DIRECTED ALGEBRAIC TOPOLOGY

AND

CONCURRENCY

Emmanuel Haucourt

emmanuel.haucourt@polytechnique.edu

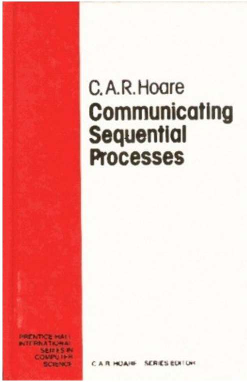
MPRI : Concurrency (2.3.1)
– Lecture 1 –

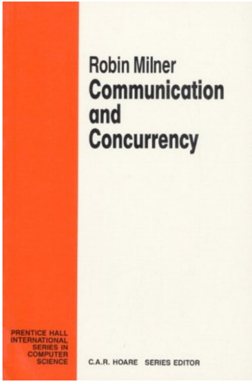
2024 – 2025

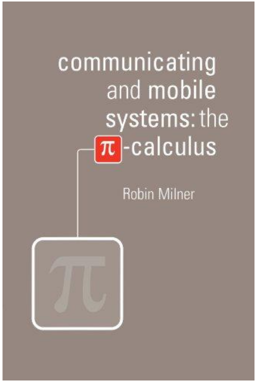
A QUICK OVERVIEW

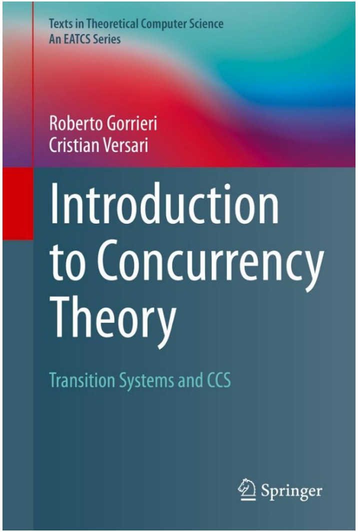
OF

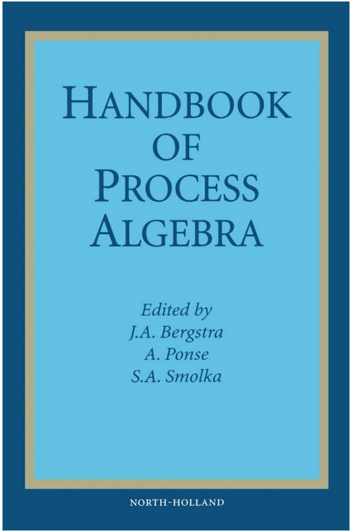
CONCURRENCY THEORY


C.A.R.Hoare
Communicating Sequential Processes


Robin Milner
Communication and Concurrency
PRENTICE HALL INTERNATIONAL SERIES IN COMPUTER SCIENCE
C.A.R. HOARE SERIES EDITOR


communicating and mobile systems: the π-calculus
Robin Milner


Texts in Theoretical Computer Science
An EATCS Series
Roberto Gorrieri
Cristian Versari
Introduction to Concurrency Theory
Transition Systems and CCS
Springer

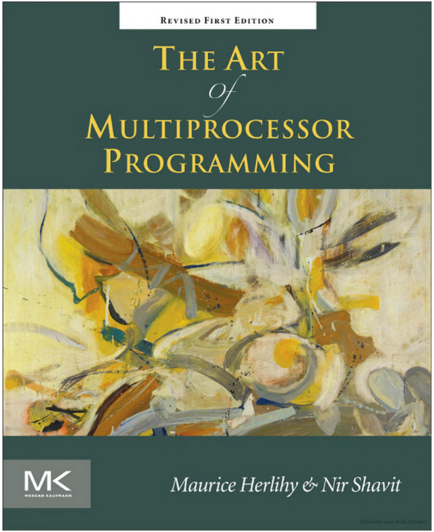

Handbook of Process Algebra
Edited by
J.A. Bergstra
A. Ponse
S.A. Smolka
NORTH-HOLLAND

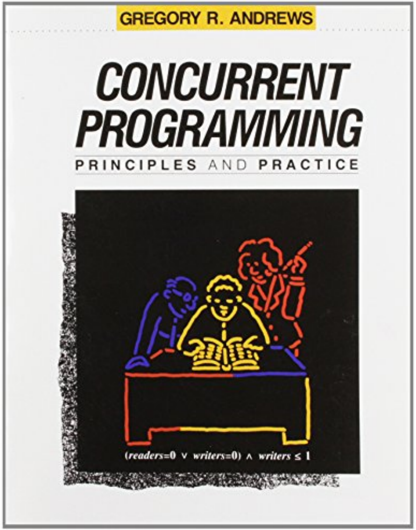
Handbook of Logic in Computer Science

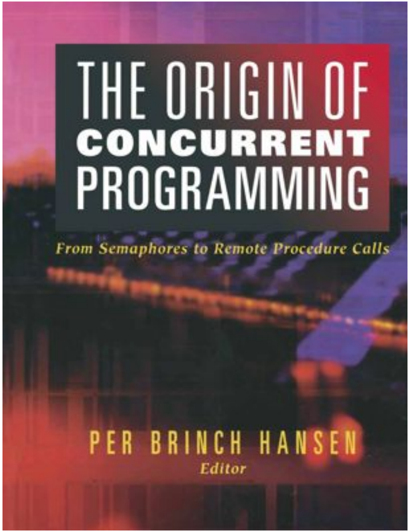
VOLUME 4

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

OXFORD SCIENCE PUBLICATIONS


Revised First Edition
The Art of Multiprocessor Programming
MK
Morgan Kaufmann
Maurice Herlihy & Nir Shavit


GREGORY R. ANDREWS
CONCURRENT PROGRAMMING
PRINCIPLES AND PRACTICE
(readers=0 ∨ writers=0) ∧ writers ≤ 1


THE ORIGIN OF
CONCURRENT
PROGRAMMING
From Semaphores to Remote Procedure Calls
PER BRINCH HANSEN
Editor

PARALLEL AUTOMATA META LANGUAGE

Syntax

Paradigm

Cooperating sequential processes, E. W. Dijkstra, 1965.
System deadlocks, E. G. Coffman, M. J. Elphick, and A. Shoshani, 1971.
The geometry of semaphore programs, S. D. Carson and P. F. Reynolds, 1987.

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

Paradigm

Cooperating sequential processes, E. W. Dijkstra, 1965.
System deadlocks, E. G. Coffman, M. J. Elphick, and A. Shoshani, 1971.
The geometry of semaphore programs, S. D. Carson and P. F. Reynolds, 1987.

- 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

Cooperating sequential processes, E. W. Dijkstra, 1965.
System deadlocks, E. G. Coffman, M. J. Elphick, and A. Shoshani, 1971.
The geometry of semaphore programs, S. D. Carson and P. F. Reynolds, 1987.

- 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

Cooperating sequential processes, E. W. Dijkstra, 1965.
System deadlocks, E. G. Coffman, M. J. Elphick, and A. Shoshani, 1971.
The geometry of semaphore programs, S. D. Carson and P. F. Reynolds, 1987.

- 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

Cooperating sequential processes, E. W. Dijkstra, 1965.
System deadlocks, E. G. Coffman, M. J. Elphick, and A. Shoshani, 1971.
The geometry of semaphore programs, S. D. Carson and P. F. Reynolds, 1987.

- 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

Cooperating sequential processes, E. W. Dijkstra, 1965.
System deadlocks, E. G. Coffman, M. J. Elphick, and A. Shoshani, 1971.
The geometry of semaphore programs, S. D. Carson and P. F. Reynolds, 1987.

- 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:

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

Paradigm

Cooperating sequential processes, E. W. Dijkstra, 1965.
System deadlocks, E. G. Coffman, M. J. Elphick, and A. Shoshani, 1971.
The geometry of semaphore programs, S. D. Carson and P. F. Reynolds, 1987.

- 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

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

Paradigm

Cooperating sequential processes, E. W. Dijkstra, 1965.
System deadlocks, E. G. Coffman, M. J. Elphick, and A. Shoshani, 1971.
The geometry of semaphore programs, S. D. Carson and P. F. Reynolds, 1987.

- 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

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

Declarations

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

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`

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`

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`

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`

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>

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`

Expressions and values

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

v	content of $v \in \mathcal{V}$	$x \in \mathbb{R}$	constant
$\wedge$	minimum	$\vee$	maximum
$+$	addition	$-$	substraction
$*$	multiplication	$/$	division
$\leqslant$	less or equal	$\geqslant$	greater of equal
$<$	strictly less	$>$	strictly greater
$=$	equal	$\neq$	not equal
$\neg$	complement	$\%$	modulo
$\perp$	bottom		

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

Non branching instructions

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 occurence of the resource *identifier* (there are *arity* available tokens), stops the process otherwise

