A QUICK OVERVIEW

OF

CONCURRENCY THEORY
PARALLEL AUTOMATA META LANGUAGE
Syntax
Paradigm


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`

\(^1\)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)

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. POSIX\(^1\) Threads

\(^1\)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\(^1\) Threads
- Parallel compound can occur anywhere in a program e.g.

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

\(^1\)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\(^1\) Threads

- Parallel compound can occur anywhere in a program e.g.

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

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

\(^1\)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\(^1\) 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:
  - Operator `||` in outermost position: only sequential processes are executed in parallel

\(^1\)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 \( \text{parbegin} \ldots \text{parend} \)
- 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.

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

- The Carson and Reynolds language is a restriction of Dijkstra’s language:
  - Operator \( \mid \mid \) 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
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 $||$ 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

<table>
<thead>
<tr>
<th>Declaration Type</th>
<th>Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>sem</td>
<td>int set of identifiers</td>
<td>sem 3 a b c d</td>
</tr>
<tr>
<td>sync</td>
<td>int set of identifiers</td>
<td>sync 3 a b c d</td>
</tr>
<tr>
<td>mtx</td>
<td>set of identifiers</td>
<td>mtx a b c d</td>
</tr>
<tr>
<td>var</td>
<td>identifier = constant</td>
<td>var x = 0</td>
</tr>
<tr>
<td>proc</td>
<td>identifier = basic block</td>
<td>proc y =</td>
</tr>
<tr>
<td>init</td>
<td>multiset of identifiers</td>
<td>init a 2b 3c</td>
</tr>
</tbody>
</table>
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`
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`
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`
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>
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:

- $v \in V$ (content of $v$)
- $x \in R$ (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)

For nullary and unary operators, the conditions are:

- $\bot, x \in R, v \in V$
- $\land, \lor, +, -, \times, /, <, >, \leq, \geq, =, \neq \in R$

Expressiveness is captured by the ability to form $\neg v, v + 1$. The support of $\bot$ for forming $\bot = v$ is a key aspect of the language's expressiveness.

The language has $2^{\Gamma} - 1$ distinct (nullary) operators and $2^{\Pi} - 1$ (nullary or unary) operators, where $\Gamma$ and $\Pi$ are the sets of nullary and unary operators, respectively.

- $\Pi$ contains $\bot$ and $\neg$
- $\Gamma$ contains $\bot$ and $\neg$

This structure allows for the formation of complex expressions through the application of these operators to values and other expressions.
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>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ν</td>
<td>content of $\nu \in \mathcal{V}$</td>
</tr>
<tr>
<td>∧</td>
<td>minimum</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td>≤</td>
<td>less or equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>strictly less</td>
</tr>
<tr>
<td>=</td>
<td>equal</td>
</tr>
<tr>
<td>¬</td>
<td>complement</td>
</tr>
<tr>
<td>⊥</td>
<td>bottom</td>
</tr>
<tr>
<td>x ∈ ℝ</td>
<td>constant</td>
</tr>
<tr>
<td>ν</td>
<td>maximum</td>
</tr>
<tr>
<td>−</td>
<td>substraction</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td>≥</td>
<td>greater of equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>strictly greater</td>
</tr>
<tr>
<td>≠</td>
<td>not equal</td>
</tr>
<tr>
<td>%</td>
<td>modulo</td>
</tr>
<tr>
<td>✗</td>
<td>nullary</td>
</tr>
<tr>
<td>⊤</td>
<td>unary</td>
</tr>
<tr>
<td>¬</td>
<td>binary</td>
</tr>
</tbody>
</table>

The table above shows the operators and their meanings in the context of the Parallel Automata Meta Language.
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

- 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

- \textit{identifier} := \textit{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
Non branching instructions

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

- \texttt{identifier:=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 \texttt{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 \texttt{arity} + 1 of them are stopped by the barrier \texttt{identifier}
- \texttt{J(identifier)} the execution of the process is stopped and the one of a copy of \texttt{identifier} starts. There is no return mechanism.
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
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.
Branching

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

\[(L_1) + [e_1] + (L_2) + [e_2] + \cdots + (L_n) + [e_n] + (L_{n+1})\]
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
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
- If all the expressions evaluate to zero, then \(L_{n+1}\) is triggered.
Describing a process

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

For example, 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)
```

Due to the branchings, basic blocks are actually trees.
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

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

Due to the branchings, basic blocks are actually trees.
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

```
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)
```
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

