gamma_i(k + 1)$ or $\partial^+ \gamma_i(k) = \gamma_i(k + 1)$ for all $i \in \{1, \ldots, n\}$. 
Directed paths and sequences of multi-instructions

A directed path $\gamma$ on $(G_1, \ldots, G_n)$ is a sequence $(\gamma(k))_{k \in \{0, \ldots, q\}}$ of points such that for all $k \in \{0, \ldots, q - 1\}$ we have

- $\gamma_i(k) = \gamma_i(k + 1)$ or $\gamma_i(k) = \partial^- \gamma_i(k + 1)$ for all $i \in \{1, \ldots, n\}$, or

- $\gamma_i(k) = \gamma_i(k + 1)$ or $\partial^+ \gamma_i(k) = \gamma_i(k + 1)$ for all $i \in \{1, \ldots, n\}$.

Then $\gamma$ is associated with a sequence of multi-instructions $(\mu_k)_{k \in \{0, \ldots, q - 1\}}$ defined for $k \in \{0, \ldots, q - 1\}$ by
Directed paths and sequences of multi-instructions

A directed path $\gamma$ on $(G_1, \ldots, G_n)$ is a sequence $(\gamma(k))_{k \in \{0, \ldots, q\}}$ of points such that for all $k \in \{0, \ldots, q-1\}$ we have

- $\gamma_i(k) = \gamma_i(k+1)$ or $\gamma_i(k) = \partial^- \gamma_i(k+1)$ for all $i \in \{1, \ldots, n\}$, or

- $\gamma_i(k) = \gamma_i(k+1)$ or $\partial^+ \gamma_i(k) = \gamma_i(k+1)$ for all $i \in \{1, \ldots, n\}$.

Then $\gamma$ is associated with a sequence of multi-instructions $(\mu_k)_{k \in \{0, \ldots, q-1\}}$ defined for $k \in \{0, \ldots, q-1\}$ by

- $\text{dom}(\mu_k) = \{ i \in \{1, \ldots, n\} \mid \gamma_i(k+1) = \partial^+ \gamma_i(k) \text{ or } \lambda_i(\gamma_i(k+1)) = W(\_\_\_)\}$
A directed path $\gamma$ on $(G_1, \ldots, G_n)$ is a sequence $(\gamma(k))_{k \in \{0, \ldots, q\}}$ of points such that for all $k \in \{0, \ldots, q - 1\}$ we have

- $\gamma_i(k) = \gamma_i(k + 1)$ or $\gamma_i(k) = \partial^- \gamma_i(k + 1)$ for all $i \in \{1, \ldots, n\}$, or
- $\gamma_i(k) = \gamma_i(k + 1)$ or $\partial^+ \gamma_i(k) = \gamma_i(k + 1)$ for all $i \in \{1, \ldots, n\}$.

Then $\gamma$ is associated with a sequence of multi-instructions $(\mu_k)_{k \in \{0, \ldots, q - 1\}}$ defined for $k \in \{0, \ldots, q - 1\}$ by

- $\text{dom}(\mu_k) = \{i \in \{1, \ldots, n\} \mid \gamma_i(k + 1) = \partial^+ \gamma_i(k) \text{ or } \lambda_i(\gamma_i(k + 1)) = W(\_))$
- $\mu_k(i) = \lambda_i(\gamma_i(k + 1))$ for all $k \in \{0, \ldots, q - 1\}$ and all $i \in \text{dom}(\mu_k)$
Discrete paths are “continuous”
Discrete paths are “continuous”
Discrete paths are “continuous”
Discrete paths are “continuous”
Discrete paths are “continuous”
Discrete paths are “continuous”
Discrete paths are “continuous”
Discrete paths are “continuous”
Discrete paths are “continuous”
Discrete paths are “continuous”
Discrete paths are “continuous”
Discrete paths are "continuous"
Discrete paths are “continuous”
Admissible paths and execution traces

Given a state of the program, a directed path is said to be admissible at that state when so is its associated sequence of multi-instructions at state. In this case we define the action of $\gamma$ on the right of $\sigma$ as follows.

