Model Checking
Recursive Programs

Stefan Schwoon

LSV, ENS Cachan & CNRS, INRIA Saclay

MOVEP 2010, Aachen, 28/06/2010

“Causes of infinity” in software

Data: integers, lists, trees and other pointer structures, . . .

Control: Procedures, dynamic process creation, . . .

Asynchronous communication: unbounded FIFO channels

Environment: number of participants in a protocol, . . .

Time: discrete or continuous time

Some of these features (or combinations thereof) make the underlying computation models Turing-powerful.

Topic of the talk: Infinities due to control

Example: Quicksort – Find the bug!

```
void quicksort (int left, int right) {
  int lo,hi,piv;
  if (left >= right) return;
  piv = a[right]; lo = left; hi = right;
  while (lo <= hi) {
    if (a[hi] > piv) {
      hi--;
    } else {
      swap a[lo],a[hi]; lo++;
    }
  }
  quicksort(left,hi); quicksort(lo,right);
}
```

Desirable properties: Correct sorting, termination

Recursive calls use (unbounded) stack!

Contents of the talk

Pushdown systems (PDS) as models of recursive programs

Basic PDS model checking: Reachability, LTL

Overview of extensions: weights / concurrency

$\Rightarrow$ a lot of work in this area in the last 10–15 years by many people!

Pushdown systems as models of recursive programs

Features of sequential programs

Programs (in languages like C, Java) are defined by

- control flow, interacting procedures (calls, parameters, return values)
- global variables accessible to all procedures
- local variables in every procedure

Restrictions/assumptions: finite data types, no concurrency!

Features of sequential programs

The state space is spanned by

- program pointer
- values of global variables
- values of local variables (of current procedure)
- activation records (return adresses, copies of locals)

Actions are independent of activation records.

Pushdown Systems

A pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$ features

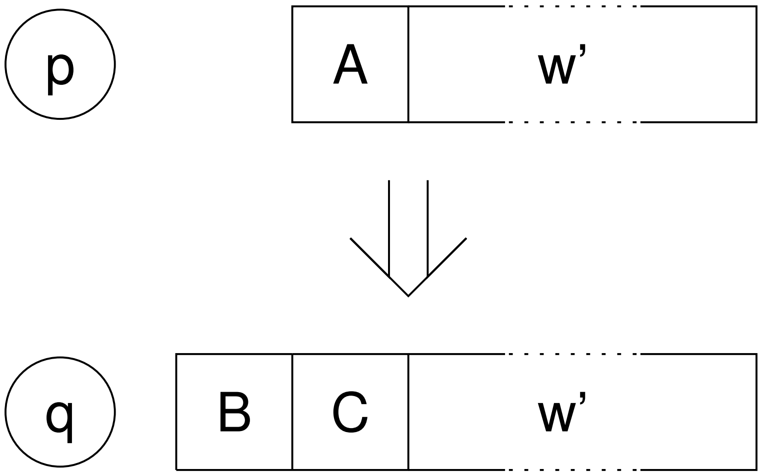
a finite set of control locations P;

a finite stack alphabet Γ;

(a pair $\langle p, w \rangle$, $p \in P$, $w \in \Gamma^*$ is called configuration of $\mathcal{P}$)

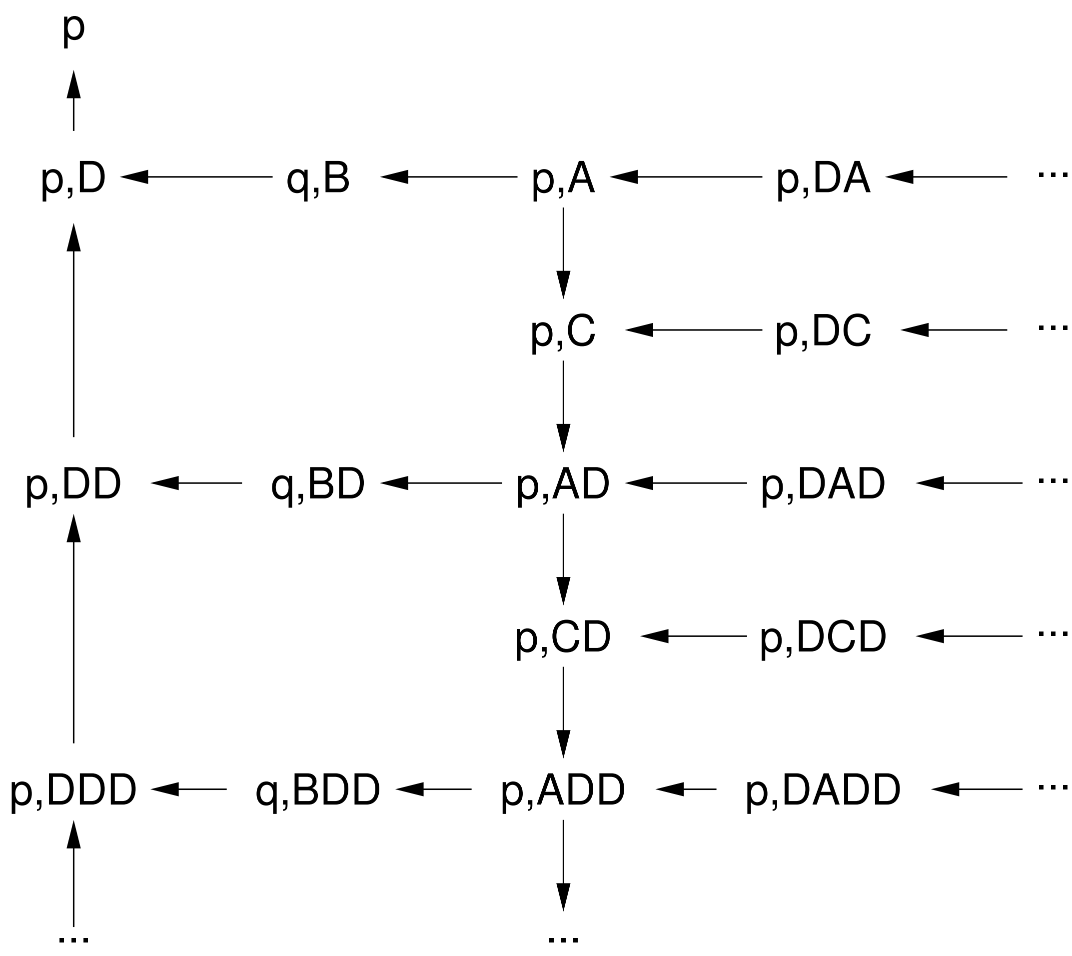
a finite set of rules $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$.

Meaning of a rule $\langle p, A \rangle \hookrightarrow \langle q, BC \rangle$:



Pushdown Graph

$$\langle p, A \rangle \hookrightarrow \langle q, B \rangle$$
$$\langle p, A \rangle \hookrightarrow \langle p, C \rangle$$
$$\langle q, B \rangle \hookrightarrow \langle p, D \rangle$$
$$\langle p, C \rangle \hookrightarrow \langle p, AD \rangle$$
$$\langle p, D \rangle \hookrightarrow \langle p, \varepsilon \rangle$$



Translating programs into pushdown systems

Assignment: `n7: x = x + 1; n8: ...`

$$\langle q, n_7 \rangle \hookrightarrow \langle q, n_8 \rangle$$

Procedure call: `n8: p(); n9: ...`

$$\langle q, n_8 \rangle \hookrightarrow \langle q, p_0\, n_9 \rangle$$

Return: `n5: return;`

$$\langle q, n_5 \rangle \hookrightarrow \langle q, \varepsilon \rangle$$

Encoding data into a PDS

In a configuration $\langle p, A\,w \rangle$, ...

p encodes the values of global variables;

A is a tuple containing the program counter and local variables (of the current procedure);

and w contains the activation records (return addresses, saved local variables).

Basic pushdown model checking

Reachability

A well-known result (Büchi, Caucal):

If the set of configuration C is regular, then so are $pre^*(C)$ and $post^*(C)$.

$$pre^*(C) = \{\, c \mid c' \in C \colon c \Rightarrow^* c' \,\} \qquad \text{(the predecessors of } C\text{)}$$

$$post^*(C) = \{\, c \mid c' \in C \colon c' \Rightarrow^* c \,\} \qquad \text{(the successors of } C\text{)}$$

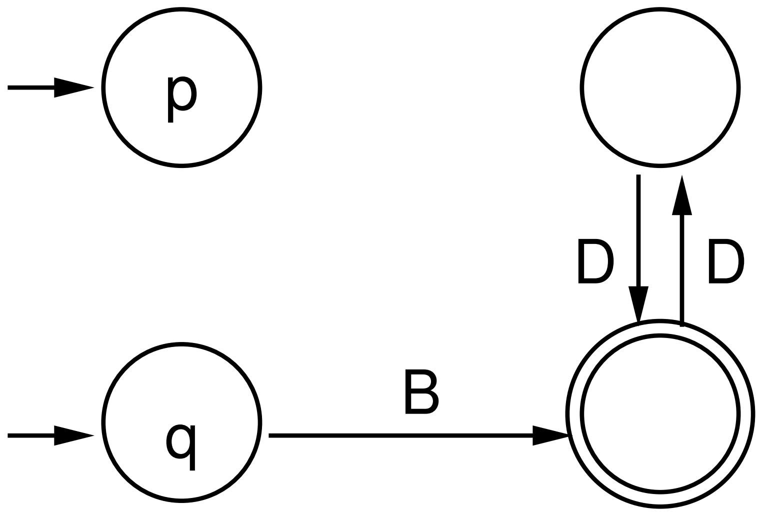
How it works (for pre^*):