```
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

\[ x := f \]

\[ a_1 \]

\[ b_1 \]

\[ \phi? \]

\[ a_1 \]

\[ b_1 \]

\[ No \]

\[ b_2 \]

\[ a_1 \]

\[ HALT \]

\[ a_1 \]

\[ START \]

\[ b_1 \]

\[ a_1 \]

\[ a_2 \]

\[ b_1 \]
The Hasse-Syracuse algorithm in PAML

var x = 7

proc p = ()+[x=1]+J(q)

proc q = (x:=x/2) + [x%2=0] + (x:=3*x+1) ; J(p)

init p
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

\[ J(q) \]

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

\( x = 1 \)

\[ J(q) \]

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

\[ J(p) \]

\[ x = x/2 \]

\[ x \mod 2 = 0 \]

\[ J(p) \]

\[ x = 3x + 1 \]
Building the control flow graph of the Hasse-Syracuse algorithm

\[ x = 1 \]

\[ J(q) \]

\[ x = x / 2 \]

\[ x \% 2 = 0 \]

\[ J(p) \]

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

\[ J(p) \]
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 \( q \)

\( x = 1 \)

\( x := 3 \times x + 1 \)

\( x := x / 2 \)

\( x \% 2 = 0 \)
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 \( q \)
Building the control flow graph of the Hasse-Syracuse algorithm
Reducing the Control Flow Graph
of the Hasse-Syracuse algorithm
Reducing the Control Flow Graph of the Hasse-Syracuse algorithm
Reducing the Control Flow Graph
of the Hasse-Syracuse algorithm
An execution trace on a 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
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

\[ x := x / 2 \]

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

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 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

- $x := x / 2$
- $x := 3 * x + 1$
- $x \mod 2 = 0$
- The current value of $x$ is 11

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

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

x := x/2

x := 3*x + 1

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

entry point

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

entry point

$x := \frac{x}{2}$

$x := 3x + 1$

$x \mod 2 = 0$

$x = 1$

the current value of $x$ is 16

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
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

entry point

$\text{x := x/2}$

$x \equiv 1$

$x \% 2 = 0$

$x := 3 \times x + 1$

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
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

The current value of $x$ is 2
Parallel Automata Meta Language

The control flow graphs

An execution trace on a control flow graph

of the Hasse-Syracuse algorithm

$x:=x/2$

$x:=3*x+1$

$x=1$

entry point

$x\%2=0$

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
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
- The (infinite) collection of paths is entirely determined by the (finite) control flow graph
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 goals of the course is to provide such a structure for a large class of PAML programs.
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* be *changed* 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 be changed at runtime
- There is no pointer arithmetics
Abstract Machine
Abstract expressions
Abstract expressions

- The set of variables of a program is $X$. 
Abstract expressions

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

Interpretation of expressions

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

- $\llbracket x \rrbracket_\nu = \nu(x)$ for all $x \in \mathcal{X}$
The abstract machine

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 \textbf{true} while 0 stands for \textbf{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
- \([\neg] : \mathbb{R}_\bot \rightarrow \mathbb{R}_\bot\),
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
- $[\neg] : \mathbb{R}_\perp \to \mathbb{R}_\perp$
  $[\neg](0) = 1$,
Interpretation of expressions
only depends on the current memory state

- $\llbracket x \rrbracket_\nu = \nu(x)$ for all $x \in X$
- Any value in $\mathbb{R} \setminus \{0\}$ stands for true while 0 stands for false
- $\llbracket \neg \rrbracket : \mathbb{R}_\bot \to \mathbb{R}_\bot$, 
  $\llbracket \neg \rrbracket(0) = 1$, 
  $\llbracket \neg \rrbracket(\bot) = \bot$, and
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 \( x \in \mathbb{R} \setminus \{0\} \)
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 $x \in \mathcal{X}$
- Any value in $\mathbb{R} \setminus \{0\}$ stands for true while 0 stands for false
- $[-] : \mathbb{R}_\bot \to \mathbb{R}_\bot,$
  $[-](0) = 1,$
  $[-](\bot) = \bot,$ and
  $[-](x) = 0$ 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
Parallel Automata Meta Language

The abstract machine

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 $X \times E$ yet we write $x := \varepsilon$ instead of $(x, \varepsilon)$. By extension $F(x := \varepsilon) = F(\varepsilon)$. 
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 $x := \varepsilon$ instead of $(x, \varepsilon)$. By extension $\mathcal{F}(x := \varepsilon) = \mathcal{F}(\varepsilon)$.
- Given a graph

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

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 $\mathcal{F}(\beta) \subseteq \mathcal{X}$ 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 $\mathcal{X} \times \mathcal{E}$ yet we write $x := \varepsilon$ instead of $(x, \varepsilon)$. By extension $\mathcal{F}(x := \varepsilon) = \mathcal{F}(\varepsilon)$.