Non branching instructions

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

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

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

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

Describing a process

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

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

Describing a process

The body of a process is just a (possibly empty) sequence of intructions, 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]+()
```

Describing a process

The body of a process is just a (possibly empty) sequence of intructions, 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 intructions, 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 intructions, 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

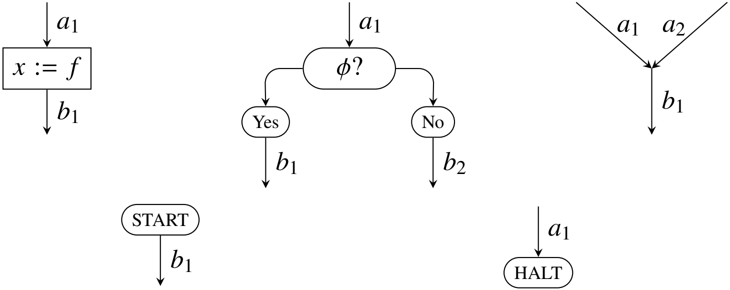
- Compilers and static analyzers internal representation of programs.
- No theoretical definition yet control flow graphs must be finite for practical reasons.
- At the core of many softwares dealing with source code
  e.g. GCC (cf. "basic blocks"), LLVM, Frama-C.

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



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

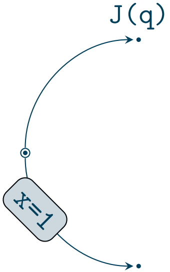
of the Hasse-Syracuse algorithm

Building the control flow graph

of the Hasse-Syracuse algorithm

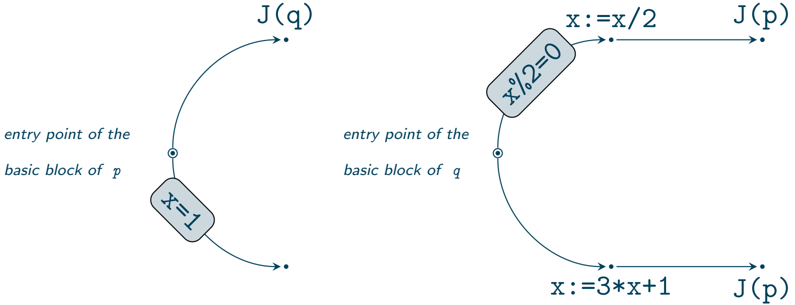
entry point of the

basic block of p



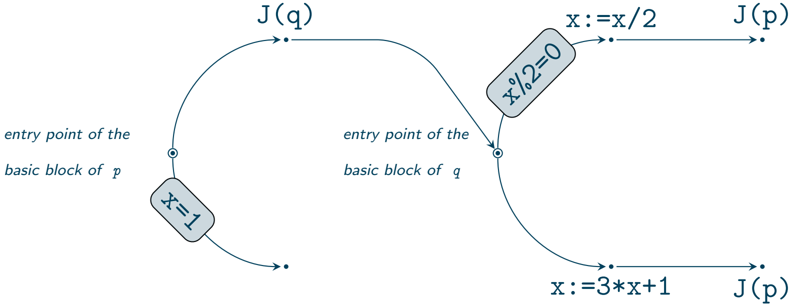
Building the control flow graph

of the Hasse-Syracuse algorithm



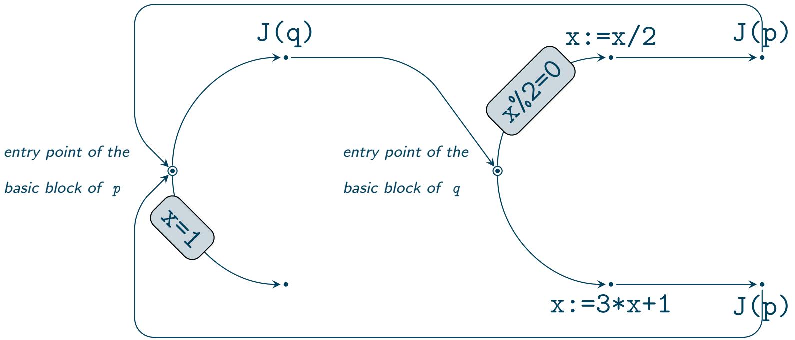
Building the control flow graph

of the Hasse-Syracuse algorithm



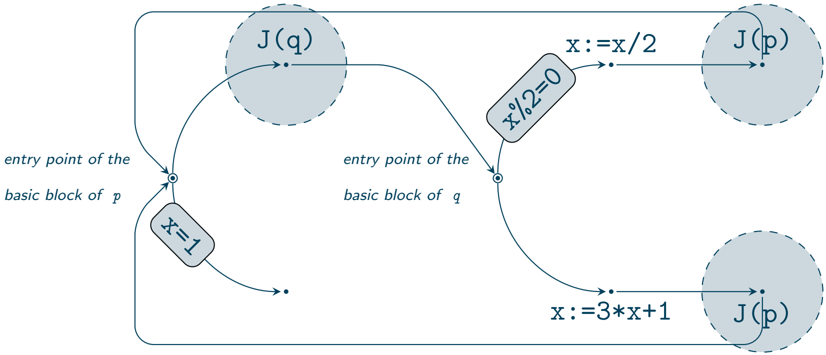
Building the control flow graph

of the Hasse-Syracuse algorithm



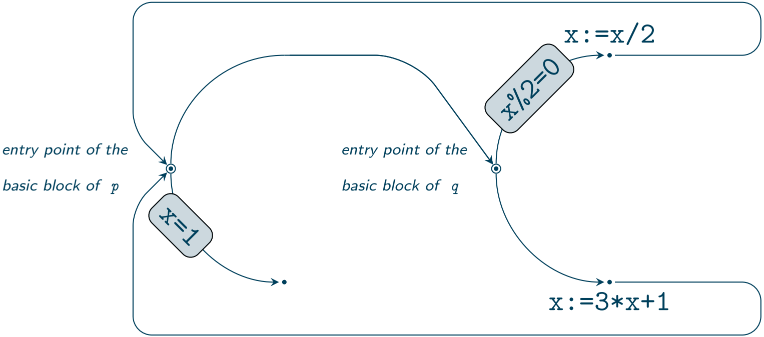
Building the control flow graph

of the Hasse-Syracuse algorithm



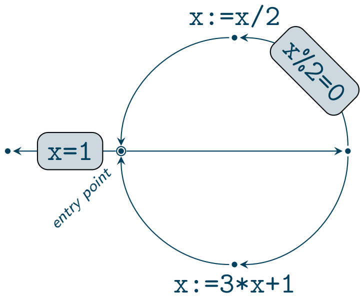
Building the control flow graph

of the Hasse-Syracuse algorithm



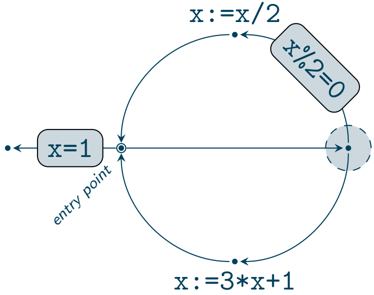
Reducing the Control Flow Graph

of the Hasse-Syracuse algorithm



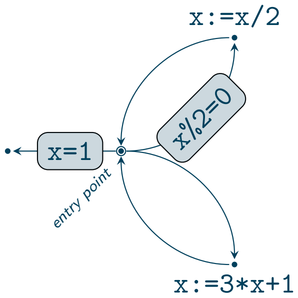
Reducing the Control Flow Graph

of the Hasse-Syracuse algorithm



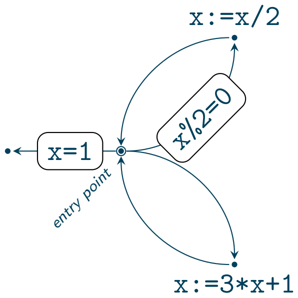
Reducing the Control Flow Graph

of the Hasse-Syracuse algorithm



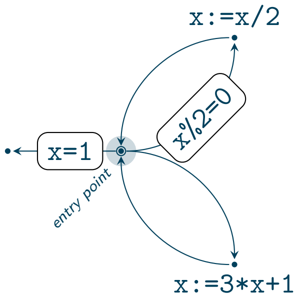
An execution trace on a control flow graph

of the Hasse-Syracuse algorithm



An execution trace on a control flow graph

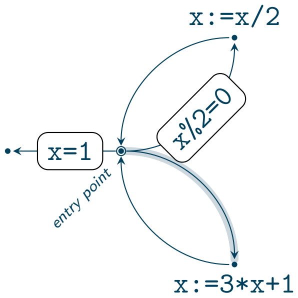
of the Hasse-Syracuse algorithm



the current value of x is 7

An execution trace on a control flow graph

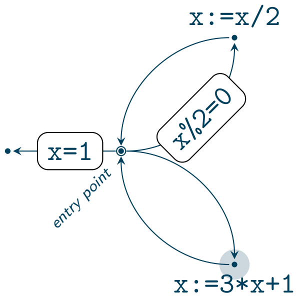
of the Hasse-Syracuse algorithm



the current value of x is 7

An execution trace on a control flow graph

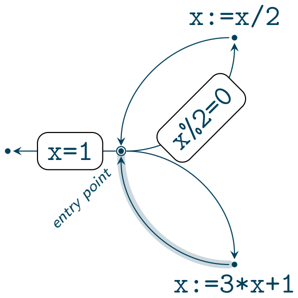
of the Hasse-Syracuse algorithm



the current value of `x` is 22

An execution trace on a control flow graph

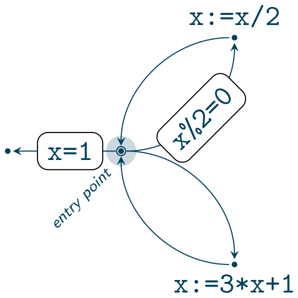
of the Hasse-Syracuse algorithm



the current value of x is 22

An execution trace on a control flow graph

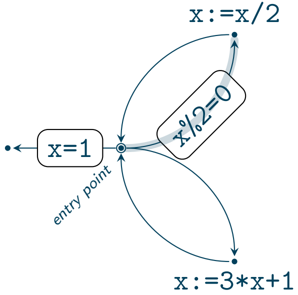
of the Hasse-Syracuse algorithm



the current value of x is 22

An execution trace on a control flow graph

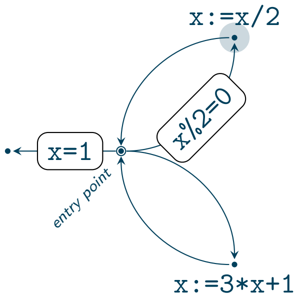
of the Hasse-Syracuse algorithm



the current value of `x` is 22

An execution trace on a control flow graph

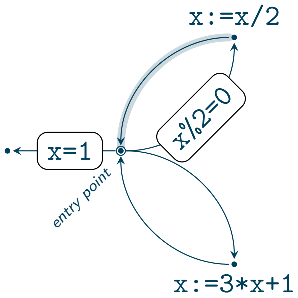
of the Hasse-Syracuse algorithm



the current value of x is 11

An execution trace on a control flow graph

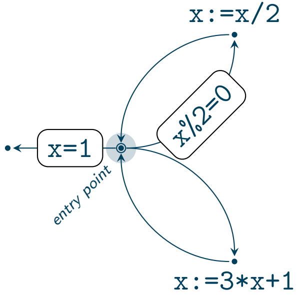
of the Hasse-Syracuse algorithm



the current value of x is 11

An execution trace on a control flow graph

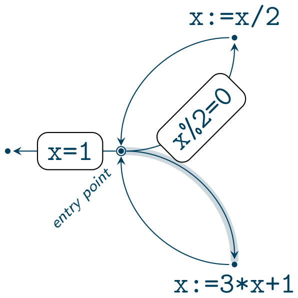
of the Hasse-Syracuse algorithm



the current value of x is 11

An execution trace on a control flow graph

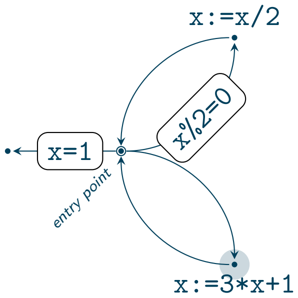
of the Hasse-Syracuse algorithm



the current value of `x` is 11

An execution trace on a control flow graph

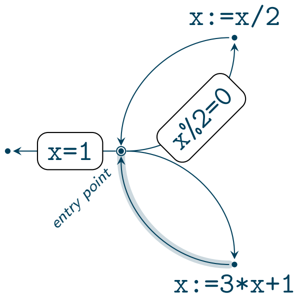
of the Hasse-Syracuse algorithm



the current value of x is 34

An execution trace on a control flow graph

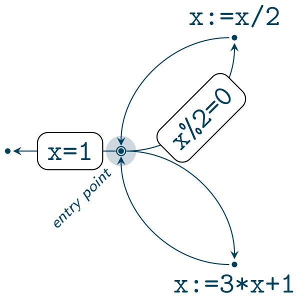
of the Hasse-Syracuse algorithm



the current value of `x` is 34

An execution trace on a control flow graph

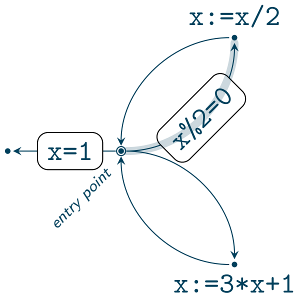
of the Hasse-Syracuse algorithm



the current value of `x` is 34

An execution trace on a control flow graph

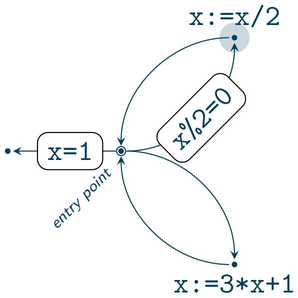
of the Hasse-Syracuse algorithm



the current value of x is 34

An execution trace on a control flow graph

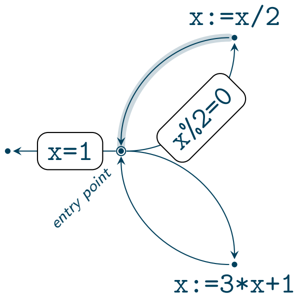
of the Hasse-Syracuse algorithm



the current value of x is 17

An execution trace on a control flow graph

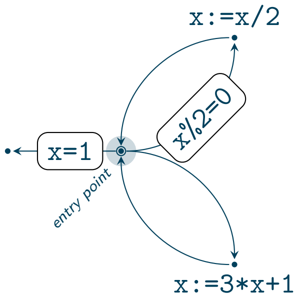
of the Hasse-Syracuse algorithm



the current value of x is 17

An execution trace on a control flow graph

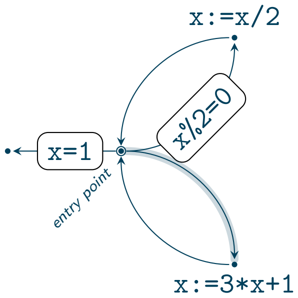
of the Hasse-Syracuse algorithm



the current value of `x` is 17

An execution trace on a control flow graph

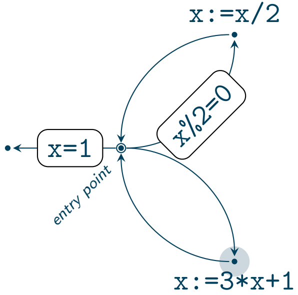
of the Hasse-Syracuse algorithm



the current value of x is 17

An execution trace on a control flow graph

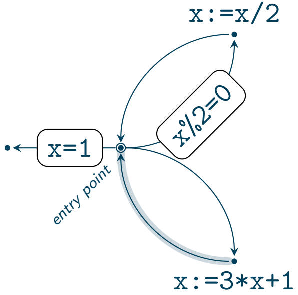
of the Hasse-Syracuse algorithm



the current value of x is 52

An execution trace on a control flow graph

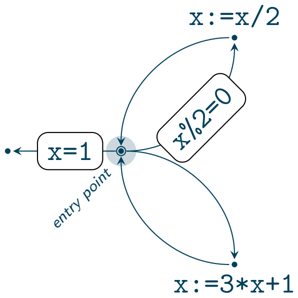
of the Hasse-Syracuse algorithm



the current value of x is 52

An execution trace on a control flow graph

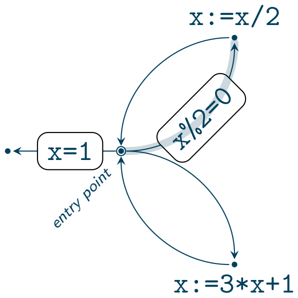
of the Hasse-Syracuse algorithm



the current value of `x` is 52

An execution trace on a control flow graph

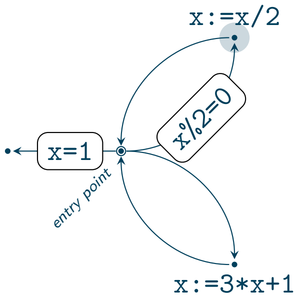
of the Hasse-Syracuse algorithm



the current value of `x` is 52

An execution trace on a control flow graph

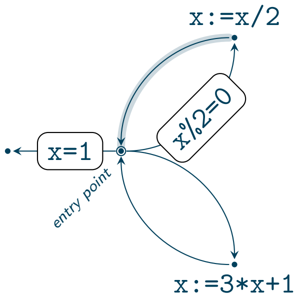
of the Hasse-Syracuse algorithm



the current value of x is 26

An execution trace on a control flow graph

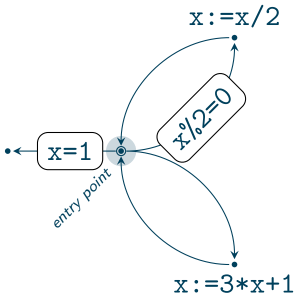
of the Hasse-Syracuse algorithm



the current value of x is 26

An execution trace on a control flow graph

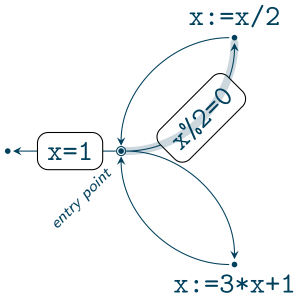
of the Hasse-Syracuse algorithm



the current value of `x` is 26

An execution trace on a control flow graph

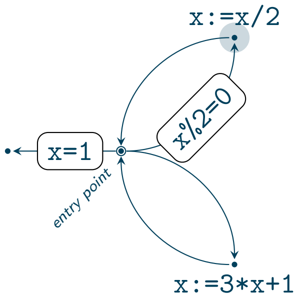
of the Hasse-Syracuse algorithm



the current value of x is 26

An execution trace on a control flow graph

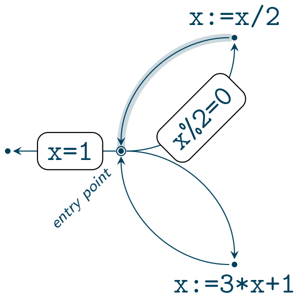
of the Hasse-Syracuse algorithm



the current value of x is 13

An execution trace on a control flow graph

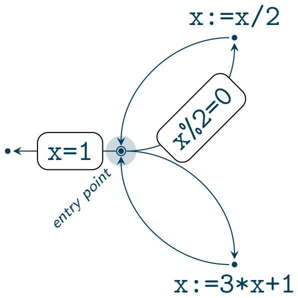
of the Hasse-Syracuse algorithm



the current value of x is 13

An execution trace on a control flow graph

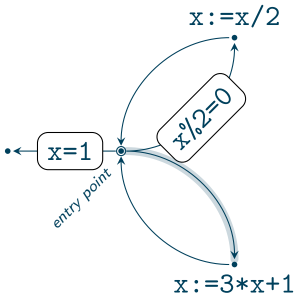
of the Hasse-Syracuse algorithm



the current value of x is 13

An execution trace on a control flow graph

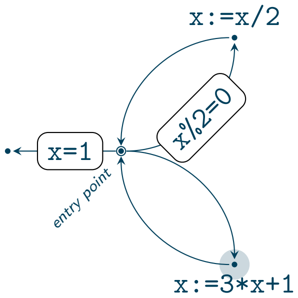
of the Hasse-Syracuse algorithm



the current value of x is 13

An execution trace on a control flow graph

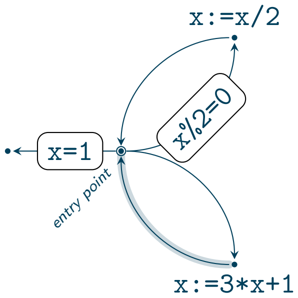
of the Hasse-Syracuse algorithm



the current value of x is 40

An execution trace on a control flow graph

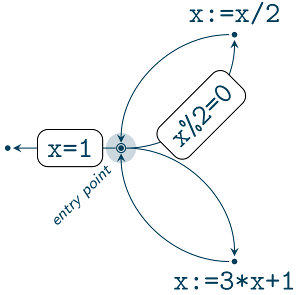
of the Hasse-Syracuse algorithm



the current value of `x` is 40

An execution trace on a control flow graph