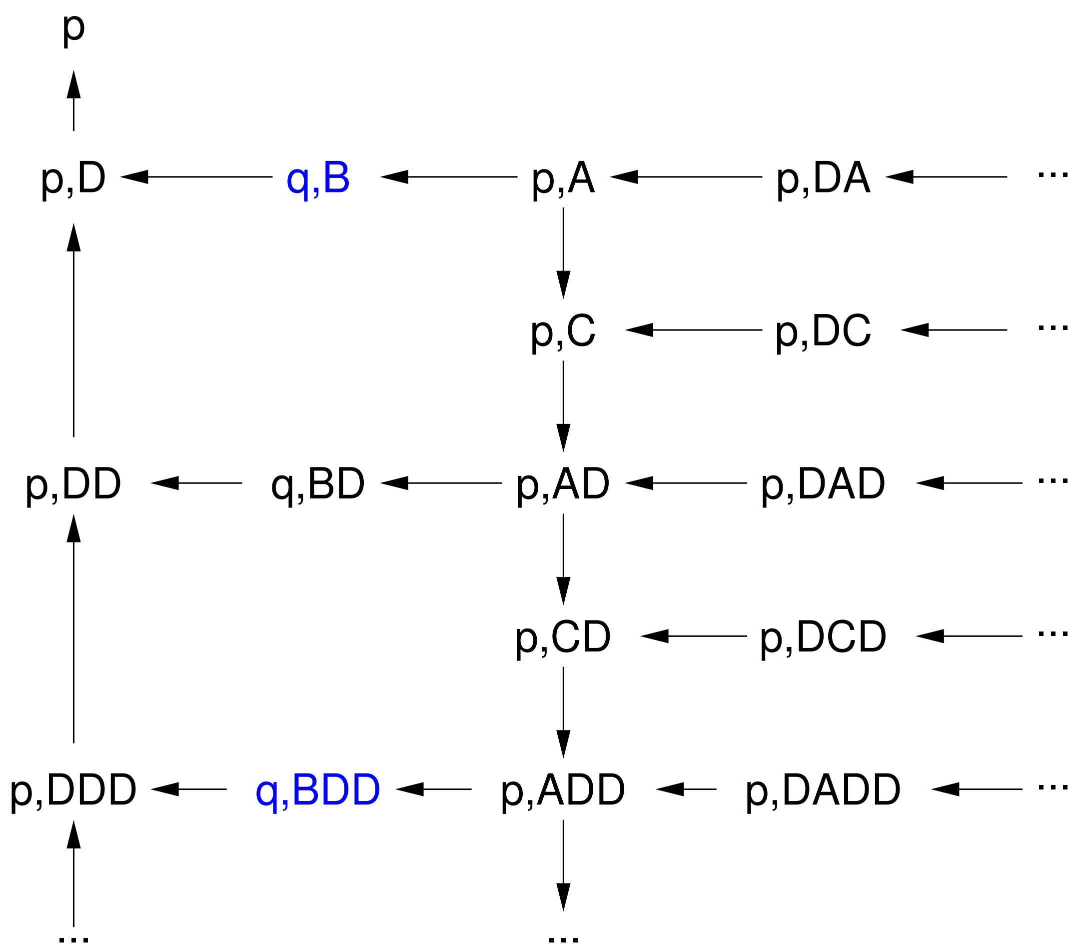
Construct automaton $\mathcal{A}$ accepting C.

Extend $\mathcal{A}$ to $\mathcal{A}'$ by adding transitions (according to some rule).

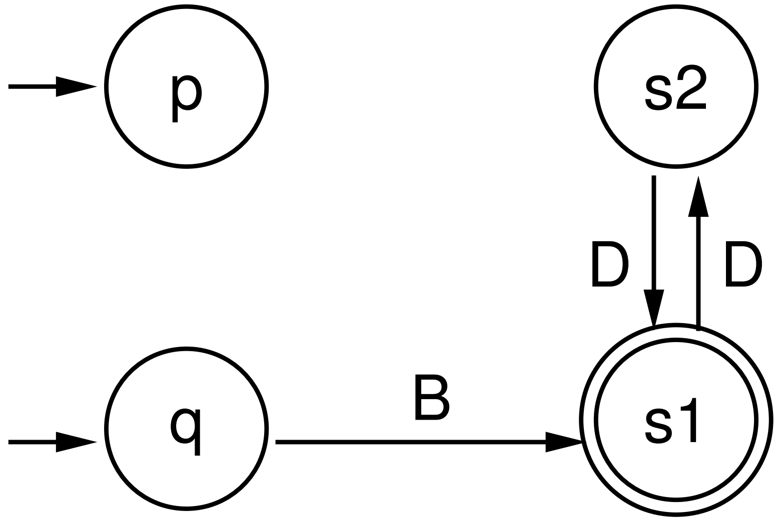
$post^*$ works in a similar fashion.

Example: an automaton accepting *C*



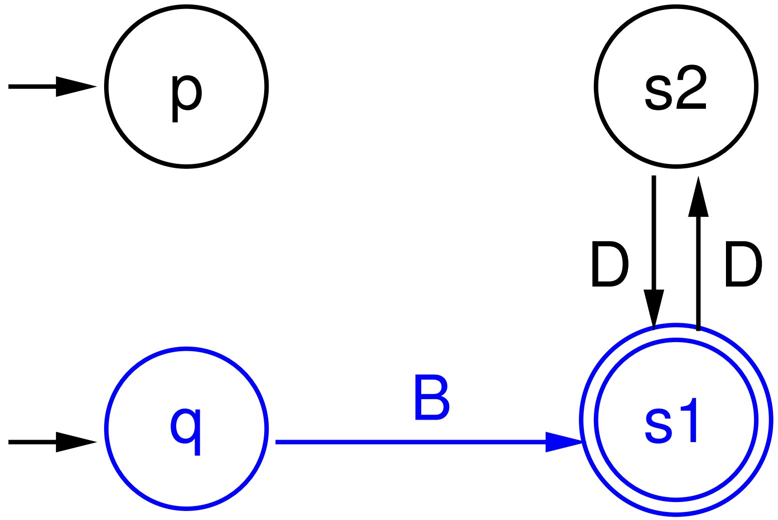


Extend $\mathcal{A}$ by adding transitions



Extend $\mathcal{A}$ by adding transitions

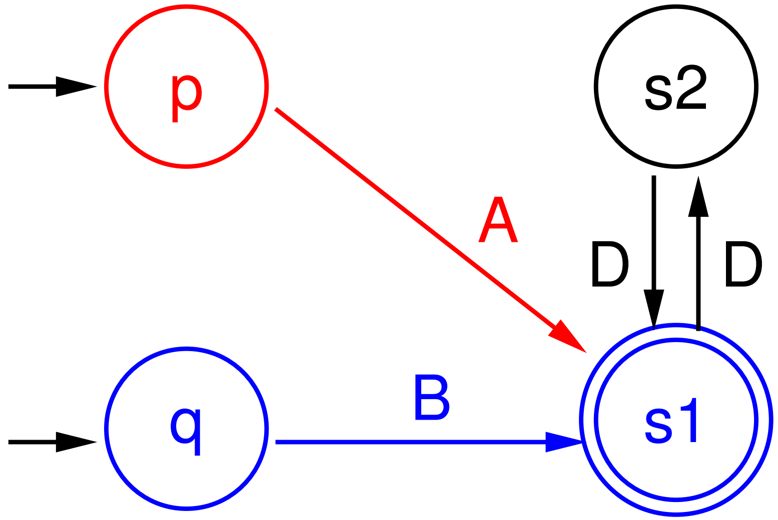
If the right-hand side of a rule has a path, .



Rule: $\langle p, A\rangle \hookrightarrow \langle q, B\rangle$ Path: $q \xrightarrow{B} s_1$

Extend $\mathcal{A}$ by adding transitions

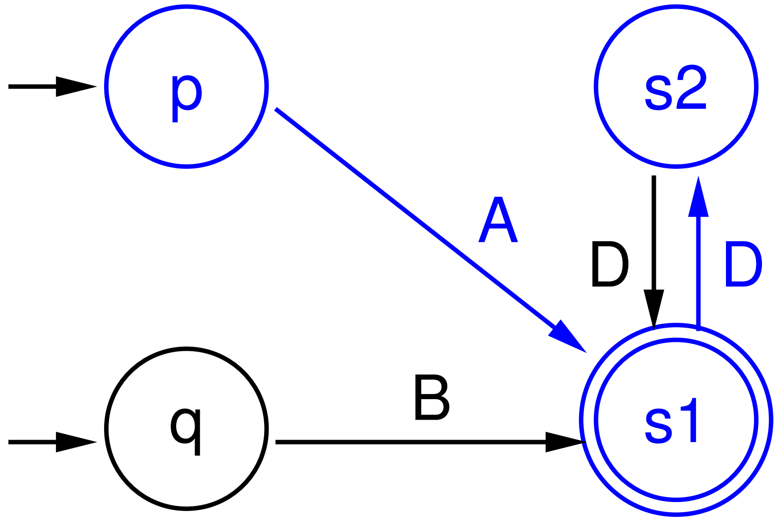
If the right-hand side of a rule has a path, add the left-hand side.



Rule: $\langle p, A \rangle \hookrightarrow \langle q, B \rangle$ Path: $q \xrightarrow{B} s_1$ New Path: $p \xrightarrow{A} s_1$

Extend $\mathcal{A}$ by adding transitions

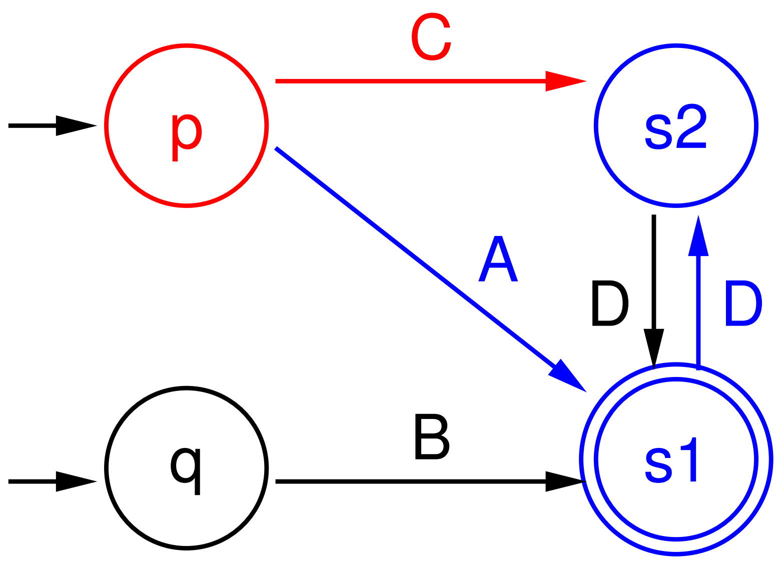
If the right-hand side of a rule has a path, .



