Program Analysis and Verification

0368-4479

http://www.cs.tau.ac.il/~maon/teaching/2013-2014/paav/paav1314b.html

Noam Rinetzky

Lecture 13: Numerical, Pointer & Shape Domains

Slides credit: Roman Manevich, Mooly Sagiv, Eran Yahav, Ganesan Ramalingam

Abstract Interpretation [Cousot'77]

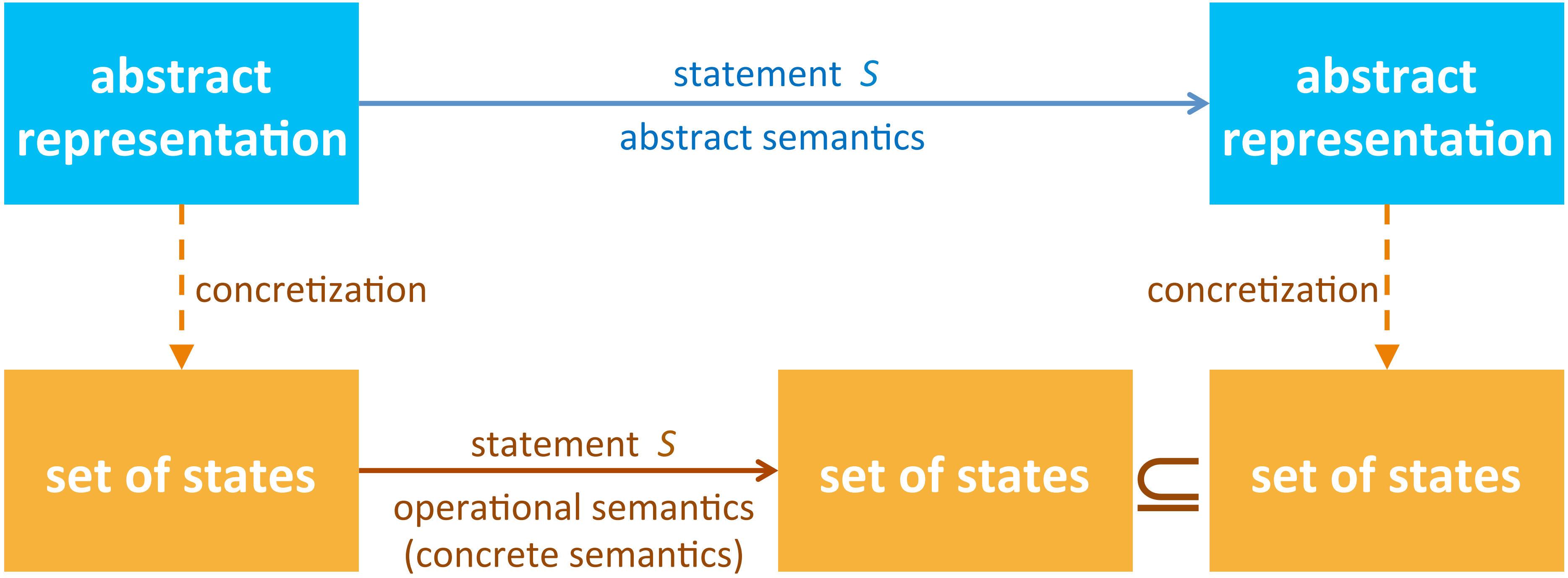
- Mathematical foundation of static analysis
  - Abstract domains
    - Abstract states
    - Join ($\sqcup$)
  - Transformer functions
    - Abstract steps
  - Chaotic iteration
    - Abstract computation
    - Structured Programs