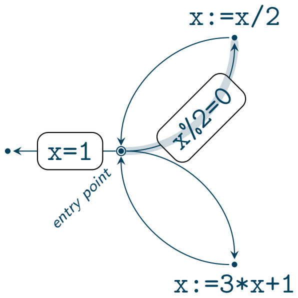
of the Hasse-Syracuse algorithm



the current value of `x` is 40

An execution trace on a control flow graph

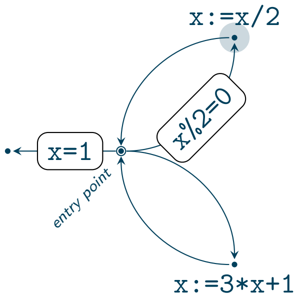
of the Hasse-Syracuse algorithm



the current value of `x` is 40

An execution trace on a control flow graph

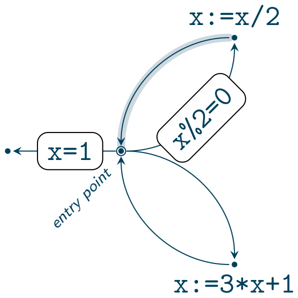
of the Hasse-Syracuse algorithm



the current value of `x` is 20

An execution trace on a control flow graph

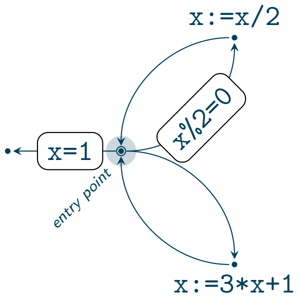
of the Hasse-Syracuse algorithm



the current value of x is 20

An execution trace on a control flow graph

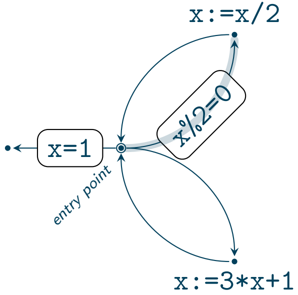
of the Hasse-Syracuse algorithm



the current value of x is 20

An execution trace on a control flow graph

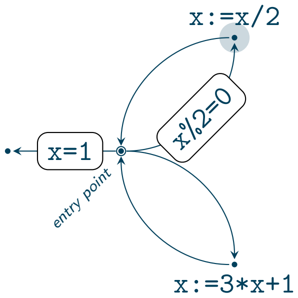
of the Hasse-Syracuse algorithm



the current value of x is 20

An execution trace on a control flow graph

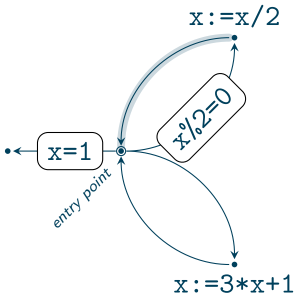
of the Hasse-Syracuse algorithm



the current value of x is 10

An execution trace on a control flow graph

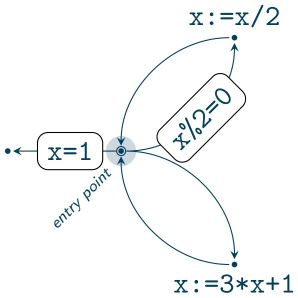
of the Hasse-Syracuse algorithm



the current value of `x` is 10

An execution trace on a control flow graph

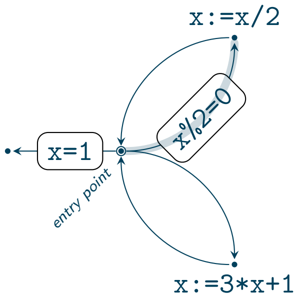
of the Hasse-Syracuse algorithm



the current value of `x` is 10

An execution trace on a control flow graph

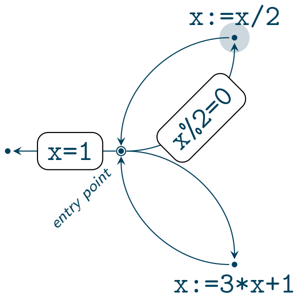
of the Hasse-Syracuse algorithm



the current value of x is 10

An execution trace on a control flow graph

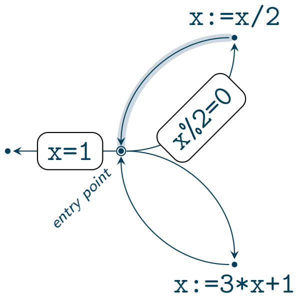
of the Hasse-Syracuse algorithm



the current value of `x` is 5

An execution trace on a control flow graph

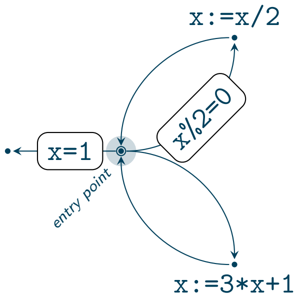
of the Hasse-Syracuse algorithm



the current value of `x` is 5

An execution trace on a control flow graph

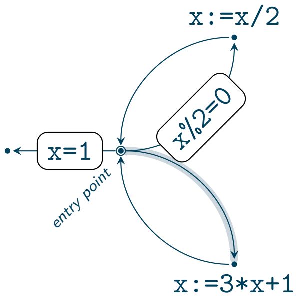
of the Hasse-Syracuse algorithm



the current value of `x` is 5

An execution trace on a control flow graph

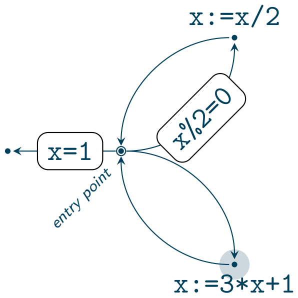
of the Hasse-Syracuse algorithm



the current value of `x` is 5

An execution trace on a control flow graph

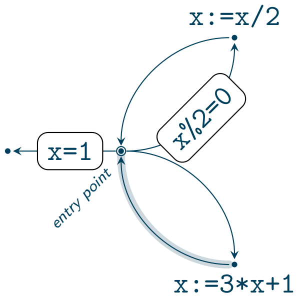
of the Hasse-Syracuse algorithm



the current value of `x` is 16

An execution trace on a control flow graph

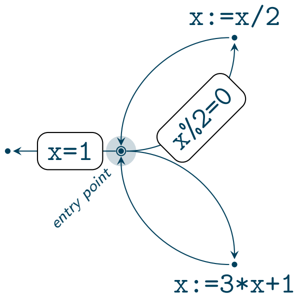
of the Hasse-Syracuse algorithm



the current value of `x` is 16

An execution trace on a control flow graph

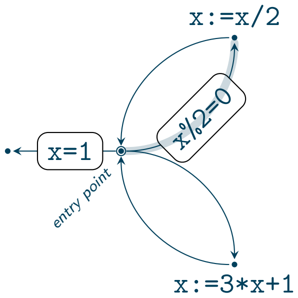
of the Hasse-Syracuse algorithm



the current value of x is 16

An execution trace on a control flow graph

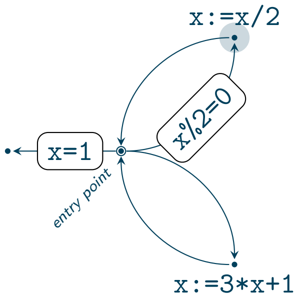
of the Hasse-Syracuse algorithm



the current value of `x` is 16

An execution trace on a control flow graph

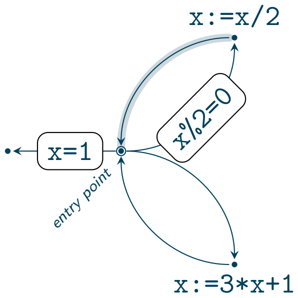
of the Hasse-Syracuse algorithm



the current value of `x` is 8

An execution trace on a control flow graph

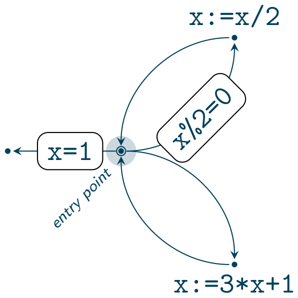
of the Hasse-Syracuse algorithm



the current value of `x` is 8

An execution trace on a control flow graph

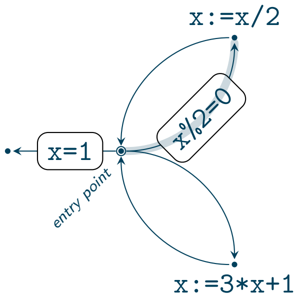
of the Hasse-Syracuse algorithm



the current value of x is 8

An execution trace on a control flow graph

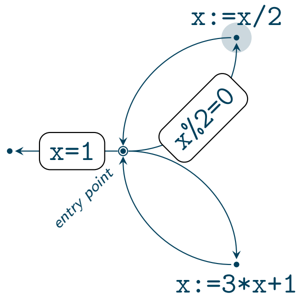
of the Hasse-Syracuse algorithm



the current value of `x` is 8

An execution trace on a control flow graph

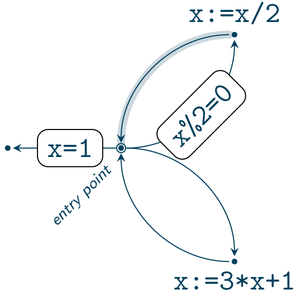
of the Hasse-Syracuse algorithm



the current value of x is 4

An execution trace on a control flow graph

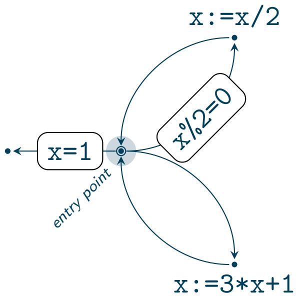
of the Hasse-Syracuse algorithm



the current value of `x` is 4

An execution trace on a control flow graph

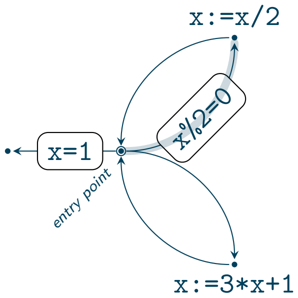
of the Hasse-Syracuse algorithm



the current value of x is 4

An execution trace on a control flow graph

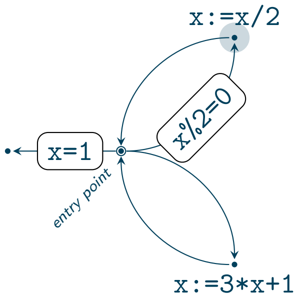
of the Hasse-Syracuse algorithm



the current value of `x` is 4

An execution trace on a control flow graph

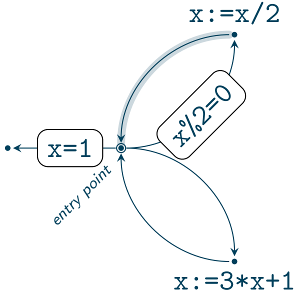
of the Hasse-Syracuse algorithm



the current value of `x` is 2

An execution trace on a control flow graph

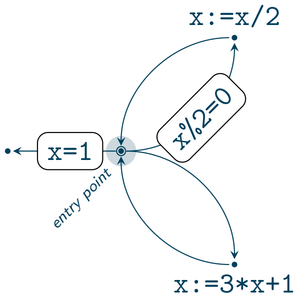
of the Hasse-Syracuse algorithm



the current value of `x` is 2

An execution trace on a control flow graph

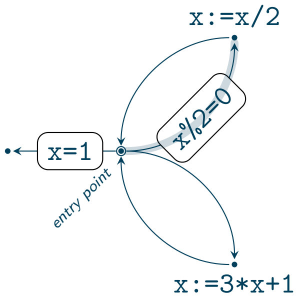
of the Hasse-Syracuse algorithm



the current value of x is 2

An execution trace on a control flow graph

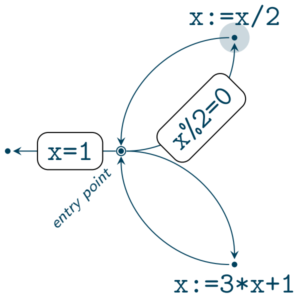
of the Hasse-Syracuse algorithm



the current value of `x` is 2

An execution trace on a control flow graph

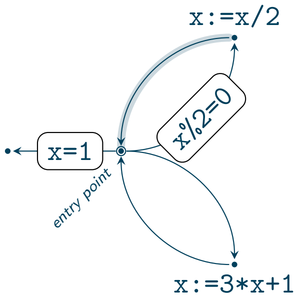
of the Hasse-Syracuse algorithm



the current value of `x` is 1

An execution trace on a control flow graph

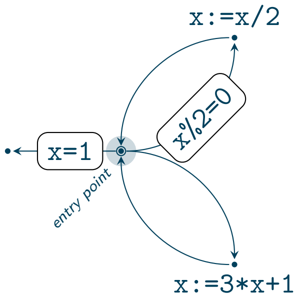
of the Hasse-Syracuse algorithm



the current value of x is 1

An execution trace on a control flow graph

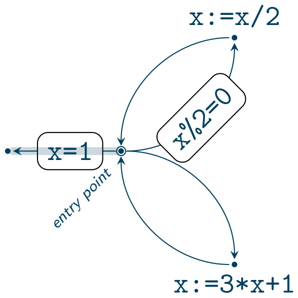
of the Hasse-Syracuse algorithm



the current value of `x` is 1

An execution trace on a control flow graph

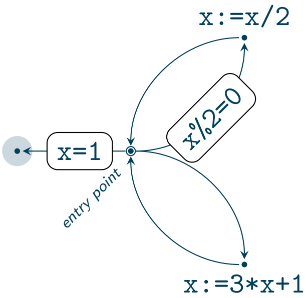
of the Hasse-Syracuse algorithm



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

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

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

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

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
- 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 $\nu : \mathcal{X} \to \mathbb{R}_\perp = \mathbb{R} \cup \{\perp\}$.

Abstract expressions

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

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

- $[\![x]\!]_\nu = \nu(x)$ for all $x \in \mathcal{X}$
- Any value in $\mathbb{R} \setminus \{0\}$ stands for true while 0 stands for false
- $[\![\neg]\!] : \mathbb{R}_\perp \to \mathbb{R}_\perp$,
  $[\![\neg]\!](0) = 1$,
  $[\![\neg]\!](\perp) = \perp$, and

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
- $[\![\neg]\!] : \mathbb{R}_\bot \to \mathbb{R}_\bot$,
  $[\![\neg]\!](0) = 1$,
  $[\![\neg]\!](\bot) = \bot$, and
  $[\![\neg]\!](x) = 0$ for all $x \in \mathbb{R} \setminus \{0\}$

Interpretation of expressions

only depends on the current memory state

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

Abstract instructions

Abstract instructions

The sets of semaphores, and barriers of a program are respectively $\mathcal{S}$ and $\mathcal{B}$.

Abstract instructions

The sets of semaphores, and barriers of a program are respectively $\mathcal{S}$ and $\mathcal{B}$.

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

Abstract instructions

The sets of semaphores, and barriers of a program are respectively $\mathcal{S}$ and $\mathcal{B}$.

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

$$G : A \underset{\partial^+}{\overset{\partial^-}{\rightrightarrows}} V$$

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

$$\beta : \{\text{valuations}\} \to \{a \in A \mid \partial^- a = v\}$$

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

Abstract instructions

The sets of semaphores, and barriers of a program are respectively $\mathcal{S}$ and $\mathcal{B}$.

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

$$G : A \underset{\partial^+}{\overset{\partial^-}{\rightrightarrows}} V$$

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

$$\beta : \{\text{valuations}\} \to \{a \in A \mid \partial^- a = v\}$$