Rule: $\langle p, C\rangle \hookrightarrow \langle p, AD\rangle$ Path: $p \xrightarrow{a} s_1 \xrightarrow{D} s_2$

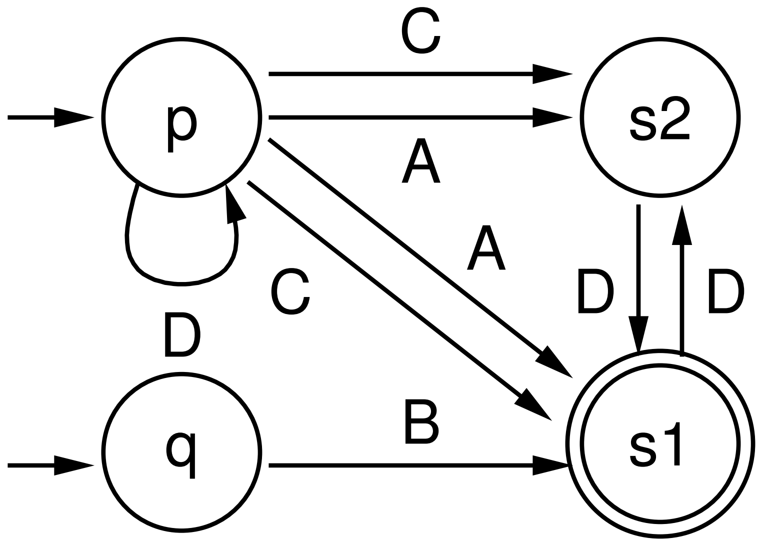
Extend $\mathcal{A}$ by adding transitions

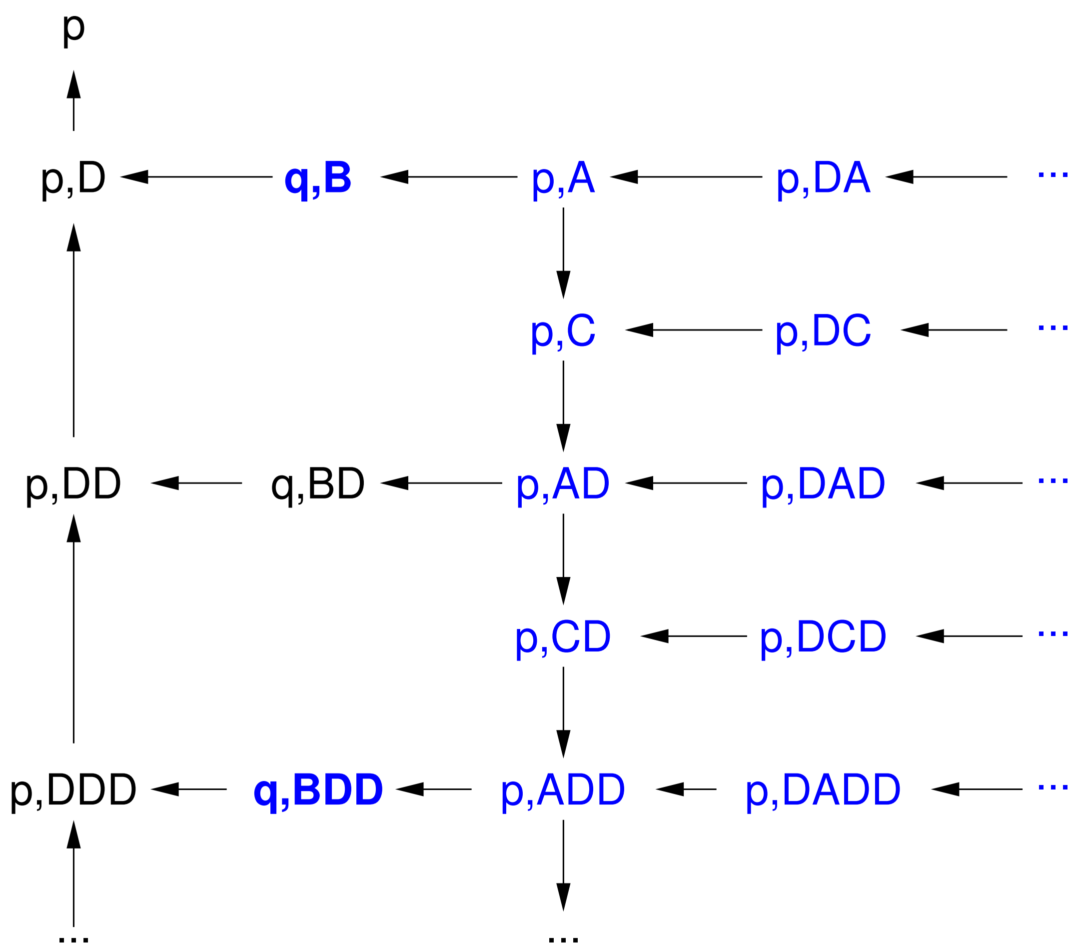
If the right-hand side of a rule has a path, add the left-hand side.



Rule: $\langle p, C \rangle \hookrightarrow \langle p, AD \rangle$ Path: $p \xrightarrow{a} s_1 \xrightarrow{D} s_2$ New Path: $p \xrightarrow{C} s_2$

Result: an automaton accepting $pre^*(C)$





Complexity of the procedure

Theorem: [Esparza, Hansel, Rossmanith, S.] Given a pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$ and an automaton $\mathcal{A}$ with states Q (where $P \subseteq Q$), computing the automaton for $pre^*(\mathcal{L}(\mathcal{A}))$ takes

$$\mathcal{O}(|Q|^2 \cdot |\Delta|) \text{ time.}$$

If $\mathcal{P}$ is derived from a program, ...

...the number (resp. size) of global variables is a cubic factor;

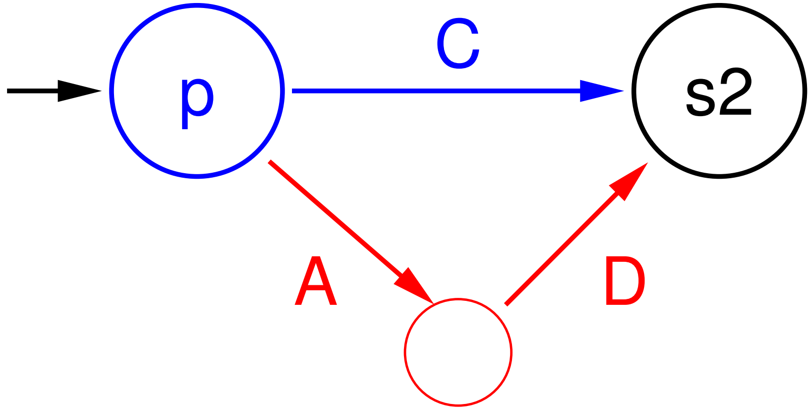
...the size of the control (i.e. number of lines) is a linear factor;

...the number (resp. size) of local variables is a linear factor.

Thus, the approach scales well for large programs.

Example: *post**

If the left-hand side of a rule has a path, add the right-hand side.



Rule: $\langle p, C\rangle \hookrightarrow \langle p, AD\rangle$ Path: $p \xrightarrow{C} s_2$ New Path: $p \xrightarrow{AD} s_2$

PDS vs inlining

Programs without recursive procedures could be translated to finite-state automata by inlining the procedure calls.

$\rightarrow$ leads to exponential blowup (in the worst case).

$\rightarrow$ state descriptors include data of both current and calling procedures.

$\rightarrow$ procedures may need to be re-evaluated many times.

Reachability algorithms for PDS exploit the inherent locality and modularity of procedural programming languages.

Model checking LTL

Reachability algorithms can be used to check safety properties such as:
"Whenever the quicksort procedure finishes, the array is sorted correctly."

LTL (Linear Temporal Logic) can express liveness properties such as:
"The quicksort procedure will always terminate."

LTL on finite-state systems

Translate liveness property into a Büchi automaton

Generate cross product of system and Büchi automaton

Find a “lasso”: a loop containing an accepting Büchi state

LTL on pushdown systems

Again, generate the cross product with a Büchi automaton. . .

Problem: infinitely many states, faulty executions may not loop around the same configuration.

However, a faulty execution is bound to contain a repeating head:

$\Rightarrow$ search for occurrences $\langle pA \rangle \Rightarrow^* \langle pAw \rangle$ passing an accepting Büchi state

Primitive solution: one reachability query per tuple pA

More efficient solution: Esparza, Hansel, Rossmanith, S.

Tool support

Reachability and LTL model checking implemented in the Moped tool.
`http://www7.in.tum.de/tools/moped/`

Control flow represented using pushdown systems.

Data represented symbolically, using binary decision diagrams.

Features:

Frontend for Boolean programs (device drivers)

Java frontend [Suwimonteerabuth, S. E. 05]

Abstraction refinement process [E., Kiefer, S. 06]

Test case generation for Java [Suwim., Berger, S., E. 07]

Eclipse plugin, extension to concurrent programs [Suwim. 09]

Related tools

Bebop/Static Driver Verifier (Ball, Rajamani, MSR)

checks safety properties of Windows device drivers written in C

C programs abstracted to Boolean Programs, i.e. programs with only boolean variables that represent predicates about the program states.

Boolean programs equivalent to PDS

Getafix (Madhudusan, Parlato, La Torre)

pushdown/boolean program checking based on games

Further extensions

Higher-order pushdown systems (Hague, Ong)

order 1: normal stacks; order 2: stack of stacks; etc

Ground tree rewrite systems (Löding; Gaiser)

PDS: configurations = words; GTRS: configurations = trees

μ-calculus (Bernhard, Steffen; Walukiewicz)

more powerful logics, but also computationally harder

pre* (but not *post**) on context-free grammars (E., Rossmanith, S.)

Solution: Quicksort algorithm contains a bug!

```
void quicksort (int left, int right) {
  int lo,hi,piv;
  if (left >= right) return;
  piv = a[right]; lo = left; hi = right;
  while (lo <= hi) {
    if (a[hi] > piv) {
      hi--;
    } else {
      swap a[lo],a[hi]; lo++;
    }
  }
  quicksort(left,hi); quicksort(lo,right);
}
```

If the chosen pivot happens to be the largest number . . .

Solution: Quicksort algorithm contains a bug!

```
void quicksort (int left, int right) {
  int lo,hi,piv;
  if (left >= right) return;
  piv = a[right]; lo = left; hi = right;
  while (lo <= hi) {
    if (a[hi] > piv) {
      hi--;
    } else {
      swap a[lo],a[hi]; lo++;
    }
  }
  quicksort(left,hi); quicksort(lo,right);
}
```

. . . then the if-condition will always be false . . .