$\sigma \cdot \gamma = \sigma \cdot \mu_0 \cdots \mu_{q-1}$

An admissible path is an execution trace when all the conditional branchings met along the way are respected: for all $k \in \{0, \ldots, q-2\}$ and all $i \in \{1, \ldots, n\}$ such that $\mu_k(i)$, which is by definition $\lambda_i(\gamma_i(k + 1))$, is a branching, we have

$\mu_k(i)(\sigma \cdot \mu_0 \cdots \mu_{k-1}) = \gamma_i(k + 2)$
Admissible paths and execution traces

Given \( \sigma \) a state of the program, a directed path is said to be admissible at \( \sigma \) when so is its associated sequence of multi-instructions at state \( \sigma \). In this case we define the action of \( \gamma \) on the right of \( \sigma \) as follows.

\[
\sigma \cdot \gamma = \sigma \cdot \mu_0 \cdots \mu_{q-1}
\]
Admissible paths and execution traces

Given $\sigma$ a state of the program, a directed path is said to be admissible at $\sigma$ when so is its associated sequence of multi-instructions at state $\sigma$. In this case we define the action of $\gamma$ on the right of $\sigma$ as follows.

$$\sigma \cdot \gamma = \sigma \cdot \mu_0 \cdot \ldots \cdot \mu_{q-1}$$

An admissible path is an execution trace when all the conditional branchings met along the way are respected: for all $k \in \{0, \ldots, q - 2\}$ and all $i \in \{1, \ldots, n\}$ such that $\mu_k(i)$, which is by definition $\lambda_i(\gamma_i(k + 1))$, is a branching, we have

$$(\mu_k(i)) (\sigma \cdot \mu_0 \cdot \ldots \cdot \mu_{k-1}) = \gamma_i(k + 2)$$
Concurrent access

```
var x = 0

proc p = x:=1

proc q = x:=2

init p q
```
Admissible execution trace

the value of $x$ is 0
Admissible execution trace

the value of $x$ is 0
Admissible execution trace

the value of $x$ is 0
Admissible execution trace

the value of x is 1
Admissible execution trace

the value of $x$ is 2
Admissible execution trace

the value of \( x \) is 2
Admissible execution trace

The value of $x$ is 2
Admissible execution trace

the value of x is 2
Not admissible execution trace

the value of $x$ is 0
Not admissible execution trace

the value of $x$ is 0
Not admissible execution trace

the value of $x$ is 0
Not admissible execution trace

the value of x is ?
Lack of resources

sem 1 a

proc p = P(a);V(a)

init 2p
Admissible concurrent execution trace
Admissible concurrent execution trace

sem 1 a
Admissible concurrent execution trace

sem 1 a
Admissible concurrent execution trace

sem 1 a
Admissible concurrent execution trace

\[ \text{sem } 1 \ a \]
Admissible concurrent execution trace
Admissible concurrent execution trace

sem 1 a
Admissible concurrent execution trace

\text{sem 1 a}
Admissible concurrent execution trace

\[ \text{sem 1 a} \]
Admissible concurrent execution trace
Admissible concurrent execution trace

sem 1 a
Admissible concurrent execution trace
Not admissible concurrent execution trace

\[ \text{sem } 1 \ a \]
Not admissible concurrent execution trace

\[ \text{sem } 1 \ a \]
Not admissible concurrent execution trace

\[ \text{sem} \ 1 \ a \]
Not admissible concurrent execution trace

sem 1 a
Synchronisation

sync 1 b

proc p = W(b)

init 2p
Concurrent execution trace

\( \text{sync } 1 \ b \)
Concurrent execution trace

sync 1 b
Concurrent execution trace

sync 1 b

\[ W(b) \]