  together with a subset $\mathcal{F}(\beta) \subseteq \mathcal{X}$ such that if the valuations ν and ν' match on $\mathcal{F}(\beta)$ then $\beta(\nu) = \beta(\nu')$.
- The synchronisation primitives $P(s)$, $V(s)$, and $W(b)$ for $s \in \mathcal{S}$ and $b \in \mathcal{B}$

Abstract processes as control flow graphs

Abstract processes as control flow graphs

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

Abstract processes as control flow graphs

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

- An entry point $v_0 \in V$ such that $\lambda(v_0) = \mathit{Skip}$.

Abstract processes as control flow graphs

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

- An entry point $v_0 \in V$ such that $\lambda(v_0) = \mathit{Skip}$.
- If $\lambda(v) \neq \mathit{Skip}$, then v has at least one outgoing arrow.

Abstract processes as control flow graphs

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

- An entry point $v_0 \in V$ such that $\lambda(v_0) = \mathit{Skip}$.
- If $\lambda(v) \neq \mathit{Skip}$, 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

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

- An entry point $v_0 \in V$ such that $\lambda(v_0) = \mathit{Skip}$.
- If $\lambda(v) \neq \mathit{Skip}$, 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 $\nu : \mathcal{X} \to \mathbb{R}$ 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 $\alpha : \mathcal{S} \sqcup \mathcal{B} \to \mathbb{N} \cup \{\infty\}$.

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

The internal states of the abstract machine

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

The internal states of the abstract machine

A state is a mapping σ 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 σ 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
- for all $s \in \mathcal{S}$, $\sigma(s)$ is a multiset over $\{1, \ldots, n\}$.

Admissible multi-instructions

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 μ is said to be admissible at state σ 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 μ is said to be admissible at state σ when:

- for $i, j \in \mathrm{dom}(\mu)$ 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 μ is said to be admissible at state σ when:

- for $i, j \in \text{dom}(\mu)$ with $i \neq j$, $\mu(i)$ and $\mu(j)$ do not conflict,
- for all $s \in \mathcal{S}$, $0 \leqslant \phi(s) \leqslant \alpha(s)$ where

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

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 μ is said to be admissible at state σ when:

- for $i, j \in \mathrm{dom}(\mu)$ with $i \neq j$, $\mu(i)$ and $\mu(j)$ do not conflict,
- for all $s \in \mathcal{S}$, $0 \leqslant \phi(s) \leqslant \alpha(s)$ where

$$\begin{array}{rcl}\phi(s) & = & |\sigma(s)| \\ & & + \ \mathrm{card}\{\mathrm{i} \in \mathrm{dom}(\mu) \mid \mu(\mathrm{i}) = \mathrm{P(s)}\} \\ & & - \ \mathrm{card}\{\mathrm{i} \in \mathrm{dom}(\mu) \mid \mu(\mathrm{i}) = \mathrm{V(s)}\}\end{array}$$

- for all $b \in \mathcal{B}$, $\mathrm{card}\{\mathrm{i} \in \mathrm{dom}(\mu) \mid \mu(\mathrm{i}) = \mathrm{W(b)}\} \notin \{1, \ldots, \alpha(\mathrm{b})\}$

Action of a multi-instruction on a state

Assuming that μ is admissible at σ

Action of a multi-instruction on a state

Assuming that μ is admissible at σ

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

Action of a multi-instruction on a state

Assuming that μ is admissible at σ

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

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

Action of a multi-instruction on a state

Assuming that μ is admissible at σ

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

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

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

Action of a multi-instruction on a state

Assuming that μ is admissible at σ

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

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

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

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

Action of a multi-instruction on a state

Assuming that μ is admissible at σ

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

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

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

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

A sequence $\mu_0, \ldots, \mu_{q-1}$ of multi-intructions is said to be admissible at state σ when for all $k \in \{0, \ldots, q-1\}$ the multi-instruction μ_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 γ 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 γ 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 γ 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 γ 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 γ 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 γ 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 γ 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

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

Directed paths and sequences of multi-instructions

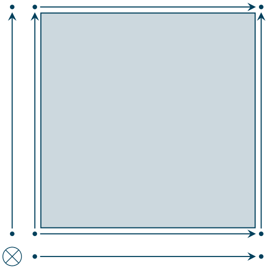
A directed path γ on $(G_1, \ldots, G_n)$ is a sequence $(\gamma(k))_{k \in \{0,\ldots,q\}}$ of points such that for all $k \in \{0, \ldots, q-1\}$ we have

- $\gamma_i(k) = \gamma_i(k+1)$ 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 $i \in \{1, \ldots, n\}$.

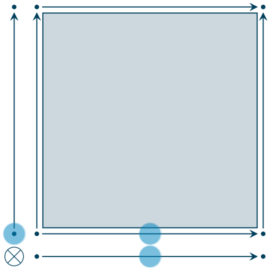
Then γ is associated with a sequence of multi-instructions $(\mu_k)_{k \in \{0,\ldots,q-1\}}$ defined for $k \in \{0, \ldots, q-1\}$ by

- $\mathrm{dom}(\mu_k) = \{i \in \{1, \ldots, n\} \mid \gamma_i(k+1) = \partial^+ \gamma_i(k) \text{ or } \lambda_i(\gamma_i(k+1)) = W(\_)\}$
- $\mu_k(i) = \lambda_i(\gamma_i(k+1))$ for all $k \in \{0, \ldots, q-1\}$ and all $i \in \mathrm{dom}(\mu_k)$

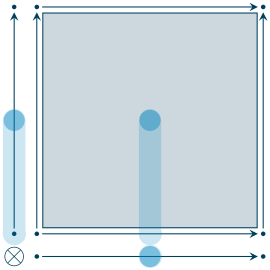
Discrete paths are “continuous”



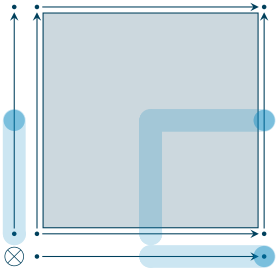
Discrete paths are “continuous”



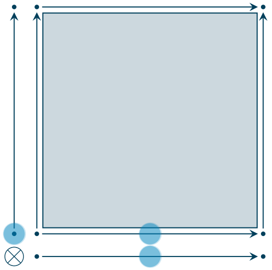
Discrete paths are “continuous”



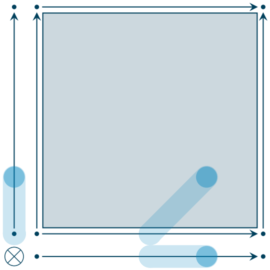
Discrete paths are "continuous"



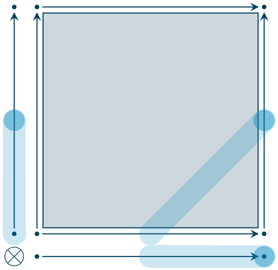
Discrete paths are "continuous"



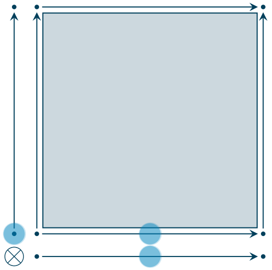
Discrete paths are "continuous"



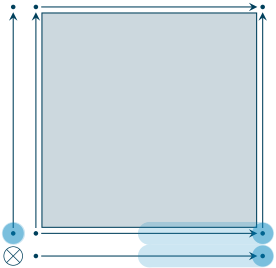
Discrete paths are "continuous"



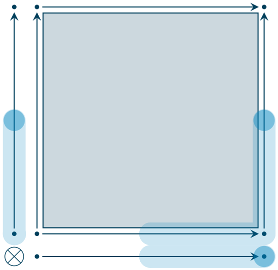
Discrete paths are “continuous”



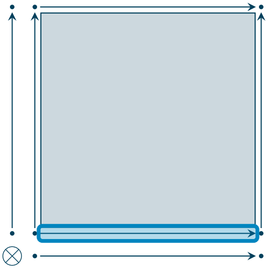
Discrete paths are “continuous”



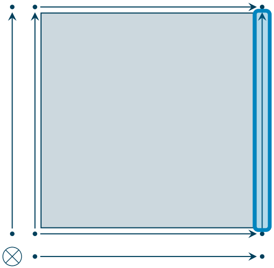
Discrete paths are "continuous"



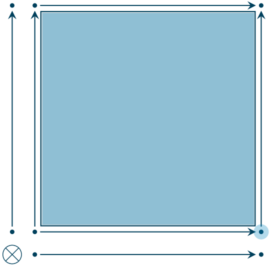
Discrete paths are “continuous”



Discrete paths are “continuous”



Discrete paths are “continuous”



Admissible paths and execution traces

Admissible paths and execution traces

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

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

Admissible paths and execution traces

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

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

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

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

Concurrent access

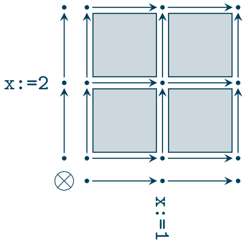
```
var x = 0
```