Solution: Quicksort algorithm contains a bug!

```
void quicksort (int left, int right) {
  int lo,hi,piv;
  if (left >= right) return;
  piv = a[right]; lo = left; hi = right;
  while (lo <= hi) {
    if (a[hi] > piv) {
      hi--;
    } else {
      swap a[lo],a[hi]; lo++;
    }
  }
  quicksort(left,hi); quicksort(lo,right);
}
```

. . . and, since `hi` equals `right`, the program contains an infinite loop.

Moped: Runtimes for Quicksort

Since Moped works with finite data domains, the user needs to supply a bound on the size of the array and the number of bits used to represent the array elements.

Experiments on a corrected version of Quicksort, checking correctness and termination:

bits	array size	time
1	40	53.5 s
2	10	74.3 s
3	6	13.7 s
4	6	64.4 s

PDS with weights

Another example program

```
int x;

void main() {
  n1: x = 5;
  n2: p();
  n3: return;
}

void p() {
  n4: if (...) {
  n5:    return;
  n6: } else if (...) {
  n7:    x = x + 1;
  n8:    p();
  n9:    x = x - 1;
      } else {
  n10:   x = x - 1;
  n11:   p();
  n12:   x = x + 1;
      }
}
```

Question: What is the value of `x` at termination?

Interprocedural Data-Flow Analysis

Goal: Determine, for each program point n, the set of data-flow facts that hold whenever execution reaches n.

In this case: Linear relations between variables
(without restricting the range of variables!)

Approach:

Programs $\widehat{=}$ Pushdown systems

Analyses $\widehat{=}$ Semirings

Pushdown Systems with Weights

Extend pushdown rules with weights drawn from a semiring.

Weights can express distances, actions (i.e., “what happens when going from c to c'?”) etc.

Weights accumulate along a path, and we compute summaries of all paths going from every source to all the targets.

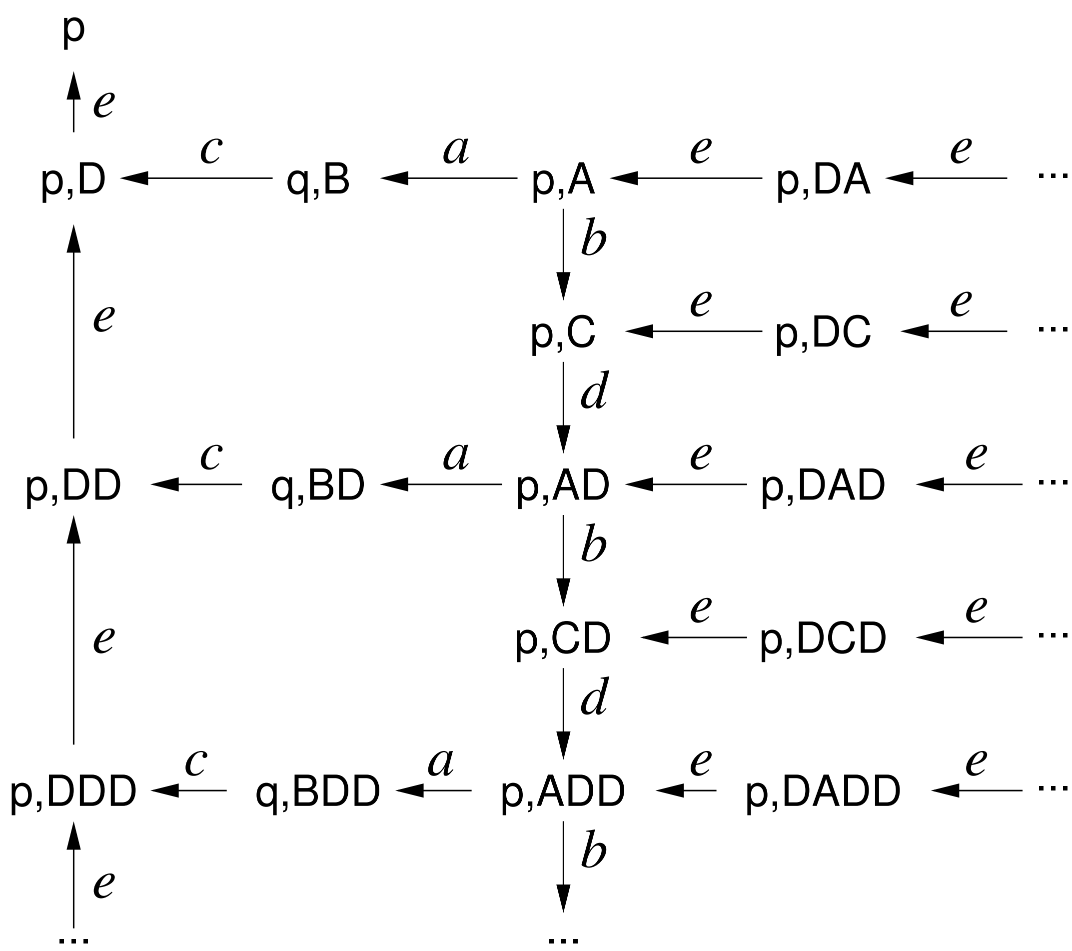
Thus, interprocedural data-flow analysis amounts to solving a generalised shortest-path problem on pushdown graphs.

Example: Weighted PDS

$$\langle p, A \rangle \overset{a}{\hookrightarrow} \langle q, B \rangle$$
$$\langle p, A \rangle \overset{b}{\hookrightarrow} \langle p, C \rangle$$
$$\langle q, B \rangle \overset{c}{\hookrightarrow} \langle p, D \rangle$$
$$\langle p, C \rangle \overset{d}{\hookrightarrow} \langle p, AD \rangle$$
$$\langle p, D \rangle \overset{e}{\hookrightarrow} \langle p, \varepsilon \rangle$$

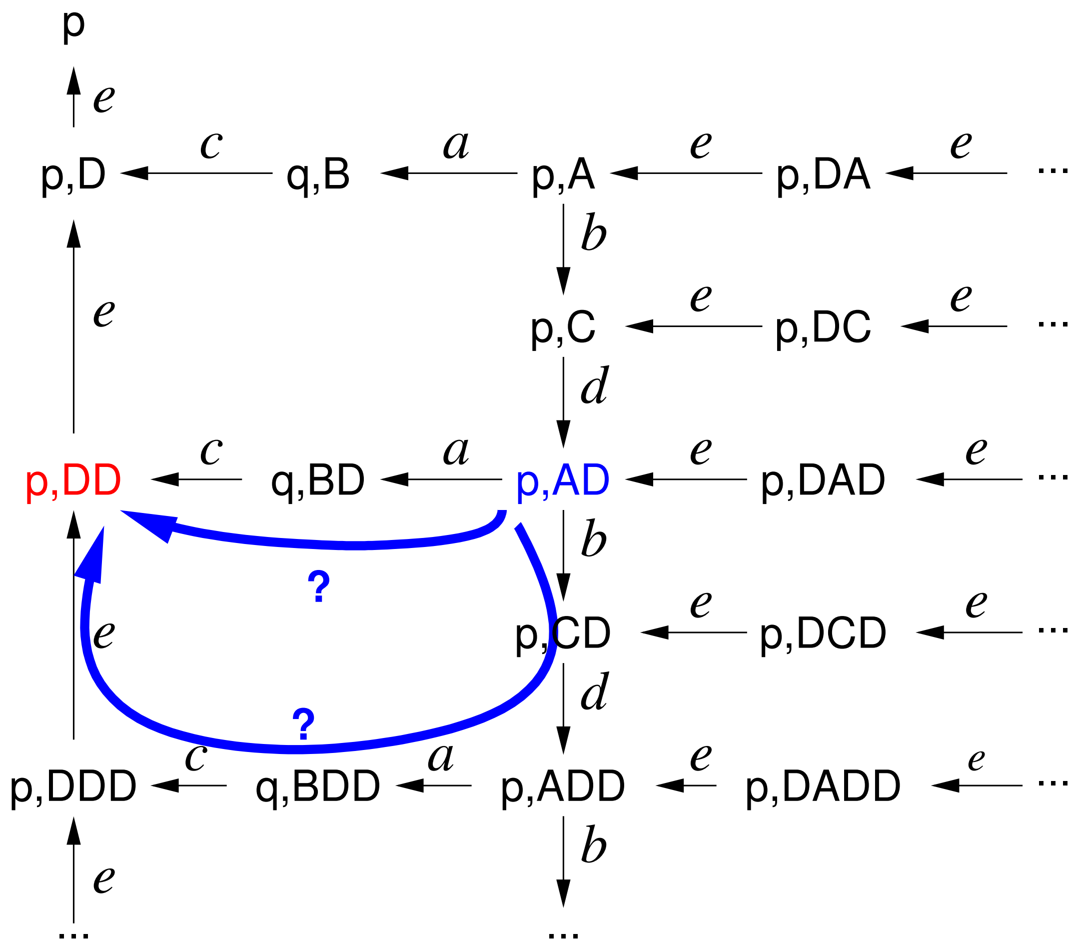
Example: Weighted PDS

$$\langle p, A\rangle \overset{a}{\hookrightarrow} \langle q, B\rangle$$
$$\langle p, A\rangle \overset{b}{\hookrightarrow} \langle p, C\rangle$$
$$\langle q, B\rangle \overset{c}{\hookrightarrow} \langle p, D\rangle$$
$$\langle p, C\rangle \overset{d}{\hookrightarrow} \langle p, AD\rangle$$
$$\langle p, D\rangle \overset{e}{\hookrightarrow} \langle p, \varepsilon\rangle$$