Lattices
$(D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$

Monotonic functions

Fixpoints

Abstract (conservative) interpretation



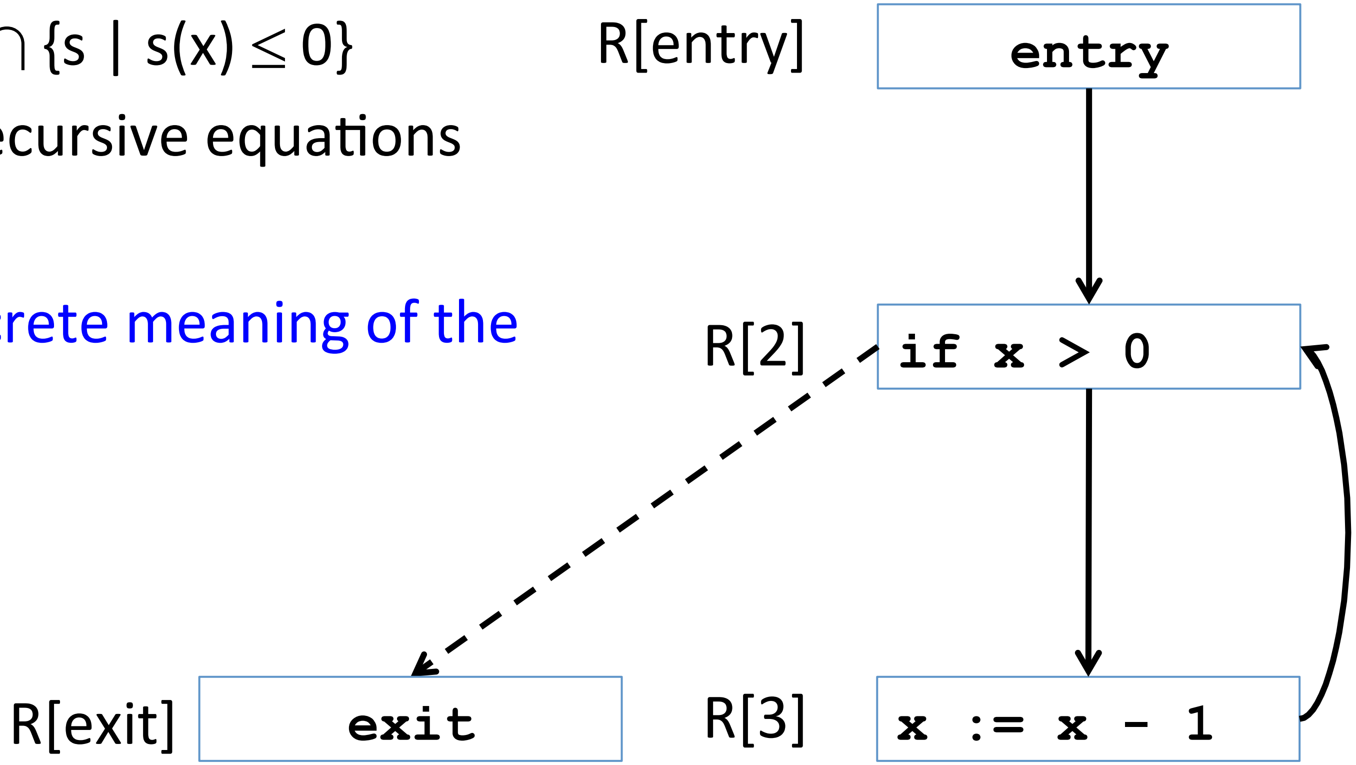
The collecting lattice

- Lattice for a given control-flow node *v*:
  L_v=($2^{\mathbf{State}}$, ⊆, ∪, ∩, ∅, **State**)
- Lattice for entire control-flow graph with nodes *V*:
  $$L_{CFG} = \mathrm{Map}(V, L_v)$$
- We will use this lattice as a baseline for static analysis and define abstractions of its elements

Equational definition of the semantics

- $R[2] = R[\text{entry}] \cup [\![\mathbf{x{:=}x{-}1}]\!]\, R[3]$
- $R[3] = R[2] \cap \{s \mid s(x) > 0\}$
- $R[\text{exit}] = R[2] \cap \{s \mid s(x) \leq 0\}$
- A system of recursive equations