- Given a graph

$$G : A \xrightarrow{\partial^-} \xrightarrow{\partial^+} V$$

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 $\mathcal{F}(\beta) \subseteq \mathcal{X}$ such that if the valuations $\nu$ and $\nu'$ match on $\mathcal{F}(\beta)$ 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

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

- 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 $\nu : \mathcal{X} \to \mathbb{R}$ which provides the values of the variables at the beginning of each execution of the program.
- The arity map $\alpha : S \sqcup B \to \mathbb{N} \cup \{\infty\}$. 
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 $\alpha : S \cup B \to \mathbb{N} \cup \{\infty\}$.
- 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 \(\mu : \{1, \ldots, n\} \rightarrow \{\text{instructions}\} \).
The internal states of the abstract machine
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 $x \in \mathcal{X}$, $\sigma(x) \in \mathbb{R}_\perp$,
- 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 \( \mathcal{X} \sqcup S \) 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 $\mathcal{X} \sqcup S$ such that:
- for all $x \in \mathcal{X}$, $\sigma(x) \in \mathbb{R}_\perp$, and
- for all $s \in S$, $\sigma(s)$ is a multiset over $\{1, \ldots, n\}$. 
Admissible multi-instructions

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 $s \in S$, $\phi(s) \leq \alpha(s)$ where $\phi(s) = |\sigma(s)| + \text{card}\{i \in \text{dom}(\mu) | \mu(i) = \text{P}(s)\} - \text{card}\{i \in \text{dom}(\mu) | \mu(i) = \text{V}(s); \sigma(s)(i) > 0\}$
- for all $b \in B$, $\text{card}\{i \in \text{dom}(\mu) | \mu(i) = \text{W}(b)\} \notin \{1, \ldots, \alpha(b)\}$
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: \( x := \varepsilon \) vs \( x := \varepsilon' \)
- read-write: \( x := \varepsilon \) 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:

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

\[
\phi(s) = |\sigma(s)| + \text{card}\{i \in \text{dom}(\mu) \mid \mu(i) = P(s)\} - \text{card}\{i \in \text{dom}(\mu) \mid \mu(i) = V(s); \, \sigma(s)(i) > 0\}
\]
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 \( s \in S \), \( \phi(s) \leq \alpha(s) \) where
  \[
  \phi(s) = |\sigma(s)| + \text{card}\{i \in \text{dom}(\mu) \mid \mu(i) = P(s)\} - \text{card}\{i \in \text{dom}(\mu) \mid \mu(i) = V(s); \ \sigma(s)(i) > 0\}
  \]
- for all \( b \in B \), \( \text{card}\{i \in \text{dom}(\mu) \mid \mu(i) = W(b)\} \notin \{1, \ldots, \alpha(b)\} \)
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

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

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

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

$$
i \mapsto \begin{cases} 
\sigma(s)(i) + 1 & \text{if } i \in \text{dom}(\mu) \text{ and } \mu(i) = P(s) \\
\max(0, \sigma(s)(i) - 1) & \text{if } i \in \text{dom}(\mu) \text{ and } \mu(i) = V(s) \\
\sigma(s)(i) & \text{in all other cases}
\end{cases}
$$
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
  $$i \mapsto \begin{cases} 
  \sigma(s)(i) + 1 & \text{if } i \in \text{dom}(\mu) \text{ and } \mu(i) = P(s) \\
  \max(0, \sigma(s)(i) - 1) & \text{if } i \in \text{dom}(\mu) \text{ and } \mu(i) = V(s) \\
  \sigma(s)(i) & \text{in all other cases}
  \end{cases}$$

A sequence $\mu_0, \ldots, \mu_{q-1}$ of multi-intructions 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 $\sigma \cdot \mu_0 \cdots \mu_{k-1}$. 
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
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

- \( \text{dom}(\mu_k) = \{ i \in \{1, \ldots, n\} \mid \gamma_i(k+1) = \partial^+ \gamma_i(k) \text{ or } \lambda_i(\gamma_i(k+1)) = W(\_)/ W(\_1) \} \)
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

- \( \text{dom}(\mu_k) = \{ i \in \{1, \ldots, n\} \mid \gamma_i(k + 1) = \partial^+ \gamma_i(k) \text{ or } \lambda_i(\gamma_i(k + 1)) = W(-) \} \)

- \( \mu_k(i) = \lambda_i(\gamma_i(k + 1)) \) 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
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.