Example: Weighted PDS

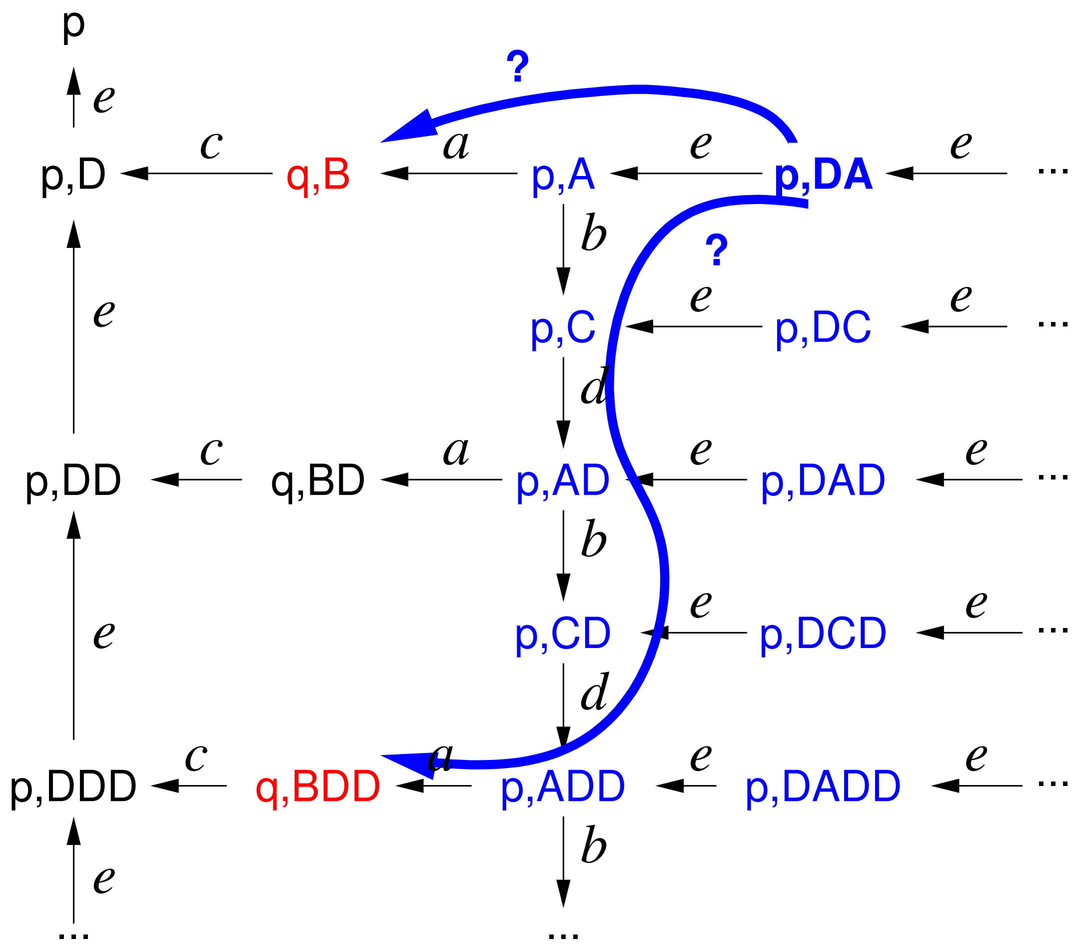
$\langle p, A\rangle \overset{a}{\hookrightarrow} \langle q, B\rangle$
$\langle p, A\rangle \overset{b}{\hookrightarrow} \langle p, C\rangle$
$\langle q, B\rangle \overset{c}{\hookrightarrow} \langle p, D\rangle$
$\langle p, C\rangle \overset{d}{\hookrightarrow} \langle p, AD\rangle$
$\langle p, D\rangle \overset{e}{\hookrightarrow} \langle p, \varepsilon\rangle$



Value of the paths leading from $\langle p, AD\rangle$ to $\langle p, DD\rangle$?

Example: Weighted PDS

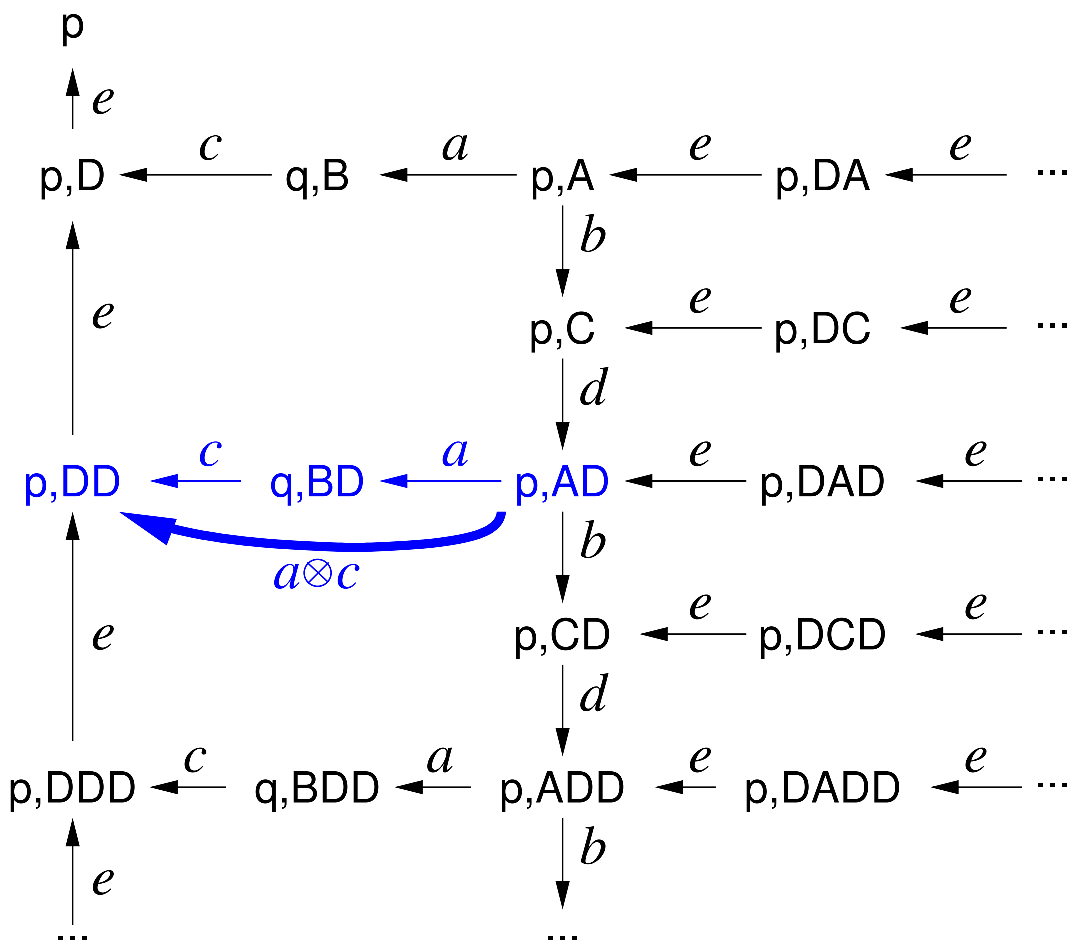
$$\langle p, A\rangle \xhookrightarrow{a} \langle q, B\rangle$$
$$\langle p, A\rangle \xhookrightarrow{b} \langle p, C\rangle$$
$$\langle q, B\rangle \xhookrightarrow{c} \langle p, D\rangle$$
$$\langle p, C\rangle \xhookrightarrow{d} \langle p, AD\rangle$$
$$\langle p, D\rangle \xhookrightarrow{e} \langle p, \varepsilon\rangle$$



For any $c \in pre^*(C)$, value of the paths leading from c into C?

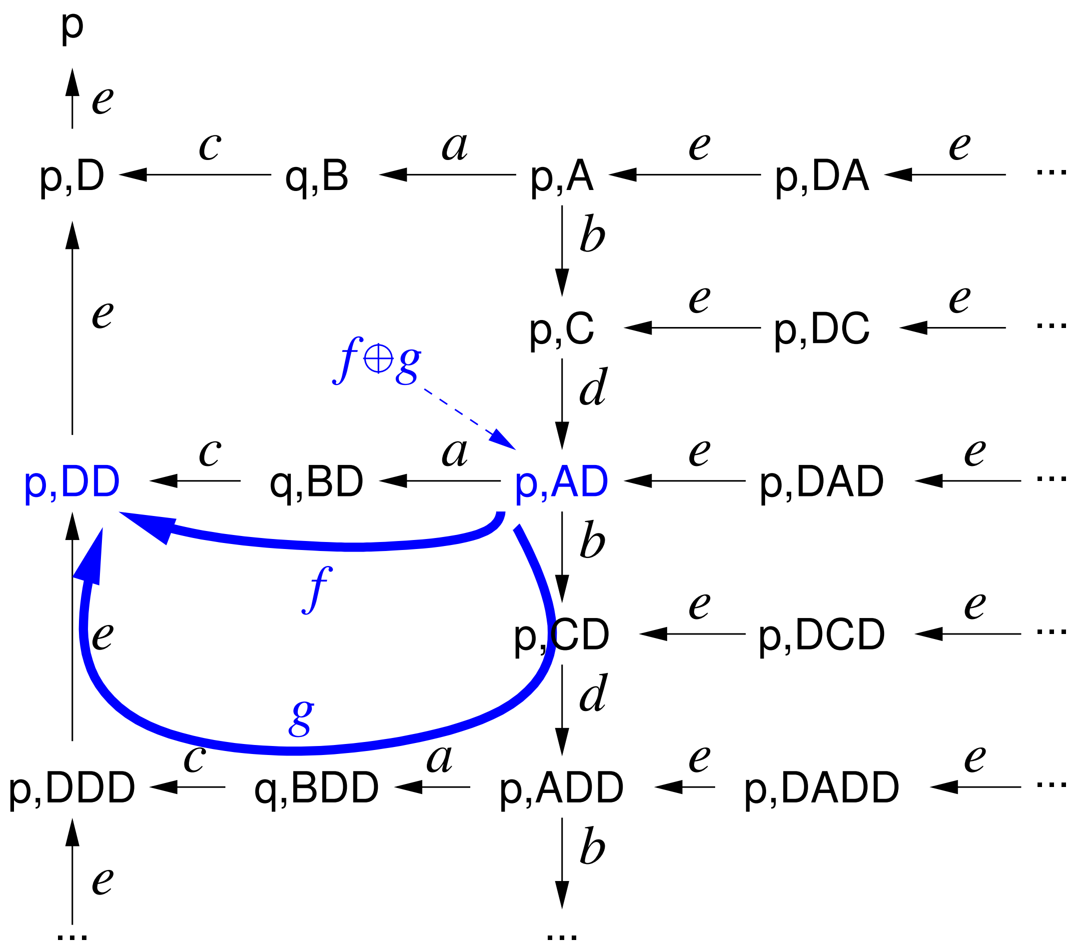
Extender operation: $\otimes$

Used to join values along a path in the PDS:



Combiner operation: $\oplus$

Summarizes the results of different paths:



Weighted PDS

A weighted pushdown system $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ features:

a pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$;

a semiring $\mathcal{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$;

a valuation function $f\colon \Delta \to D$.