\[ W(b) \otimes (q)M \]
Concurrent execution trace

sync 1 b
Concurrent execution trace

sync 1 b
Concurrent execution trace

sync 1 b
Concurrent execution trace

sync 1 b
Concurrent execution trace

sync 1 b
Not admissible concurrent execution trace

\[ \text{sync 1 b} \]
Not admissible concurrent execution trace

sync 1 b
Not admissible concurrent execution trace

\texttt{sync \ 1 \ b}
Not admissible concurrent execution trace

\[ \text{sync} \ 1 \ b \]
Next goal

Encode admissibility into a model.
CONSERVATIVE PROGRAMS
Potential Functions
The potential functions of processes and programs
The potential functions of processes and programs

A program $\Pi = (G_1, \ldots, G_n)$ is conservative when for all directed paths starting at the origin, the amount of semaphores held by the program at the end of the path only depends on its arrival point.
The potential functions of processes and programs

A program $\Pi = (G_1, \ldots, G_n)$ is conservative when for all directed paths starting at the origin, the amount of semaphores held by the program at the end of the path only depends on its arrival point.

For all initial states $\sigma$, for all directed paths $\gamma, \gamma'$ starting at the origin,

$$\partial^+ \gamma = \partial^+ \gamma' \implies \sigma \cdot \gamma|_S = \sigma \cdot \gamma'|_S$$

In particular, the program $\Pi$ comes with a potential function

$$F_\Pi : \{\text{semaphores}\} \times \{\text{points}\} \to \mathbb{N} \cong \{\text{points}\} \to \{\text{multisets over } S\}$$
The potential functions of processes and programs

A program \( \Pi = (G_1, \ldots, G_n) \) is conservative when for all directed paths starting at the origin, the amount of semaphores held by the program at the end of the path only depends on its arrival point.

For all initial states \( \sigma \), for all directed paths \( \gamma, \gamma' \) starting at the origin,

\[
\partial^+ \gamma = \partial^+ \gamma' \quad \Rightarrow \quad \sigma \cdot \gamma|_S = \sigma \cdot \gamma'|_S
\]

In particular, the program \( \Pi \) comes with a potential function

\[
F_\Pi : \{\text{semaphores}\} \times \{\text{points}\} \to \mathbb{N} \cong \{\text{points}\} \to \{\text{multisets over } S\}
\]

**Proposition:** The program \( \Pi \) is conservative if and only if
The potential functions of processes and programs

A program $\Pi = (G_1, \ldots, G_n)$ is conservative when for all directed paths starting at the origin, the amount of semaphores held by the program at the end of the path only depends on its arrival point.

For all initial states $\sigma$, for all directed paths $\gamma, \gamma'$ starting at the origin,

$$\partial^+ \gamma = \partial^+ \gamma' \quad \Rightarrow \quad \sigma \cdot \gamma|_S = \sigma \cdot \gamma'|_S$$

In particular, the program $\Pi$ comes with a potential function

$$F_{\Pi} : \{\text{semaphores}\} \times \{\text{points}\} \rightarrow \mathbb{N} \cong \{\text{points}\} \rightarrow \{\text{multisets over } S\}$$

**Proposition:** The program $\Pi$ is conservative if and only if so are its processes $G_1, \ldots, G_n$.
A program $\Pi = (G_1, \ldots, G_n)$ is conservative when for all directed paths starting at the origin, the amount of semaphores held by the program at the end of the path only depends on its arrival point.