```
proc p = x:=1

proc q = x:=2
```

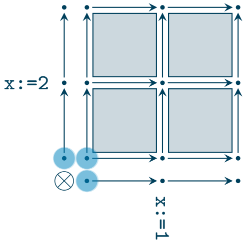
```
init p q
```

Admissible execution trace



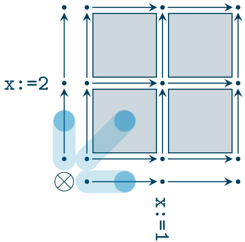
the value of x is 0

Admissible execution trace



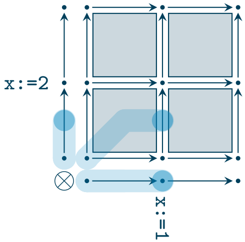
the value of x is 0

Admissible execution trace



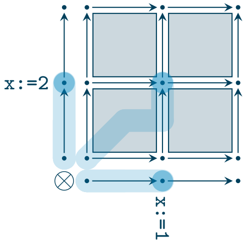
the value of x is 0

Admissible execution trace



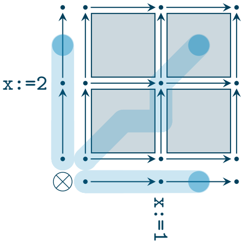
the value of x is 1

Admissible execution trace



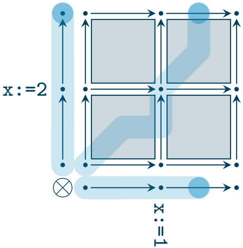
the value of x is 2

Admissible execution trace



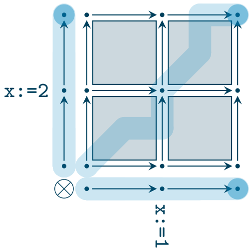
the value of x is 2

Admissible execution trace



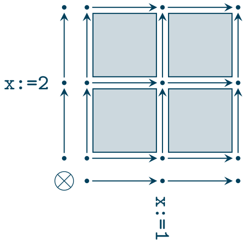
the value of x is 2

Admissible execution trace



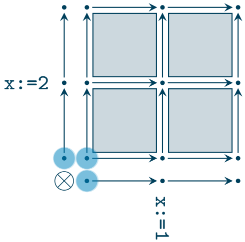
the value of x is 2

Not admissible execution trace



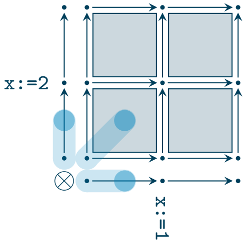
the value of x is 0

Not admissible execution trace



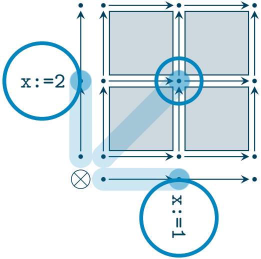
the value of `x` is 0

Not admissible execution trace



the value of x is 0

Not admissible execution trace



the value of x is ?

Lack of resources

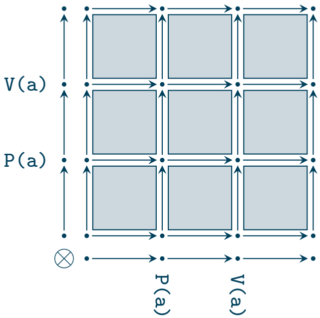
```
sem 1 a
```

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

```
init 2p
```

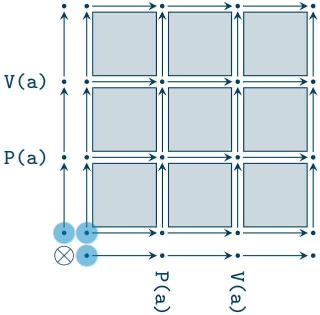
Admissible concurrent execution trace

`sem 1 a`



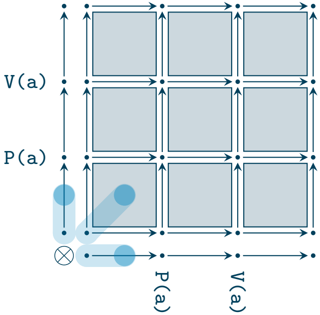
Admissible concurrent execution trace

`sem 1 a`



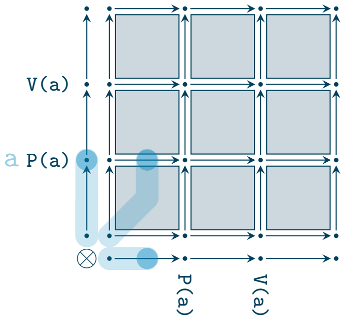
Admissible concurrent execution trace

`sem 1 a`



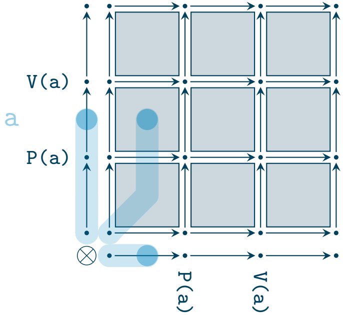
Admissible concurrent execution trace

`sem 1 a`



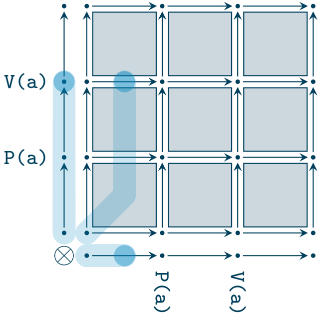
Admissible concurrent execution trace

`sem 1 a`



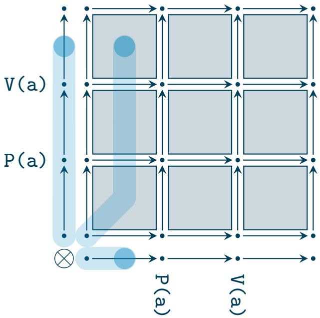
Admissible concurrent execution trace

`sem 1 a`



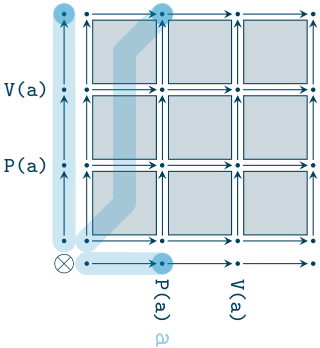
Admissible concurrent execution trace

`sem 1 a`



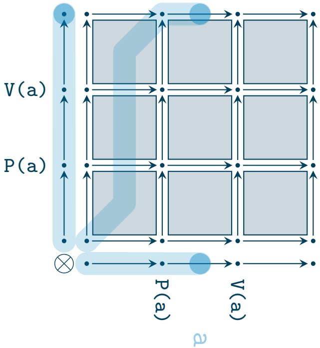
Admissible concurrent execution trace

`sem 1 a`



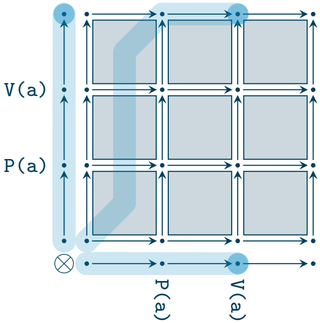
Admissible concurrent execution trace

`sem 1 a`



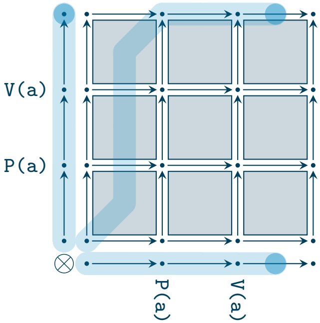
Admissible concurrent execution trace

`sem 1 a`



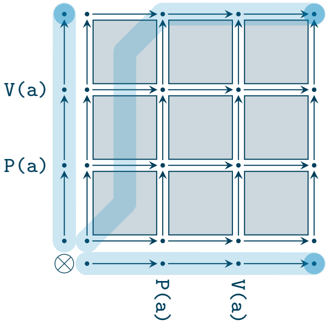
Admissible concurrent execution trace

`sem 1 a`



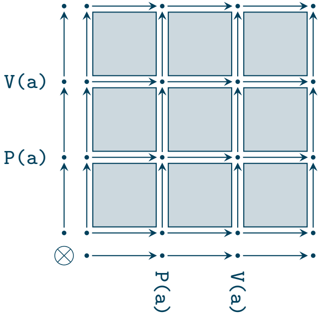
Admissible concurrent execution trace

`sem 1 a`



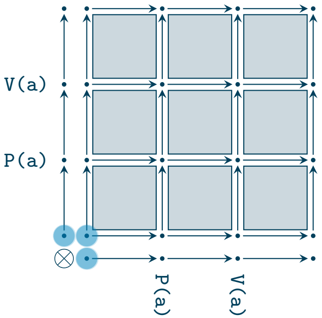
Not admissible concurrent execution trace

`sem 1 a`



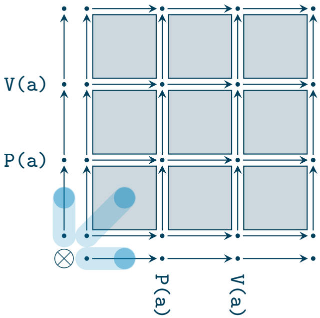
Not admissible concurrent execution trace

`sem 1 a`



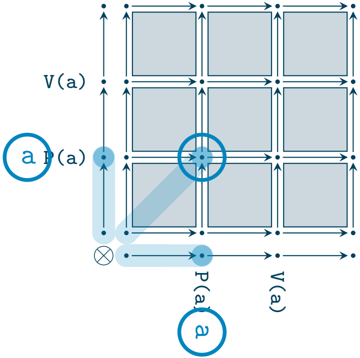
Not admissible concurrent execution trace

`sem 1 a`



Not admissible concurrent execution trace

`sem 1 a`



Synchronisation

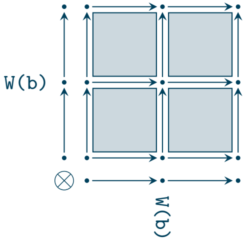
```
sync 1 b
```