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

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

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

```plaintext
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

\[ x := 2 \]

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

\[
\text{sem } 1 \ a \\
\text{proc } p = P(a); V(a) \\
\text{init } 2p
\]
Admissible concurrent execution trace

\texttt{sem 1 a}
Admissible concurrent execution trace
Admissible concurrent execution trace

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

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

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

sem 1 a
Admissible concurrent execution trace

sem 1 a
Admissible concurrent execution trace
Admissible concurrent execution trace

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

sem 1 a
Admissible concurrent execution trace

sem 1 a
Admissible concurrent execution trace

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

sem 1 a
Not admissible concurrent execution trace

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

sem 1 a
Not admissible concurrent execution trace

sem 1 a
Synchronisation

sync 1 b

proc p = W(b)

init 2p
Concurrent execution trace

sync 1 b
Concurrent execution trace

\[ \text{sync } 1 \ b \]
Concurrent execution trace

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

sync 1 b
Concurrent execution trace

sync 1 \( b \)
Concurrent execution trace

```
sync 1 b
```
Concurrent execution trace

\[ \text{sync} \quad 1 \quad b \]
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

sync 1 b
Not admissible concurrent execution trace

sync 1 b
Next goal

Encode admissibility into a model.
CONSERVATIVE PROGRAMS
Potential Functions
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 \( F_\Pi: \{\text{semaphores}\} \times \{\text{points}\} \rightarrow \mathbb{N} \)

\[
\sim: = \{\text{points}\} \rightarrow \text{multisets over } S
\]

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

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

$$F_\Pi : \{\text{semaphores}\} \times \{\text{points}\} \to \mathbb{N} \cong \{\text{points}\} \to \{\text{multisets over } S\}$$
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
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 $G_1, \ldots, G_n$
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

$$F_\Pi(p_1, \ldots, p_n) =$$
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

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

**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

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

eexample
Conservative process

example
Conservative process

eample
Conservative process

example
Conservative process

example
Conservative process

example
Conservative process

example
Not conservative process

example
Not conservative process

example
Not conservative process

example
Not conservative process

equation
Not conservative process

example
Not conservative process

example
Not conservative process

eexample
Not conservative process

eexample
Not conservative process
dexample

$P(s)$
Not conservative process

eexample
Not conservative process

eexample
Not conservative process

example

$P(s)$
Not conservative process

example
Not conservative process

example
Not conservative process

example
Not conservative process

example
Not conservative process

example
Not conservative process

example
Conservativity is decidable

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

- 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 $\pi_n(p')$,
  - $\partial - p' = p$ or $p' = \partial + p$,

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

$$p' \mapsto \pi_n(p) \quad \text{if} \quad \partial - p' = p$$

$$p' \mapsto \lambda(p') \quad \text{if} \quad p' = \partial + p$$

- 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.
Conservativity is decidable

We inductively define a sequence of partial functions $\pi_n : \{\text{points}\} \to \mathbb{N}^S$. 
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
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 $\pi_n(p')$, 
  - $\partial_-(p') = p$ or $p' = \partial_+ + p$, 

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.

$\pi(p') = \pi(p)$ if $\partial_-(p') = p$ $\pi(p)$ if $p' = \partial_+ + p$. 

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 $p' = \partial^+ p$,
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:

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

- 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 $\pi_n : \{\text{points}\} \to \mathbb{N}^S$.

- 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 $\pi$.
- 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 \( F(p_1, \ldots, p_n, s) > \text{arity}(s) \),
- 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 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 $F(p_1, \ldots, p_n, s) > \text{arity}(s)$,
- desynchronizing when there is some synchronization barrier $b \in B$ such that $0 < |\{i \in \{1, \ldots, n\} | \lambda_i(p_i) = W(b) \leq \text{arity}(b)|$.

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

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

The discrete model 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 $\lambda_i(p)$ and $\lambda_j(p)$ 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
  $$F(p_1, \ldots, p_n, s) > \text{arity}(s),$$
- **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.

$$\{\text{points of the program}\} \setminus \{\text{forbidden points}\}$$
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
  \[
  F(p_1, \ldots, p_n, s) > \text{arity}(s) ,
  \]
  
- **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 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
  \[
  F(p_1, \ldots, p_n, s) > \text{arity}(s),
  \]
- **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
  \[
  F(p_1, \ldots, p_n, s) > \text{arity}(s),
  \]
- 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 is the complement of its forbidden set.

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

\[ V(a) \]

\[ P(a) \]

\( \otimes \)
Discrete model

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

$\text{sem } 1 \ a$
Discrete model

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

sem 1 a
Discrete model

sem 1 a
Discrete model

$P(a) \otimes V(a)$
Discrete Model

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

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

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

\text{sync 1 b}
Discrete Model

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

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

sync 1 b

\[
\begin{array}{ccc}
W(b) & \\
\emptyset & \\
\end{array}
\]
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
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
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