(i.e., assign a semiring weight to each rule)

Generalized Reachability

Let $\pi = c_0 \xrightarrow{d_1} \cdots \xrightarrow{d_n} c_n$ be a path in $\mathcal{W}$. The value of π is $v(\pi) = \bigotimes_{i=1}^{n} d_i$, and $\Pi(c, C)$ is the set of all paths from c to configurations of C.

(for *pre**): Given a weighted PDS $\mathcal{W} = (P, \mathcal{S}, f)$ and a regular set C, compute for each c in $pre^*(C)$:

the "distance" from c to C:

$$\delta(c) = \bigoplus \{ v(\pi) \mid \pi \in \Pi(c, C) \}$$

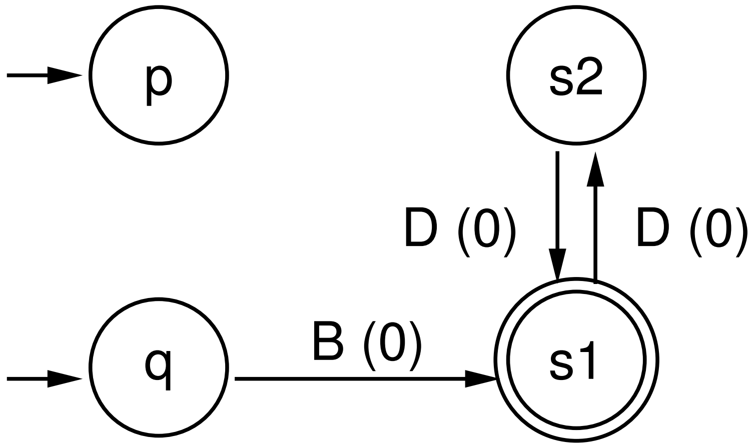
Semiring

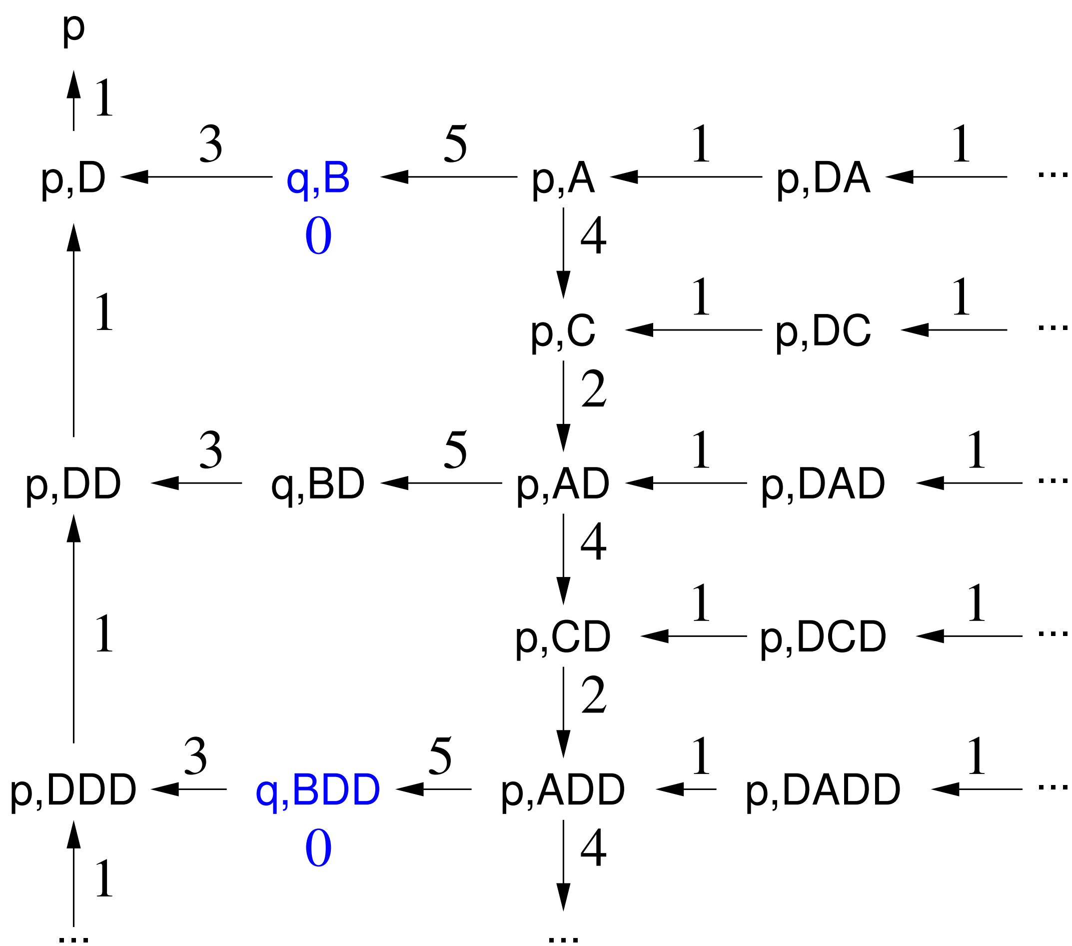
$\mathcal{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ is a bounded idempotent semiring iff:

- $(D, \oplus)$ is a commutative monoid with n.e. $\bar{0}$, and $a \oplus a = a$ for all $a \in D$.
- $(D, \otimes)$ is a monoid with n.e. $\bar{1}$.
- Distributivity, i.e. for all $a, b, c \in D$:
$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \text{ and}$$
$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$$
- Annihilator: $a \otimes \bar{0} = \bar{0} = \bar{0} \otimes a$ for all $a \in D$
- No infinite descending chains in the partial order $\sqsubseteq$ induced by $a \sqsubseteq b$ iff $a \oplus b = a$.

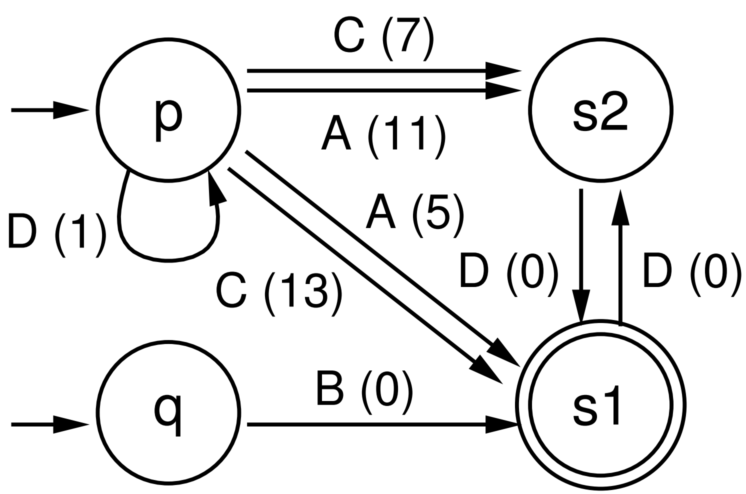
For bounded idempotent semirings, the generalized reachability problem can be solved by a simple modification of the *pre**/*post** procedure.

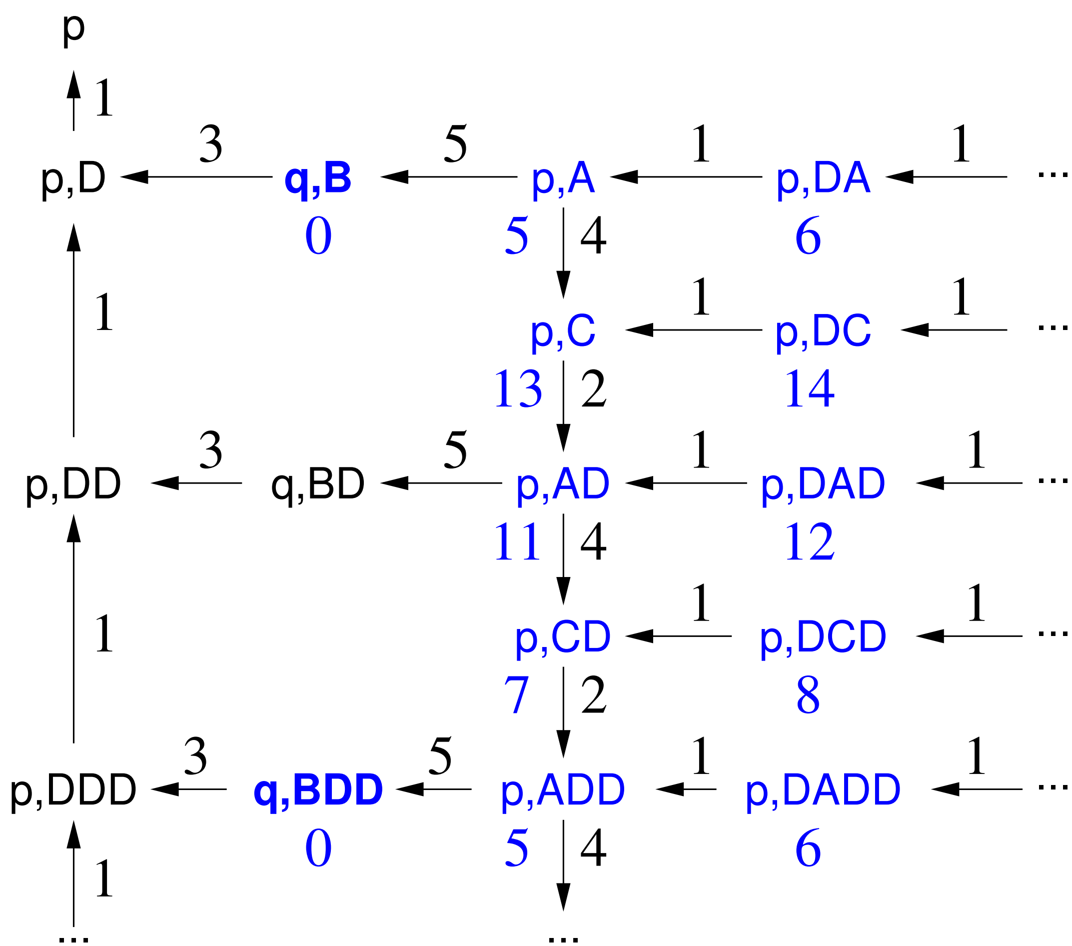
Example: $\mathcal{S} = (\mathbb{N}, \min, +, \infty, 0)$, initial automaton