For all initial states $\sigma$, for all directed paths $\gamma, \gamma'$ starting at the origin,

$$\partial^+ \gamma = \partial^+ \gamma' \implies \sigma \cdot \gamma|_S = \sigma \cdot \gamma'|_S$$

In particular, the program $\Pi$ comes with a potential function

$$F_\Pi : \{\text{semaphores}\} \times \{\text{points}\} \to \mathbb{N} \cong \{\text{points}\} \to \{\text{multisets over } S\}$$

**Proposition:** The program $\Pi$ is conservative if and only if so are its processes $G_1, \ldots, G_n$ and its potential function is given by

$$F_\Pi(p_1, \ldots, p_n) =$$
The potential functions of processes and programs

A program \( \Pi = (G_1, \ldots, G_n) \) is conservative when for all directed paths starting at the origin, the amount of semaphores held by the program at the end of the path only depends on its arrival point.

For all initial states \( \sigma \), for all directed paths \( \gamma, \gamma' \) starting at the origin,

\[
\partial^+ \gamma = \partial^+ \gamma' \Rightarrow \sigma \cdot \gamma|_S = \sigma \cdot \gamma'|_S
\]

In particular, the program \( \Pi \) comes with a potential function

\[
F_{\Pi} : \{\text{semaphores}\} \times \{\text{points}\} \rightarrow \mathbb{N} \cong \{\text{points}\} \rightarrow \{\text{multisets over } S\}
\]

**Proposition:** The program \( \Pi \) is conservative if and only if so are its processes \( G_1, \ldots, G_n \) and its potential function is given by

\[
F_{\Pi}(p_1, \ldots, p_n) = \sum_{k=1}^{n} F_{G_k}(p_k)
\]
Conservative process

example
Conservative process

diagram
Conservative process

example
Conservative process

example
Conservative process example
Conservative process

example
Conservative process

example
Conservative process

example
Conservative process

example
Conservative process

example
Conservative process

example
Conservative process

example
Conservative process

example
Conservative process

eample
Conservative process

example
Conservative process

example
Conservative process

example
Conservative process

example
Not conservative process

e.example

\begin{align*}
x &:= 0 \\
&\implies P(s) \\
&\implies x++ \\
&\implies x++ \\
\end{align*}
Not conservative process

example
Not conservative process

e Example

\[ P(s) \]
Not conservative process

example
Not conservative process

example
Not conservative process

eexample
Not conservative process

example
Not conservative process

example
Not conservative process

eexample
Not conservative process

eexample
Not conservative process

element
Not conservative process

example
Not conservative process

example
Not conservative process

example
Not conservative process

declaration
Not conservative process

example
Conservative programs

Potential functions

Not conservative process

example
Not conservative process

example
Conservativity is decidable

We inductively define a sequence of partial functions \( \pi_n \): \( \text{points} \to \mathbb{N} \). 

- The first term \( \pi_0 \) is only defined at the origin and \( \pi_0(\text{origin}) \) is the empty.

- Assuming that \( \pi_n \) is defined, for all pairs of points \( (p, p') \) such that:
  - \( \pi_n(p) \) is defined but not \( \pi_n(p') \), and
  - \( \partial_- p' = p \) or \( p' = \partial_+ p \),

we define a strict extension of \( \pi_n \), by setting:

\[
\begin{align*}
p' &\rightarrow \pi_n(p) \quad \text{if} \quad \partial_- p' = p \\
p' &\rightarrow \lambda(p') \quad \text{if} \quad p' = \partial_+ p
\end{align*}
\]

- If all these extensions are compatible, then \( \pi_{n+1} \) is their union.

- If all the points have been "visited" we have a finite chain of strict extensions \( \pi_0 \subseteq \cdots \subseteq \pi_n \subseteq \pi_n = \pi_{n+1} \) whose last element is denoted by \( \pi \).

- If the following holds for all ordered pairs of points \( (p, p') \) such that \( \partial_- p' = p \) or \( p' = \partial_+ p \), then \( G \) is conservative, otherwise it is not.

\[
\pi(p') = \begin{cases} 
\pi(p) & \text{if} \quad \partial_- p' = p \\
\lambda(p') & \text{if} \quad p' = \partial_+ p
\end{cases}
\]
Conservativity is decidable

We inductively define a sequence of partial functions $\pi_n : \{\text{points}\} \to \mathbb{N}^S$. 