```
proc p = W(b)
```

```
init 2p
```

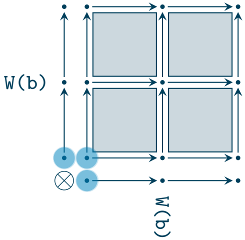
Concurrent execution trace

`sync 1 b`



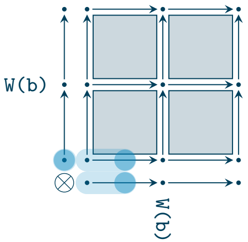
Concurrent execution trace

`sync 1 b`



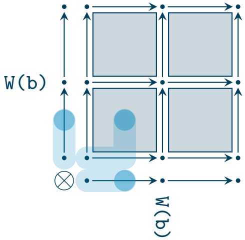
Concurrent execution trace

`sync 1 b`



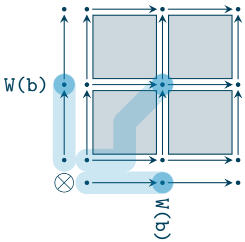
Concurrent execution trace

`sync 1 b`



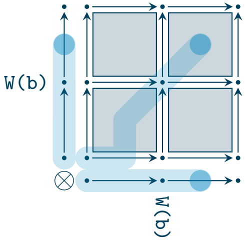
Concurrent execution trace

`sync 1 b`



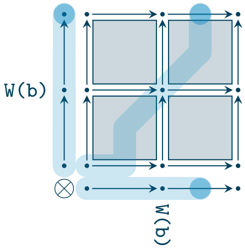
Concurrent execution trace

`sync 1 b`



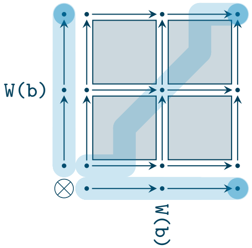
Concurrent execution trace

`sync 1 b`



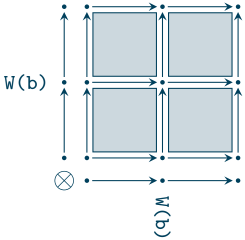
Concurrent execution trace

`sync 1 b`



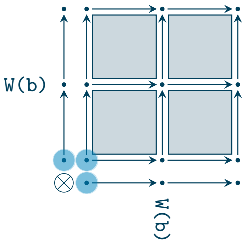
Not admissible concurrent execution trace

`sync 1 b`



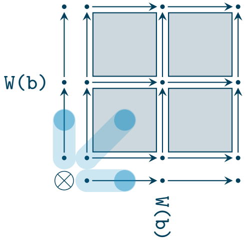
Not admissible concurrent execution trace

`sync 1 b`



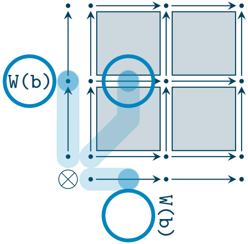
Not admissible concurrent execution trace

`sync 1 b`



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

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 σ, for all directed paths γ, γ' starting at the origin,

$$\partial^{+}\gamma = \partial^{+}\gamma' \quad \Rightarrow \quad \sigma \cdot \gamma|_{\mathcal{S}} = \sigma \cdot \gamma'|_{\mathcal{S}}$$

In particular, the program Π comes with a potential function

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

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 σ, for all directed paths γ, γ' starting at the origin,

$$\partial^+\gamma = \partial^+\gamma' \quad \Rightarrow \quad \sigma \cdot \gamma|_{\mathcal{S}} = \sigma \cdot \gamma'|_{\mathcal{S}}$$

In particular, the program Π comes with a potential function

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

Proposition: The program Π 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 σ, for all directed paths γ, γ' starting at the origin,

$$\partial^+\gamma = \partial^+\gamma' \quad \Rightarrow \quad \sigma \cdot \gamma|_{\mathcal{S}} = \sigma \cdot \gamma'|_{\mathcal{S}}$$

In particular, the program Π comes with a potential function

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

Proposition: The program Π 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 σ, for all directed paths γ, γ' starting at the origin,

$$\partial^+\gamma = \partial^+\gamma' \quad \Rightarrow \quad \sigma \cdot \gamma|_{\mathcal{S}} = \sigma \cdot \gamma'|_{\mathcal{S}}$$

In particular, the program Π comes with a potential function

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

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

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

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 σ, for all directed paths γ, γ' starting at the origin,

$$\partial^+\gamma = \partial^+\gamma' \quad \Rightarrow \quad \sigma \cdot \gamma|_{\mathcal{S}} = \sigma \cdot \gamma'|_{\mathcal{S}}$$

In particular, the program Π comes with a potential function

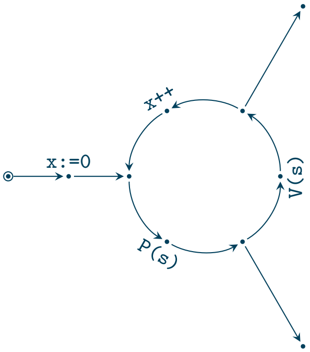
$$F_\Pi \;:\; \{\text{semaphores}\} \times \{\text{points}\} \to \mathbb{N} \quad \cong \quad \{\text{points}\} \to \{\text{multisets over } \mathcal{S}\}$$

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

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

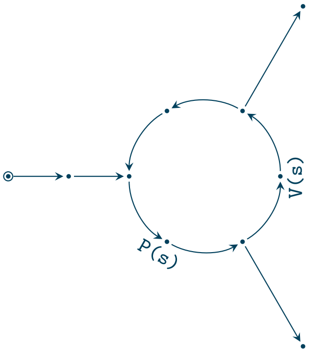
Conservative process

example



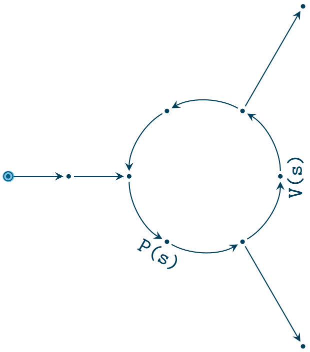
Conservative process

example



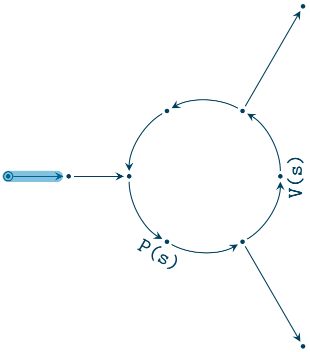
Conservative process

example



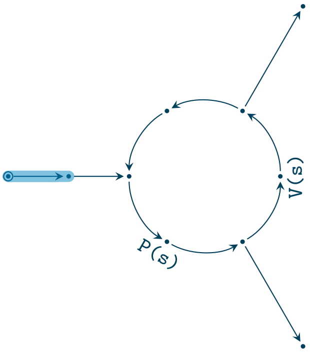
Conservative process

example



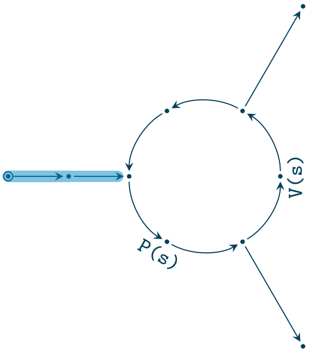
Conservative process

example



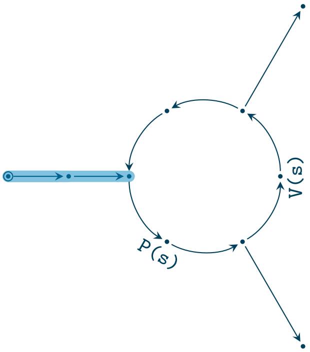
Conservative process

example



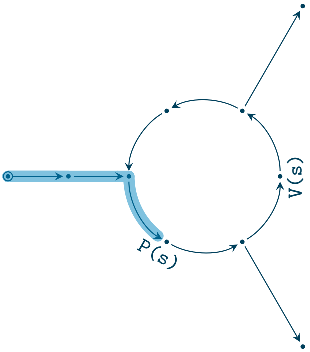
Conservative process

example



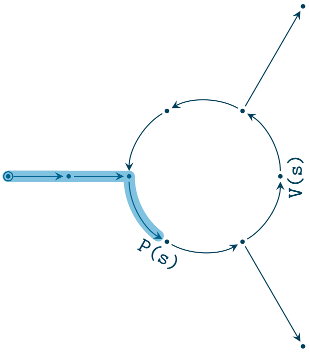
Conservative process

example



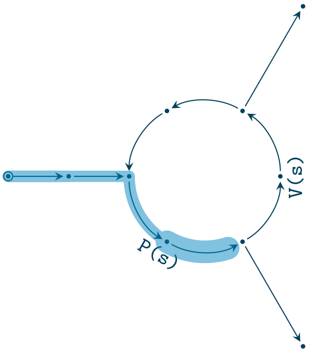
Conservative process

example



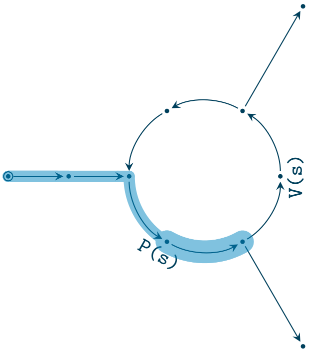
Conservative process

example



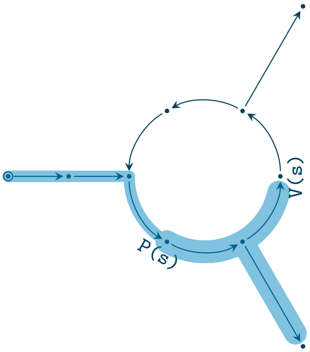
Conservative process

example



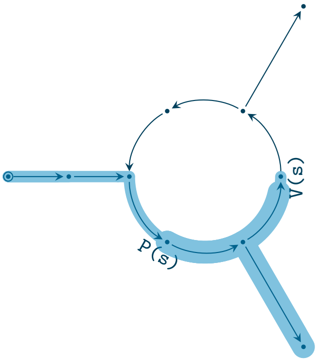
Conservative process

example



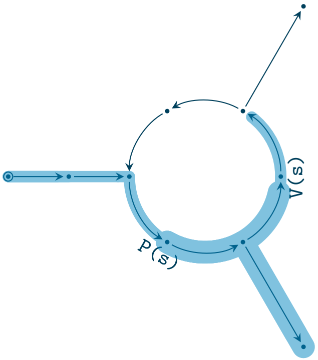
Conservative process

example



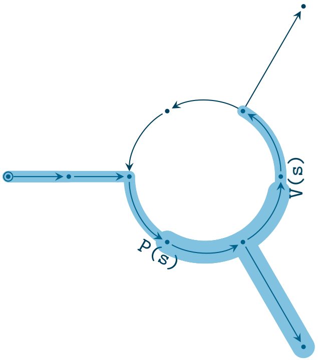
Conservative process

example



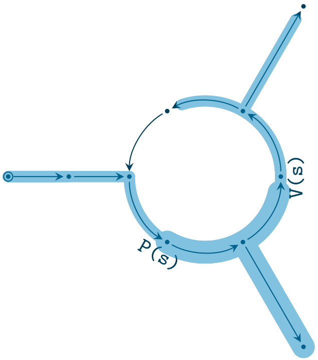
Conservative process

example



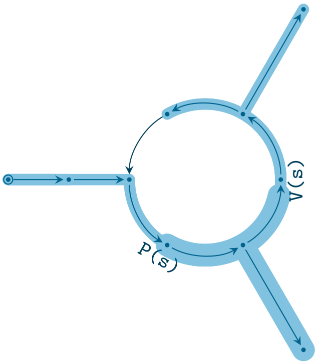
Conservative process

example



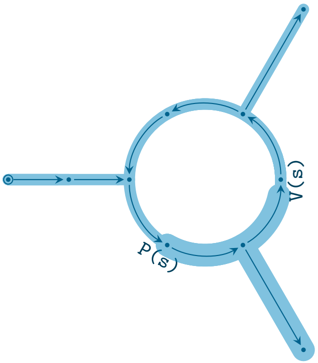
Conservative process

example



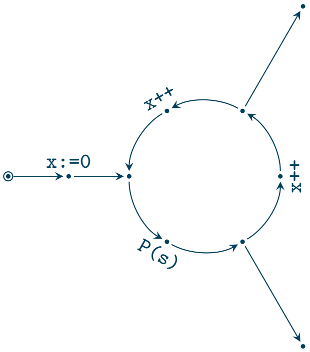
Conservative process

example



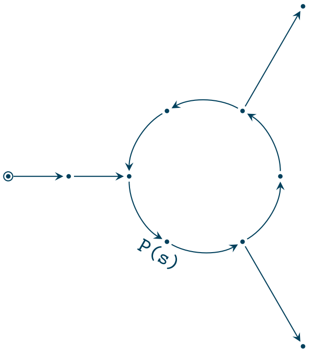
Not conservative process