- Solution: concrete meaning of the program



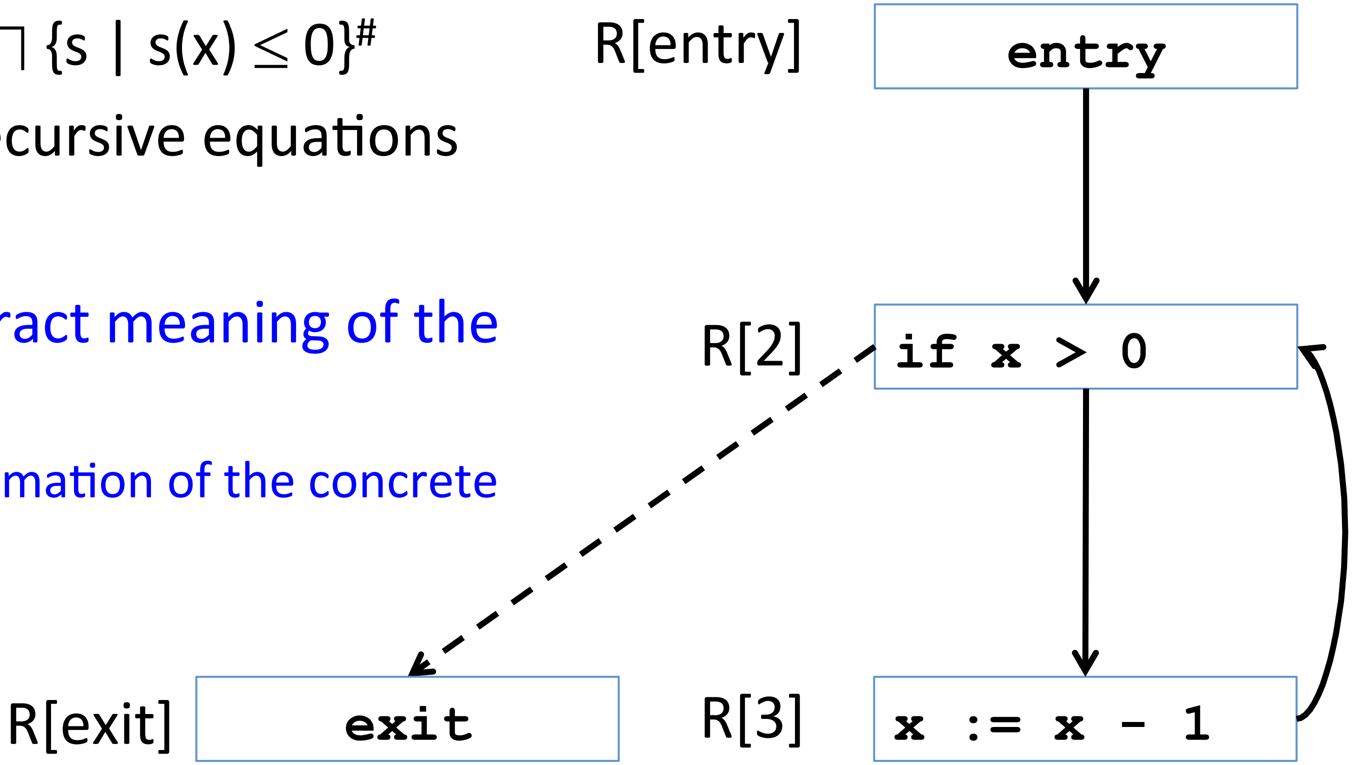
An abstract semantics

Abstract transformer for **`x:=x-1`**

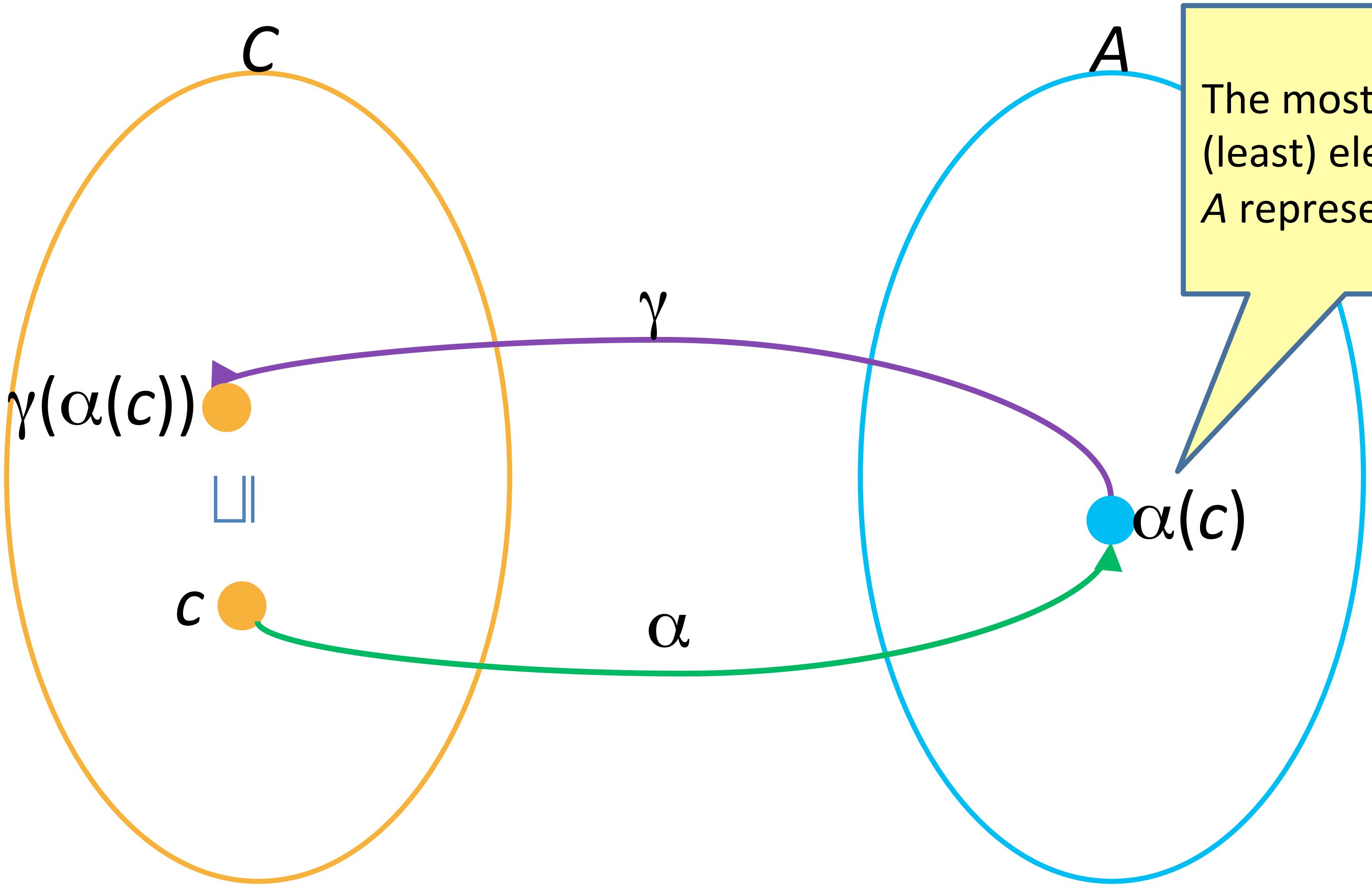
Abstract representation of {s | s(x) < 0}