- The first term $\pi_0$ is only defined at the origin and $\pi_0(\text{origin})$ is the empty.
- Assuming that $\pi_n$ is defined, for all pairs of points $(p, p')$ such that:
  - $\pi_n(p)$ is defined but not $\pi_n(p')$,
  - $\partial_- p' = p$ or $p' = \partial_+ p$,
we define a strict extension of $\pi_n$, by setting:
  - $p' \mapsto \pi_n(p)$ if $\partial_- p' = p$,
  - $\lambda(p')$ if $p' = \partial_+ p$.
- If all these extensions are compatible, then $\pi_{n+1}$ is their union.
- If all the points have been “visited” we have a finite chain of strict extensions $\pi_0 \subseteq \cdots \subseteq \pi_n \subseteq \pi_n + 1 = \pi$ whose last element is denoted by $\pi$.
- If the following holds for all ordered pairs of points $(p, p')$ such that $\partial_- p' = p$ or $p' = \partial_+ p$, then $G$ is conservative, otherwise it is not.

$\pi(p') = \pi(p)$ if $\partial_- p' = p$ $\lambda(p')$ if $p' = \partial_+ p$. 
Conservativity is decidable

We inductively define a sequence of partial functions $\pi_n : \{\text{points}\} \to \mathbb{N}^S$.
- The first term $\pi_0$ is only defined at the origin and $\pi_0(\text{origin})$ is the empty
Conservativity is decidable

We inductively define a sequence of partial functions $\pi_n : \{\text{points}\} \to \mathbb{N}^S$.

- The first term $\pi_0$ is only defined at the origin and $\pi_0(\text{origin})$ is the empty
- Assuming that $\pi_n$ is defined, for all pairs of points $(p, p')$ such that:
  - $\pi_n(p)$ is defined but not $\pi_n(p')$,
Conservativity is decidable

We inductively define a sequence of partial functions $\pi_n : \{\text{points}\} \to \mathbb{N}^S$.
- The first term $\pi_0$ is only defined at the origin and $\pi_0(\text{origin})$ is the empty
- Assuming that $\pi_n$ is defined, for all pairs of points $(p, p')$ such that:
  - $\pi_n(p)$ is defined but not $\pi_n(p')$, and
  - $\partial p' = p$ or $p' = \partial p$,
Conservativity is decidable

We inductively define a sequence of partial functions \( \pi_n : \{\text{points}\} \rightarrow \mathbb{N}^S \).

- The first term \( \pi_0 \) is only defined at the origin and \( \pi_0(\text{origin}) \) is the empty.

- Assuming that \( \pi_n \) is defined, for all pairs of points \((p, p')\) such that:
  - \( \pi_n(p) \) is defined but not \( \pi_n(p') \), and
  - \( \partial^{-} p' = p \) or \( p' = \partial^{-} p \),
  we define a strict extension of \( \pi_n \), by setting:

\[
p' \mapsto \begin{cases} 
\pi_n(p) & \text{if } \partial^{-} p' = p \\
\pi_n(p) \cdot \lambda(p') & \text{if } p' = \partial^{-} p 
\end{cases}
\]
Conservativity is decidable

We inductively define a sequence of partial functions \( \pi_n : \{\text{points}\} \to \mathbb{N}^S \).

- The first term \( \pi_0 \) is only defined at the origin and \( \pi_0(\text{origin}) \) is the empty set.

- Assuming that \( \pi_n \) is defined, for all pairs of points \((p, p')\) such that:
  - \( \pi_n(p) \) is defined but not \( \pi_n(p') \), and
  - \( \partial p' = p \) or \( p' = \partial p \),
we define a strict extension of \( \pi_n \), by setting:

\[
p' \mapsto \begin{cases} 
\pi_n(p) & \text{if } \partial p' = p \\
\pi_n(p) \cdot \lambda(p') & \text{if } p' = \partial p 
\end{cases}
\]

- If all these extensions are compatible, then \( \pi_{n+1} \) is their union. Otherwise the induction stops and the graph is not conservative.
Conservativity is decidable

We inductively define a sequence of partial functions $\pi_n : \{\text{points}\} \to \mathbb{N}^S$.