example



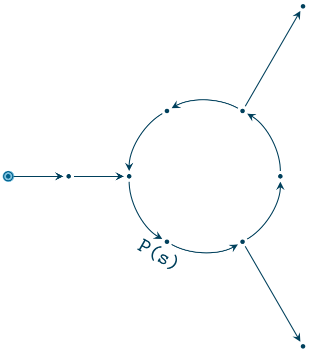
Not conservative process

example



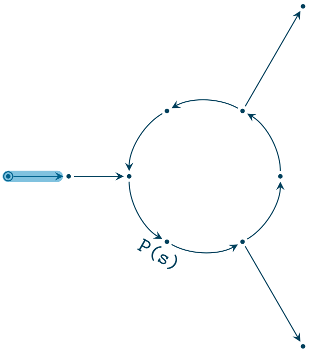
Not conservative process

example



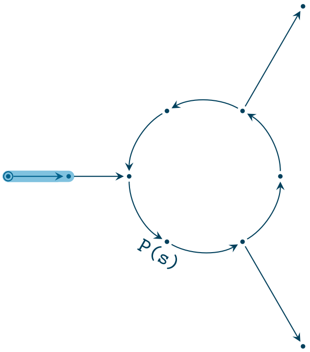
Not conservative process

example



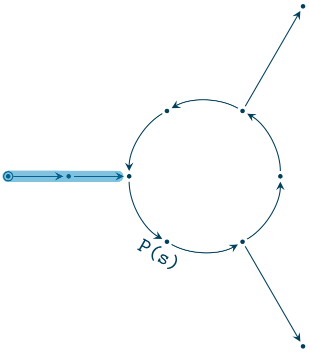
Not conservative process

example



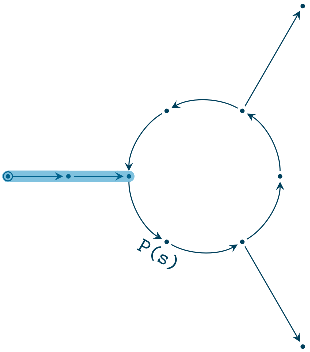
Not conservative process

example



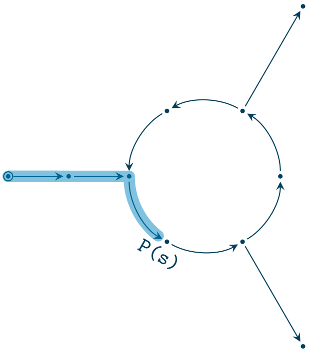
Not conservative process

example



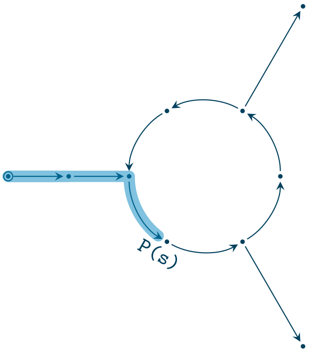
Not conservative process

example



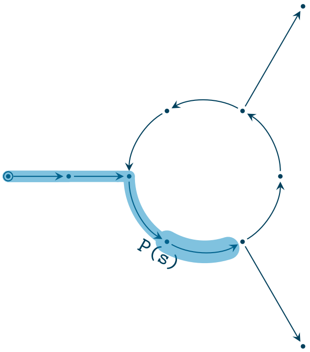
Not conservative process

example



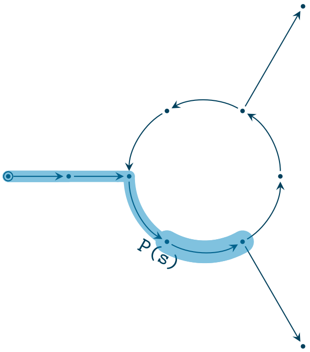
Not conservative process

example



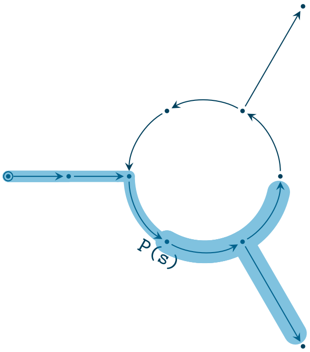
Not conservative process

example



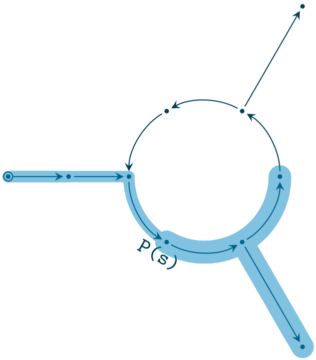
Not conservative process

example



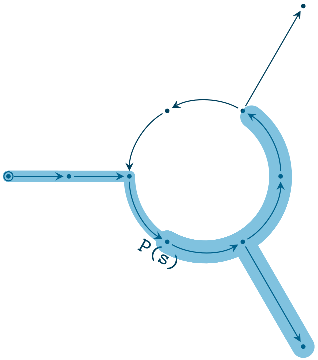
Not conservative process

example



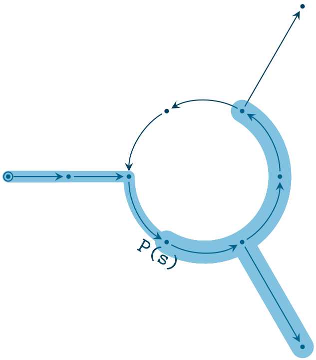
Not conservative process

example



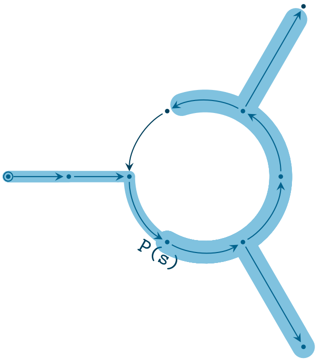
Not conservative process

example



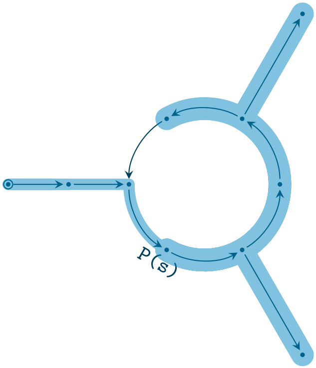
Not conservative process

example



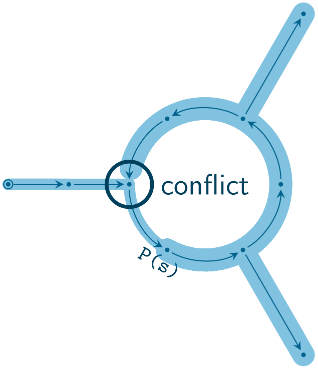
Not conservative process

example



Not conservative process

example



Conservativity is decidable

Conservativity is decidable

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

Conservativity is decidable

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

- The first term π_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 $\pi_n : \{\text{points}\} \to \mathbb{N}^{\mathcal{S}}$.

- The first term π_0 is only defined at the origin and $\pi_0(\text{origin})$ is the empty
- Assuming that π_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 $\pi_n : \{\text{points}\} \to \mathbb{N}^{\mathcal{S}}$.

- The first term π_0 is only defined at the origin and $\pi_0(\text{origin})$ is the empty
- Assuming that π_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 $\pi_n : \{\text{points}\} \to \mathbb{N}^{\mathcal{S}}$.

- The first term π_0 is only defined at the origin and $\pi_0(\text{origin})$ is the empty
- Assuming that π_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 π_n, by setting:

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

Conservativity is decidable

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

- The first term π_0 is only defined at the origin and $\pi_0(\text{origin})$ is the empty
- Assuming that π_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 π_n, by setting:

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

- If all these extensions are compatible, then π_{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 $\pi_n : \{\text{points}\} \to \mathbb{N}^{\mathcal{S}}$.

- The first term π_0 is only defined at the origin and $\pi_0(\text{origin})$ is the empty
- Assuming that π_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 π_n, by setting:

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

- If all these extensions are compatible, then π_{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 π.

Conservativity is decidable

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

- The first term π_0 is only defined at the origin and $\pi_0(\text{origin})$ is the empty
- Assuming that π_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 π_n, by setting:

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

- If all these extensions are compatible, then π_{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

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

  whose last element is denoted by π.
- 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.

$$\pi(p') \quad = \quad \begin{cases} \pi(p) & \text{if } \partial^- p' = p \\ \pi(p) \cdot \lambda(p') & \text{if } p' = \partial^+ p \end{cases}$$

Discrete Models

The discrete model of a conservative program

The discrete model of a conservative program

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

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 $s \in \mathcal{S}$ such that

$$F(p_1, \ldots, p_n, s) \quad > \quad \text{arity}(s) \ ,$$

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 $s \in \mathcal{S}$ such that

$$F(p_1, \ldots, p_n, s) \quad > \quad \mathrm{arity}(s) \ ,$$

- desynchronizing when there is some synchronization barrier $b \in \mathcal{B}$ such that

$$0 \quad < \quad \mathrm{card}\big\{\mathrm{i} \in \{1, \ldots, \mathrm{n}\} \mid \lambda_{\mathrm{i}}(\mathrm{p_i}) = \mathrm{W(b)}\big\} \quad \leqslant \quad \mathrm{arity(b)} \ ,$$

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 $s \in \mathcal{S}$ such that