- $R[2] = R[\text{entry}] \sqcup [\![\mathbf{x{:=}x{-}1}]\!]^{\#}\ R[3]$
- $R[3] = R[2] \sqcap \{s \mid s(x) > 0\}^{\#}$
- $R[\text{exit}] = R[2] \sqcap \{s \mid s(x) \leq 0\}^{\#}$
- A system of recursive equations

- Solution: abstract meaning of the program
  - Over-approximation of the concrete semantics



Galois Connection: $c \sqsubseteq \gamma(\alpha(c))$



The most precise (least) element in *A* representing *c*

Monotone functions

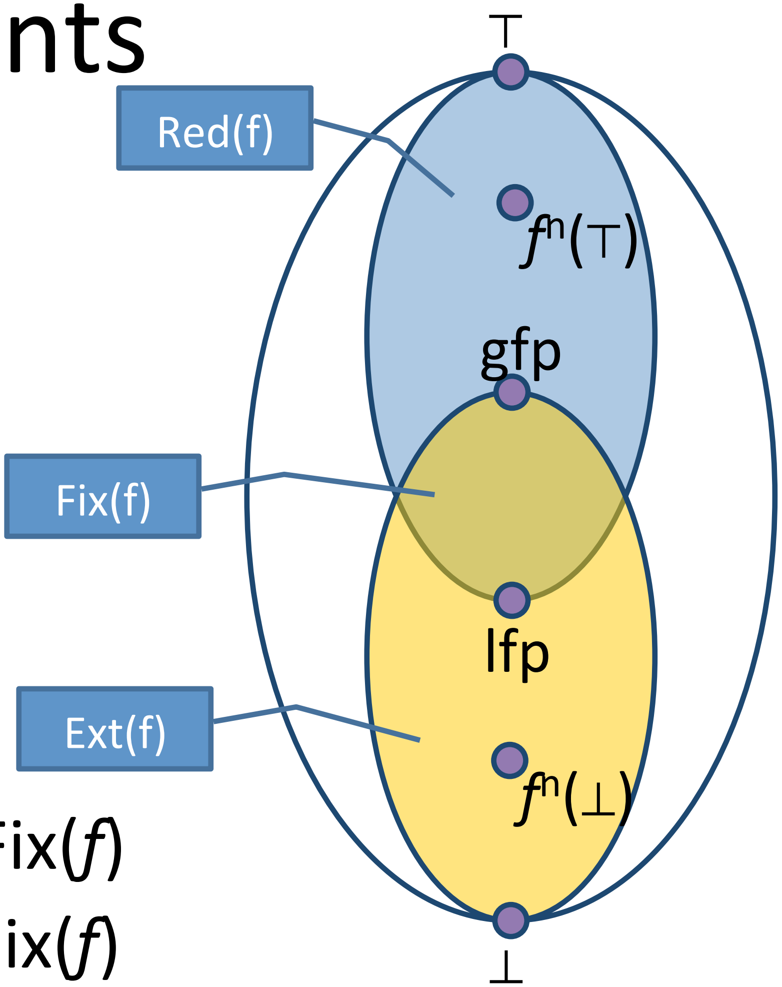
- Let L_1=(D_1, $\sqsubseteq$) and L_2=(D_2, $\sqsubseteq$) be two posets
- A function $f : D_1 \rightarrow D_2$ is monotone if for every pair x, $y \in D_1$