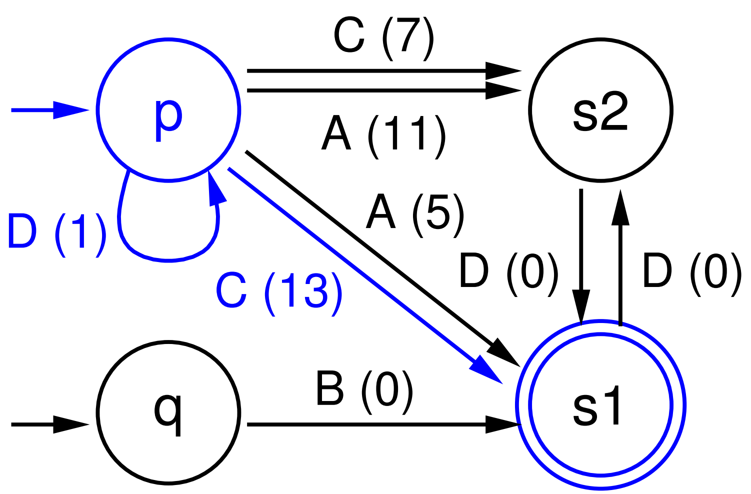


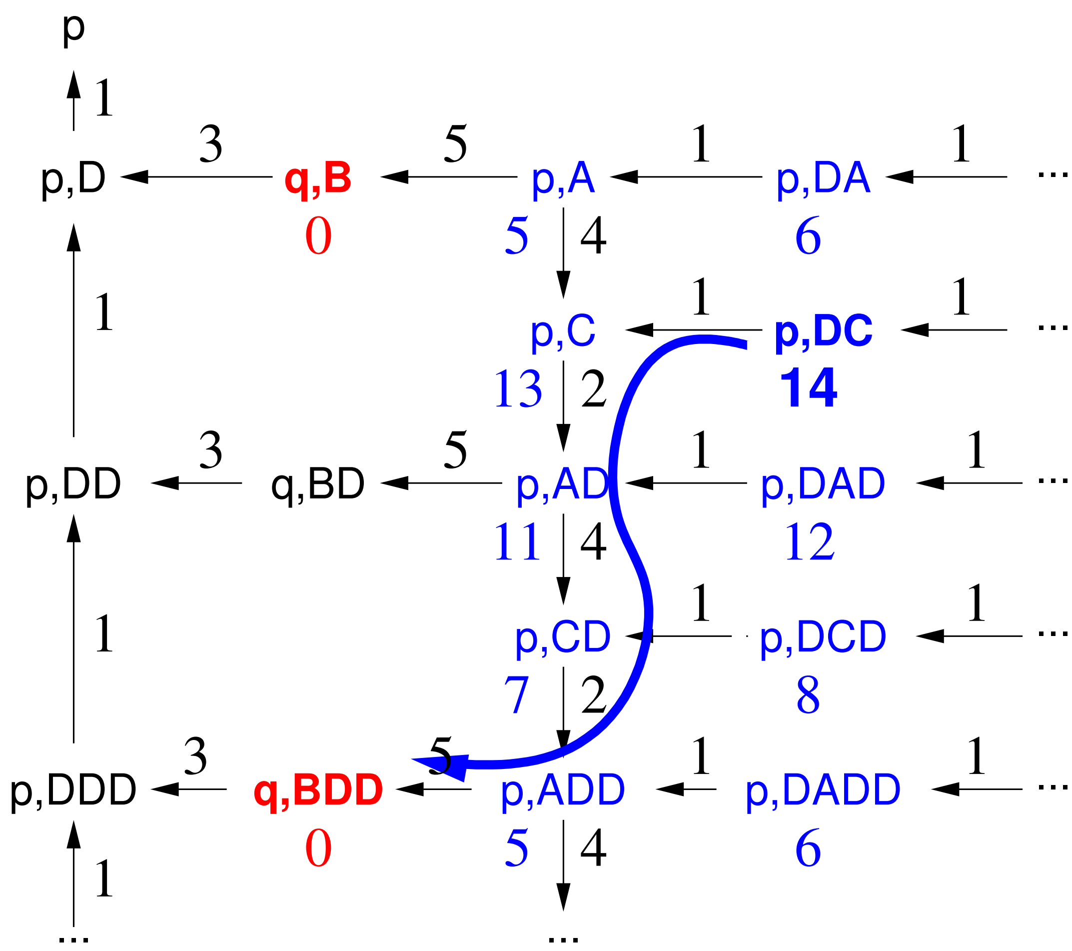
Example: $\mathcal{S} = (\mathbb{N}, \min, +, \infty, 0)$, final automaton





Example: $\mathcal{S} = (\mathbb{N}, \min, +, \infty, 0)$, final automaton





Linear-Constant Analysis as a Semiring

Evaluate expressions of the form `x := 2*y+5` or `x := 7;` .

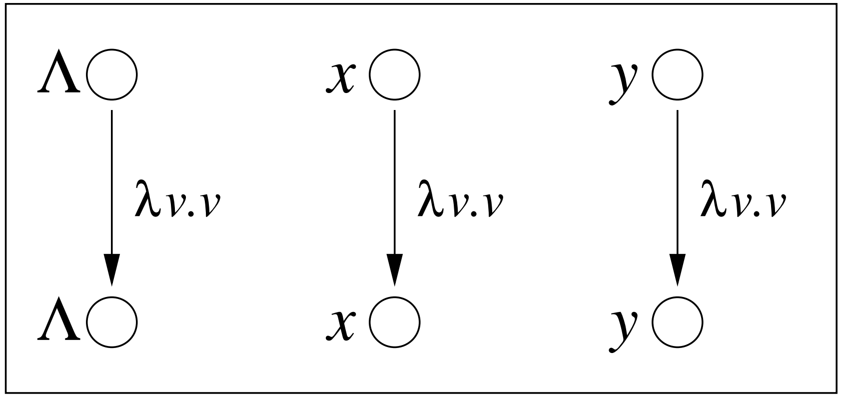
Represent everything other assignment as `x :=` $\perp$`;` (not representable).

Goal: Determine whether at program point *n*, variable `x` is a linear function of some variable `y`.

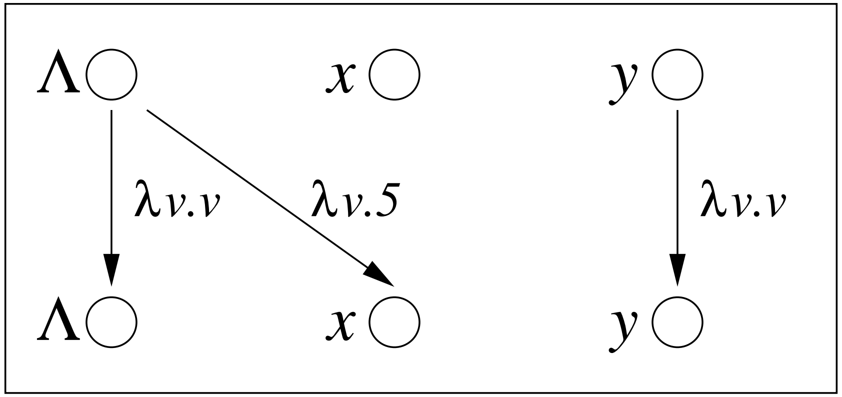
Represent the effect of statements using semiring values.

Here: Semiring values $\widehat{=}$ bipartite graphs

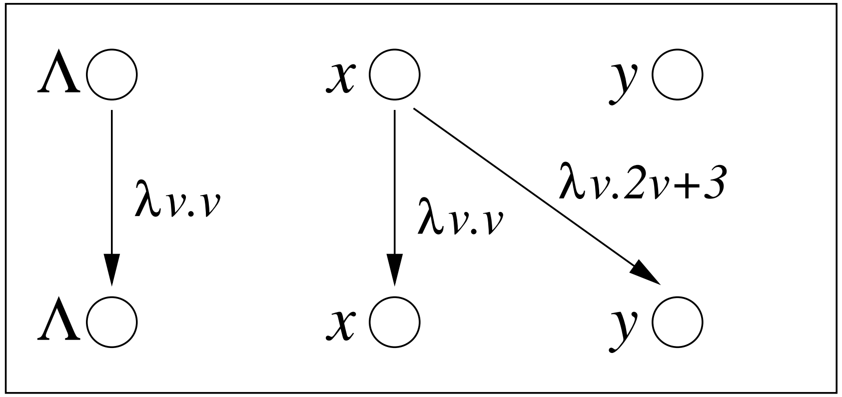
Some assignments and their representations



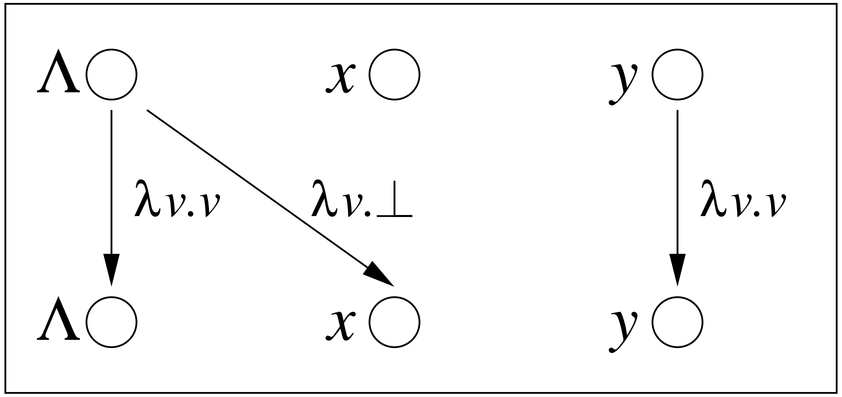
skip



x := 5

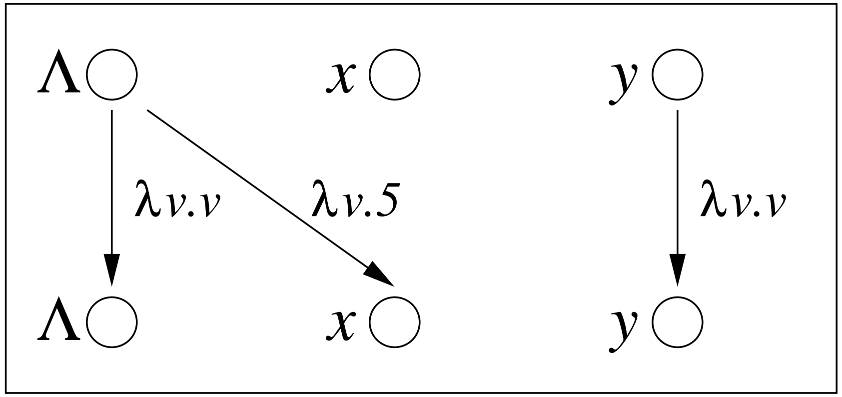


y := 2*x + 3

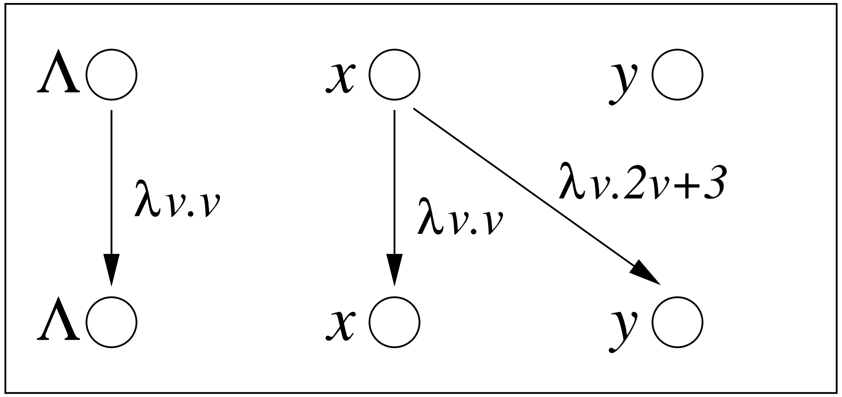


x := sin(y)