$$F(p_1, \ldots, p_n, s) \quad > \quad \text{arity}(s) \ ,$$

- desynchronizing when there is some synchronization barrier $b \in \mathcal{B}$ such that

$$0 \quad < \quad \text{card}\{\mathrm{i} \in \{1, \ldots, \mathrm{n}\} \mid \lambda_\mathrm{i}(\mathrm{p_i}) = \mathrm{W}(\mathrm{b})\} \quad \leqslant \quad \text{arity}(\mathrm{b}) \ ,$$

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

$$\{\text{fobidden}\} \quad = \quad \{\text{conflicting}\} \cup \{\text{exhausting}\} \cup \{\text{desynchronizing}\}$$

The discrete model of a conservative program

A point $p = (p_1, \dots, p_n)$ 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 $s \in \mathcal{S}$ such that

$$F(p_1, \dots, p_n, s) \quad > \quad \mathrm{arity}(s) \ ,$$

- desynchronizing when there is some synchronization barrier $b \in \mathcal{B}$ such that

$$0 \quad < \quad \mathrm{card}\{\mathrm{i} \in \{1, \dots, \mathrm{n}\} \mid \lambda_\mathrm{i}(\mathrm{p_i}) = \mathrm{W(b)}\} \quad \leqslant \quad \mathrm{arity(b)} \ ,$$

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

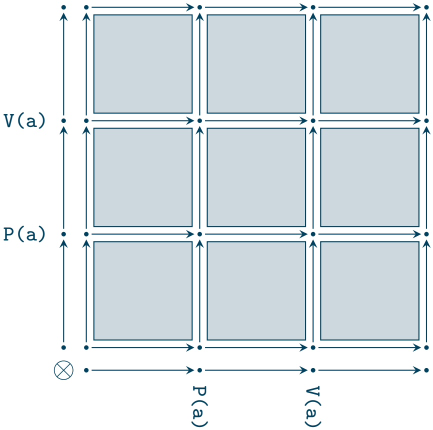
$$\{\text{fobidden}\} \quad = \quad \{\text{conflicting}\} \cup \{\text{exhausting}\} \cup \{\text{desynchronizing}\}$$

The discrete model is the complement of its forbidden set.

$$\{\text{points of the program}\} \setminus \{\text{forbidden points}\}$$

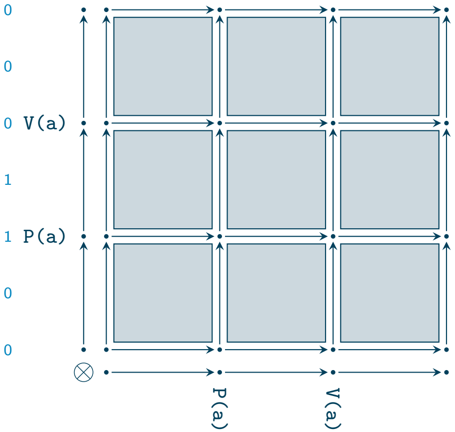
Discrete model

`sem 1 a`



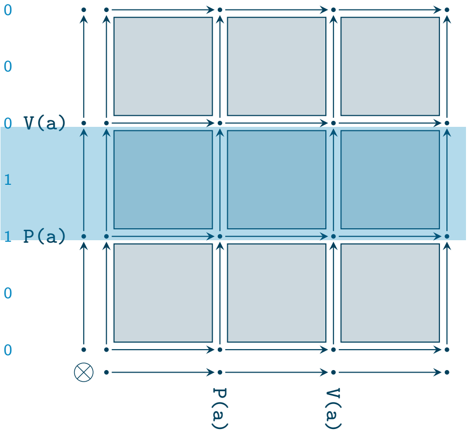
Discrete model

`sem 1 a`



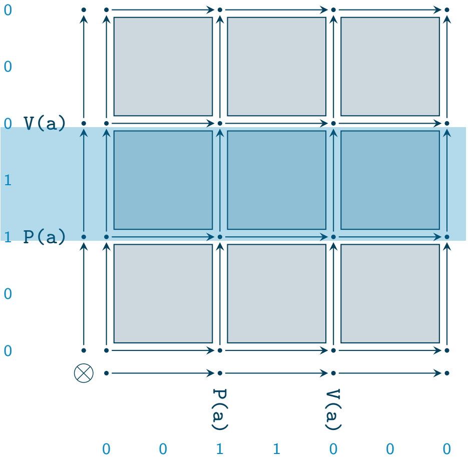
Discrete model

`sem 1 a`



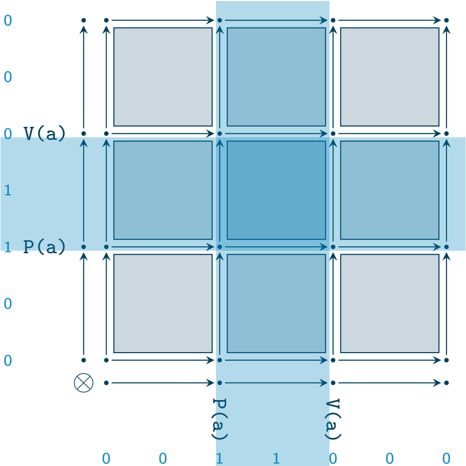
Discrete model

`sem 1 a`



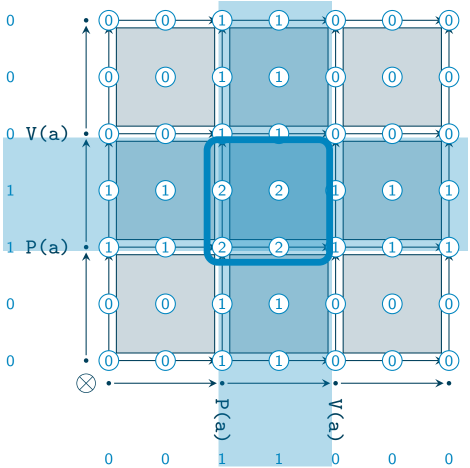
Discrete model

`sem 1 a`



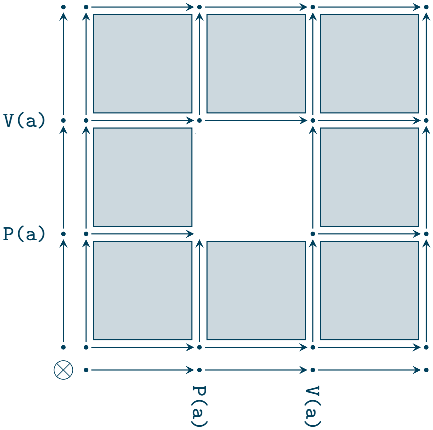
Discrete model

`sem 1 a`



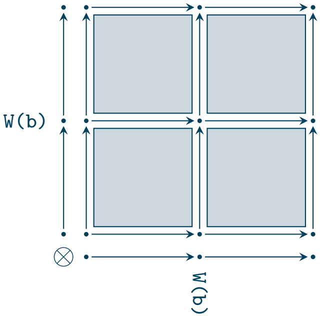
Discrete model

`sem 1 a`



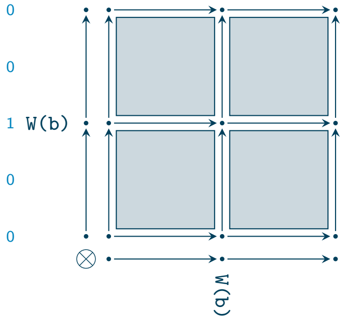
Discrete Model

`sync 1 b`



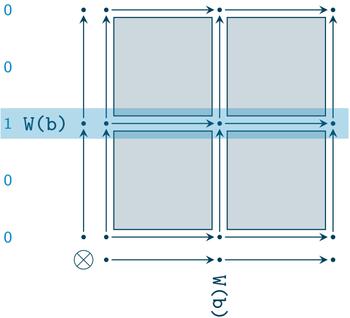
Discrete Model

`sync 1 b`



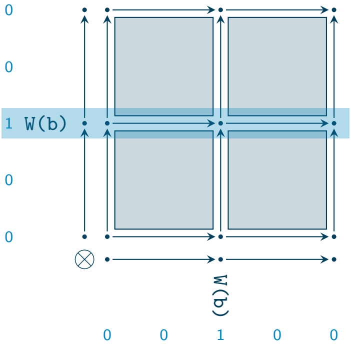
Discrete Model

`sync 1 b`



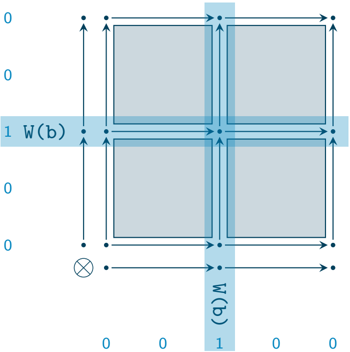
Discrete Model

`sync 1 b`



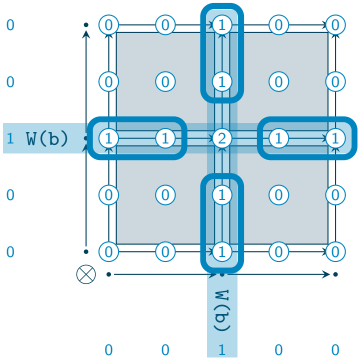
Discrete Model

`sync 1 b`



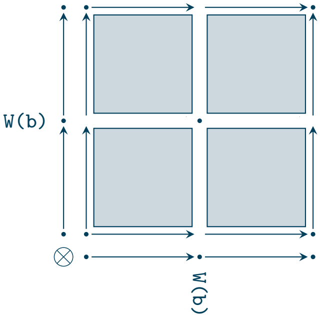
Discrete Model

`sync 1 b`



Discrete Model

`sync 1 b`



Main theorem of discrete models

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

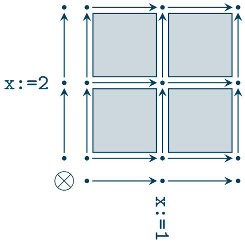
- Soundness: any directed path on a discrete model (i.e. which does not meet any forbidden point) is admissible.
- Completeness: for each admissible path which meets a forbidden point ...

Main theorem of discrete models

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

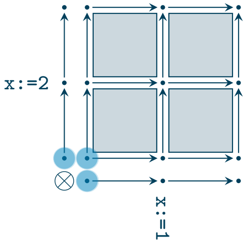
- Completeness: for each admissible path which meets a forbidden point there exists a directed path which avoids them and such that both directed paths induce the same sequence of multi-instructions.

Admissible execution trace



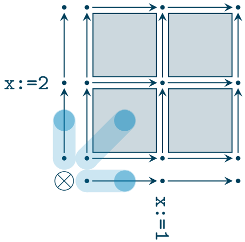
the value of x is 0

Admissible execution trace



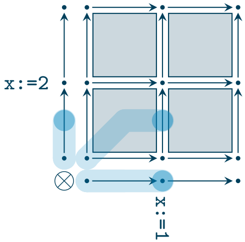
the value of x is 0

Admissible execution trace



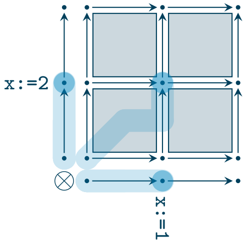
the value of $\mathtt{x}$ is 0

Admissible execution trace



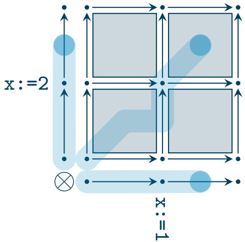
the value of x is 1

Admissible execution trace



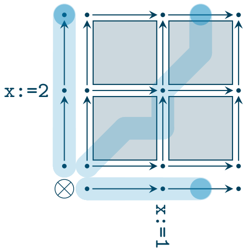
the value of x is 2

Admissible execution trace



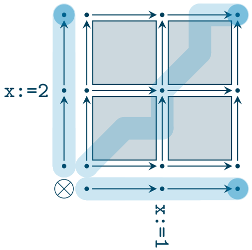
the value of x is 2

Admissible execution trace



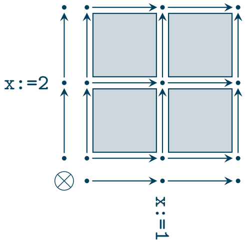
the value of x is 2

Admissible execution trace



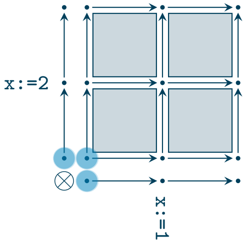
the value of $\mathtt{x}$ is 2

Admissible execution trace avoiding forbidden points



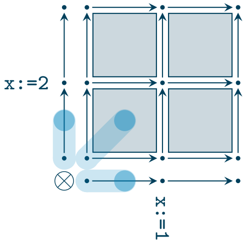
the value of x is 0

Admissible execution trace avoiding forbidden points



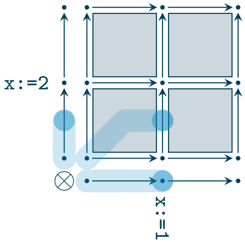
the value of x is 0

Admissible execution trace avoiding forbidden points



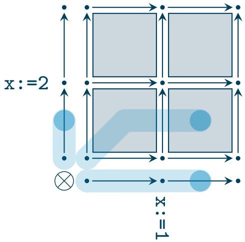
the value of x is 0

Admissible execution trace avoiding forbidden points



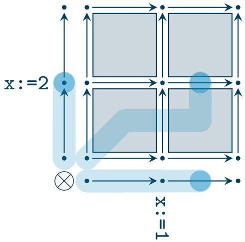
the value of x is 1

Admissible execution trace avoiding forbidden points



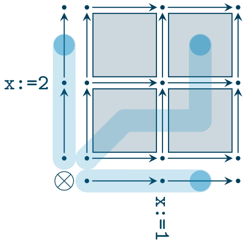
the value of x is 1

Admissible execution trace avoiding forbidden points



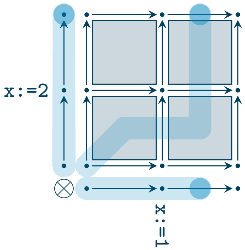
the value of x is 2

Admissible execution trace avoiding forbidden points



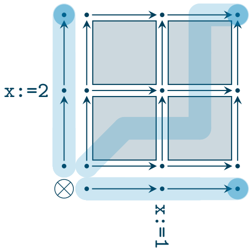
the value of x is 2

Admissible execution trace avoiding forbidden points



the value of x is 2

Admissible execution trace avoiding forbidden points



the value of $\mathtt{x}$ is 2

Replacement