- The first term $\pi_0$ is only defined at the origin and $\pi_0(\text{origin})$ is the empty

- Assuming that $\pi_n$ is defined, for all pairs of points $(p, p')$ such that:
  - $\pi_n(p)$ is defined but not $\pi_n(p')$, and
  - $\partial p' = p$ or $p' = \partial^* p$,
we define a strict extension of $\pi_n$, by setting:

\[
p' \mapsto \begin{cases} 
\pi_n(p) & \text{if } \partial p' = p \\
\pi_n(p) \cdot \lambda(p') & \text{if } p' = \partial^* p 
\end{cases}
\]

- If all these extensions are compatible, then $\pi_{n+1}$ is their union.
  Otherwise the induction stops and the graph is not conservative.

- If all the points have been “visited” we have a finite chain of strict extensions

\[
\pi_0 \subseteq \cdots \subseteq \pi_n \subseteq \pi_{n+1} = \pi
\]

whose last element is denoted by $\pi$. 
Conservativity is decidable

We inductively define a sequence of partial functions $\pi_n : \{\text{points}\} \rightarrow \mathbb{N}^S$.

- The first term $\pi_0$ is only defined at the origin and $\pi_0(\text{origin})$ is the empty

- Assuming that $\pi_n$ is defined, for all pairs of points $(p, p')$ such that:
  
  - $\pi_n(p)$ is defined but not $\pi_n(p')$, and
  - $\partial p' = p$ or $p' = \partial^* p$,

  we define a strict extension of $\pi_n$, by setting:

  $$
  p' \mapsto \begin{cases} 
  \pi_n(p) & \text{if } \partial p' = p \\
  \pi_n(p) \cdot \lambda(p') & \text{if } p' = \partial^* p
  \end{cases}
  $$

- If all these extensions are compatible, then $\pi_{n+1}$ is their union.
  Otherwise the induction stops and the graph is not conservative.

- If all the points have been “visited” we have a finite chain of strict extensions

  $$
  \pi_0 \subseteq \cdots \subseteq \pi_n \subseteq \pi_{n+1} = \pi
  $$

  whose last element is denoted by $\pi$.

- If the following holds for all ordered pairs of points $(p, p')$ such that $\partial p' = p$ or $p' = \partial^* p$, then $G$ is conservative, otherwise it is not.

  $$
  \pi(p') = \begin{cases} 
  \pi(p) & \text{if } \partial p' = p \\
  \pi(p) \cdot \lambda(p') & \text{if } p' = \partial^* p
  \end{cases}
  $$
Discrete Models
The discrete model of a conservative program

A point \( p = (p_1, \ldots, p_n) \) of the conservative program is said to be:

- conflicting when \( \lambda_i(p_i) \) and \( \lambda_j(p_j) \) conflict for some \( i \neq j \),
- exhausting when there is some semaphore \( s \in S \) such that \( F(p_1, \ldots, p_n, s) \not\in \{0, \ldots, \text{arity}(s)\} \),
- desynchronizing when there is some synchronization barrier \( b \in B \) such that \( 0 < \text{card} \{i \in \{1, \ldots, n\} \mid \lambda_i(p_i) = W(b) \leq \text{arity}(b)\} \).