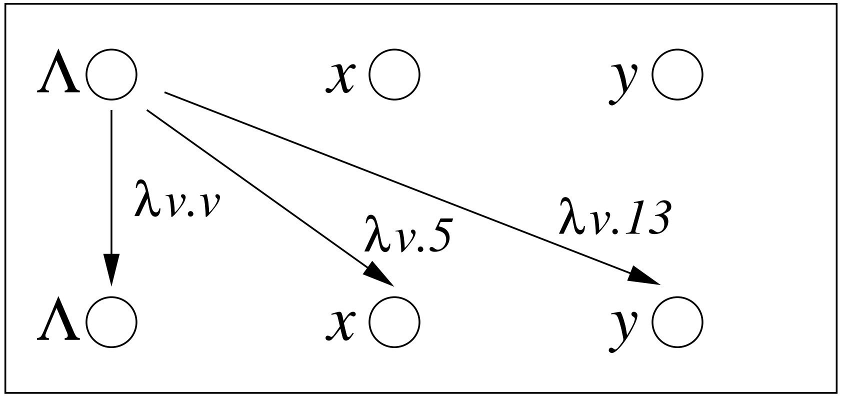
Extend: Concatenate the two graphs



x := 5

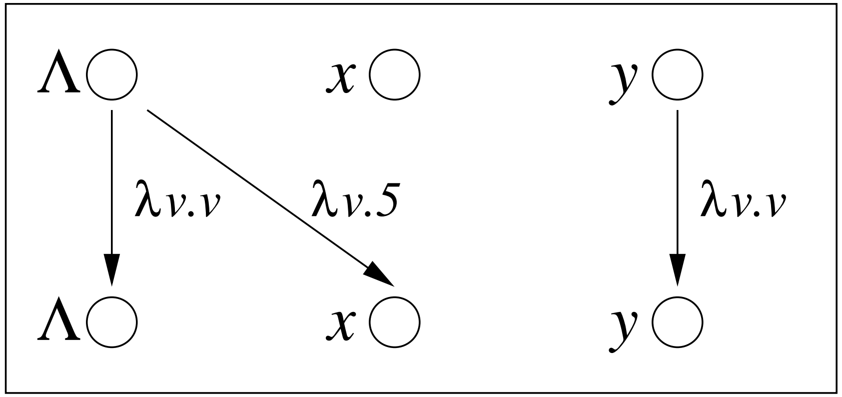


y := 2*x + 3

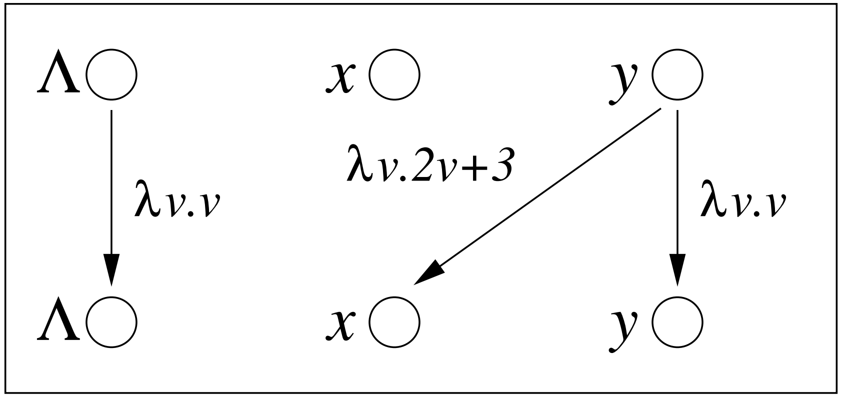


x := 5 $\otimes$ **y := 2*x + 3**

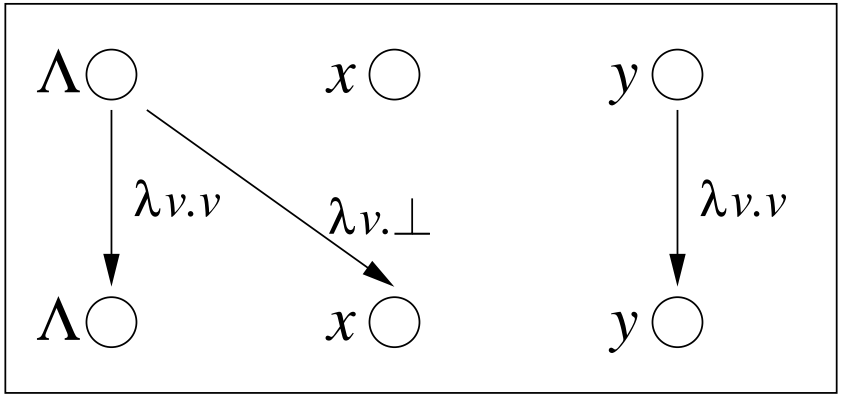
Combine: Check if information is compatible



x := 5



x := 2*y + 3

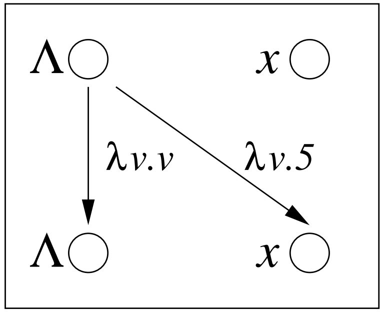


x := 5 $\oplus$ x := 2*y + 3

Example

```
int x;

void main() {
  n1: x = 5;
  n2: p();
  n3: return;
}

void p() {
  n4: if (...) {
  n5:     return;
  n6: } else if (...) {
  n7:     x = x + 1;
  n8:     p();
  n9:     x = x - 1;
      } else {
  n10:    x = x - 1;
  n11:    p();
  n12:    x = x + 1;
      }
}
```

Weighted *post** computation for $C = \{\langle q, n_1 \rangle\}$ yields $\delta(\langle q, \varepsilon \rangle) =$



i.e., upon termination `x` is always 5.

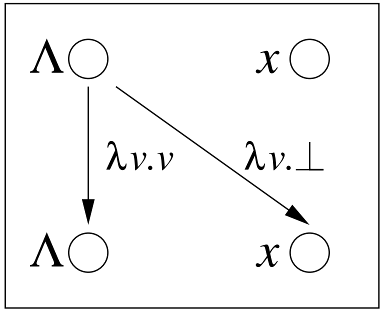
Example

```
int x;

void main() {
  n1: x = 5;
  n2: p();
  n3: return;
}

void p() {
  n4: if (...) {
  n5:    return;
  n6: } else if (...) {
  n7:    x = x + 1;
  n8:    p();
  n9:    x = x - 1;
      } else {
  n10:   x = x - 1;
  n11:   p();
  n12:   x = x + 1;
      }
}
```

Weighted *pre** operation for

$C = \{ \langle q, n_4 w \rangle \mid w \in \Gamma^* \}$

yields $\delta(\langle q, n_1 \rangle) =$



i.e., `p` may be entered with different values for `x`.

Work on PDS with weights

B.i. semirings: Reps, S., Jha, Melski 03/05, Bouajjani, E., Touili 03/04

WPDS library: `http://www.fmi.uni-stuttgart.de/szs/tools/wpds`

Lots of applications: Reps, Kidd, Lal et al, WPDS++ library

PDS as non-linear equation systems: $A \rightarrow BC$ becomes $f(a) = f(b) \otimes f(c)$

probabilistic PDS: E., Kučera, Mayr / Etessami, Yannakakis

more general semirings: E., Kiefer, Luttenberger (“Newtonian program analysis”)

PDS and concurrency

Multithreaded PDS

[Ramalingam 00] Reachability problem is undecidable for concurrent pushdown systems with shared memory

The usual remedies: approximative techniques or restricted models

[Kahlon, Ivančić, Gupta 07] Communication via nested locks

[Basler, Hague et al 10] Counter abstraction, disallow recursion (BOOM)

[Bouajjani, Müller-Olm, Touili 05] Dynamic Pushdown Network (DPN): PDS + dynamic thread creation, no communication

$$\langle q_1, a\rangle \hookrightarrow \langle q_2, b\rangle \triangleright \langle q_3, c\rangle$$

[Bouajjani, Esparza, Touili 03] compute an (over-)approximation of each thread individually, then intersect

[Qadeer, Rehof 05] under-approximation by context-bounded analysis

Context-bounded reachability analysis

Multiple stacks and one common control state

Global configurations have the form $(g, \alpha_1, \ldots, \alpha_n)$

A transition of process i modifies g and α_i

Context: a sequence of transitions performed by a single thread

Compute reachability for a fixed number of contexts

More pointers

Some additional papers on CBA:

Bouajjani, E., S., Strejček 05: process creation, improved algorithm

Bouajjani, Fratani, Qadeer 07: heap structures

La Torre, Madhudusan, Parlato 08: FIFO queues

Suwimonteerabuth, E., S. 08: symbolic implementation based on lazy splitting

Lal, Touili, Kidd, Reps 08: implementation based on transducers

Bouajjani, Atig et al: many extensions of the theory

End of Talk