The forbidden set gathers all the conflicting, exhausting, and desynchronizing points.

\[ \{\text{forbidden}\} = \{\text{conflicting}\} \cup \{\text{exhausting}\} \cup \{\text{desynchronizing}\} \]

The discrete model is the complement of its forbidden set.

\[ \{\text{points of the program}\} \setminus \{\text{forbidden points}\} \]
The discrete model of a conservative program

A point $p = (p_1, \ldots, p_n)$ of the conservative program is said to be:

- conflicting when $\lambda_i(p_i)$ and $\lambda_j(p_j)$ conflict for some $i \neq j$,
- exhausting when there is some semaphore $s \in S$ such that $F(p_1, \ldots, p_n, s) \not\in \{0, \ldots, \text{arity}(s)\}$,
- desynchronizing when there is some synchronization barrier $b \in B$ such that $0 < \text{card} \{i \in \{1, \ldots, n\} \mid \lambda_i(p_i) = W(b) \leq \text{arity}(b)$.

The forbidden set gathers all the conflicting, exhausting, and desynchronizing points:

$\{\text{forbidden}\} = \{\text{conflicting}\} \cup \{\text{exhausting}\} \cup \{\text{desynchronizing}\}$

The discrete model is the complement of its forbidden set:

$\{\text{points of the program}\} \setminus \{\text{forbidden points}\}$
The discrete model of a conservative program

A point $p = (p_1, \ldots, p_n)$ of the conservative program is said to be:
- conflicting when $\lambda_i(p_i)$ and $\lambda_j(p_j)$ conflict for some $i \neq j$,
The discrete model of a conservative program

A point $p = (p_1, \ldots, p_n)$ of the conservative program is said to be:

- **conflicting** when $\lambda_i(p_i)$ and $\lambda_j(p_j)$ conflict for some $i \neq j$,
- **exhausting** when there is some semaphore $s \in S$ such that:

$$F(p_1, \ldots, p_n, s) \not\in \{0, \ldots, \text{arity}(s)\},$$

The forbidden set gathers all the conflicting, exhausting, and desynchronizing points.

$$\{\text{forbidden}\} = \{\text{conflicting}\} \cup \{\text{exhausting}\} \cup \{\text{desynchronizing}\}$$

The discrete model is the complement of its forbidden set.

$$\{\text{points of the program}\} \setminus \{\text{forbidden points}\}$$
The discrete model of a conservative program

A point \( p = (p_1, \ldots, p_n) \) of the conservative program is said to be:
- **conflicting** when \( \lambda_i(p_i) \) and \( \lambda_j(p_j) \) conflict for some \( i \neq j \),
- **exhausting** when there is some semaphore \( s \in S \) such that
  \[
  F(p_1, \ldots, p_n, s) \notin \{0, \ldots, \text{arity}(s)\},
  \]
- **desynchronizing** when there is some synchronization barrier \( b \in B \) such that
  \[
  0 < \text{card}\{i \in \{1, \ldots, n\} \mid \lambda_i(p_i) = W(b)\} \leq \text{arity}(b),
  \]
The discrete model of a conservative program

A point $p = (p_1, \ldots, p_n)$ of the conservative program is said to be:

- **conflicting** when $\lambda_i(p_i)$ and $\lambda_j(p_j)$ conflict for some $i \neq j$,
- **exhausting** when there is some semaphore $s \in S$ such that
  \[ F(p_1, \ldots, p_n, s) \notin \{0, \ldots, \text{arity}(s)\}, \]
- **desynchronizing** when there is some synchronization barrier $b \in B$ such that
  \[ 0 < \text{card}\{i \in \{1, \ldots, n\} | \lambda_i(p_i) = W(b)\} \leq \text{arity}(b), \]

The **forbidden** set gathers all the conflicting, exhausting, and desynchronizing points.

\[ \{\text{forbidden}\} = \{\text{conflicting}\} \cup \{\text{exhausting}\} \cup \{\text{desynchronizing}\} \]
The discrete model of a conservative program

A point \( p = (p_1, \ldots, p_n) \) of the conservative program is said to be:

- **conflicting** when \( \lambda_i(p_i) \) and \( \lambda_j(p_j) \) conflict for some \( i \neq j \),
- **exhausting** when there is some semaphore \( s \in S \) such that
  \[
  F(p_1, \ldots, p_n, s) \notin \{0, \ldots, \text{arity}(s)\},
  \]
- **desynchronizing** when there is some synchronization barrier \( b \in B \) such that
  \[
  0 < \text{card}\{i \in \{1, \ldots, n\} | \lambda_i(p_i) = \text{W(b)}\} \leq \text{arity}(b),
  \]

The **forbidden** set gathers all the conflicting, exhausting, and desynchronizing points.

\[
\{\text{forbidden}\} = \{\text{conflicting}\} \cup \{\text{exhausting}\} \cup \{\text{desynchronizing}\}
\]

The **discrete model** is the complement of its forbidden set.

\[
\{\text{points of the program}\} \setminus \{\text{forbidden points}\}
\]
Discrete model

sem 1 a
Discrete model

sem 1 a
Discrete model
Discrete model

sem 1 a
Discrete model

\[ \text{sem 1 a} \]
Discrete model

\[ \text{sem } 1 \text{ a} \]
Discrete model

\[ P(a) \otimes V(a) \]
Discrete Model

sync 1 b
Discrete Model

$\text{sync } 1 \ b$
Discrete Model

sync 1 b
Discrete Model

sync 1 b

\[ W(b) \otimes 0 \]
Discrete Model

\[ \text{sync } 1 \ b \]
Discrete Model

sync \(1 \ b\)
Discrete Model

sync 1 b
Main theorem of discrete models

- Soundness: any directed path on a discrete model (i.e., which does not meet any forbidden point) is admissible.
- Completeness: for each admissible path which meets a forbidden point, there exists a directed path which avoids them and such that both directed paths induce the same sequence of multi-instructions.
Main theorem of discrete models

– Soundness:
Main theorem of discrete models

– **Soundness**: any directed path on a discrete model (i.e. which does not meet any forbidden point) is ...
Main theorem of discrete models

- **Soundness**: any directed path on a discrete model (i.e. which does not meet any forbidden point) is admissible.
Main theorem of discrete models

- **Soundness**: any directed path on a discrete model (i.e. which does not meet any forbidden point) is admissible.

- **Completeness**: 

Main theorem of discrete models

- **Soundness**: any directed path on a discrete model (i.e. which does not meet any forbidden point) is admissible.

- **Completeness**: for each admissible path which meets a forbidden point ...
Main theorem of discrete models

- **Soundness**: any directed path on a discrete model (i.e. which does not meet any forbidden point) is admissible.

- **Completeness**: for each admissible path which meets a forbidden point there exists a directed path which avoids them and such that both directed paths induce the same sequence of multi-instructions.
Admissible execution trace

the value of $x$ is 0
Admissible execution trace

the value of $x$ is 0
Admissible execution trace

the value of $x$ is 0
Conservative programs

Discrete models

Admissible execution trace

the value of $x$ is 1
Admissible execution trace

the value of $x$ is 2
Admissible execution trace

the value of $x$ is 2
Admissible execution trace

the value of $x$ is 2
Admissible execution trace

the value of $x$ is 2
Admissible execution trace avoiding forbidden points

the value of $x$ is 0
Admissible execution trace avoiding forbidden points

the value of x is 0
Admissible execution trace avoiding forbidden points

the value of x is 0
Admissible execution trace avoiding forbidden points

the value of $x$ is 1
Conservative programs

Discrete models

Admissible execution trace avoiding forbidden points

the value of $x$ is 1
Admissible execution trace avoiding forbidden points

the value of $x$ is 2
Admissible execution trace avoiding forbidden points

the value of $x$ is 2
Admissible execution trace avoiding forbidden points

The value of $x$ is 2
Admissible execution trace avoiding forbidden points

the value of \( x \) is 2
Replacement

\[ x := 2 \]

\[ x := 1 \]