  $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$
- A special case: L_1=L_2=(D, $\sqsubseteq$)

  $f : D \rightarrow D$

Fixed-points

- $L = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$
- $f : D \longrightarrow D$ **monotone**
- $\mathrm{Fix}(f) = \{ d \mid f(d) = d \}$
- $\mathrm{Red}(f) = \{ d \mid f(d) \sqsubseteq d \}$
- $\mathrm{Ext}(f) = \{ d \mid d \sqsubseteq f(d) \}$
- **Theorem** [Tarski 1955]
  - $\mathrm{lfp}(f) = \sqcap\mathrm{Fix}(f) = \sqcap\mathrm{Red}(f) \in \mathrm{Fix}(f)$
  - $\mathrm{gfp}(f) = \sqcup\mathrm{Fix}(f) = \sqcup\mathrm{Ext}(f) \in \mathrm{Fix}(f)$



1. A solution always exist
2. It unique
3. Not always computable

Continuity and ACC condition

- Let $L = (D, \sqsubseteq, \sqcup, \bot)$ be a complete partial order
  - Every ascending chain has an upper bound
- A function f is continuous if for every increasing chain $Y \subseteq D^*$,

  $$f(\sqcup Y) = \sqcup\{ f(y) \mid y \in Y \}$$
- L satisfies the ascending chain condition (ACC) if every ascending chain eventually stabilizes:

  $$d_0 \sqsubseteq d_1 \sqsubseteq \ldots \sqsubseteq d_\mathrm{n} = d_{\mathrm{n}+1} = \ldots$$

Fixed-point theorem [Kleene]

- Let $L = (D, \sqsubseteq, \sqcup, \perp)$ be a complete partial order and a **continuous** function $f: D \rightarrow D$ then

$$\mathrm{lfp}(f) = \bigsqcup_{n \in N} f^n(\perp)$$

- **Lemma:** Monotone functions on posets satisfying ACC are continuous

Resulting algorithm

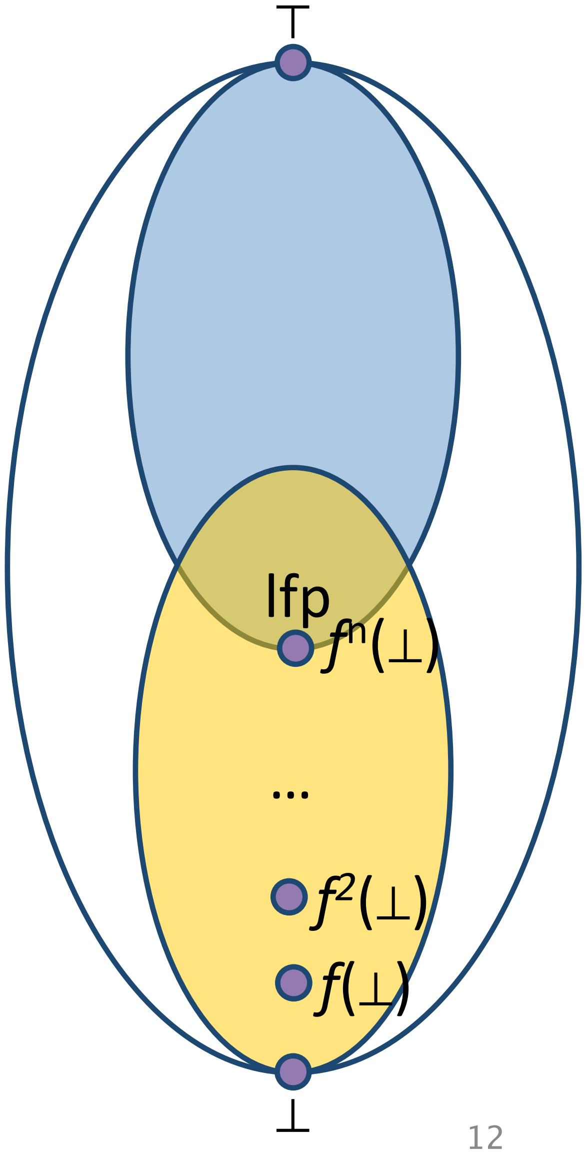
- Kleene's fixed point theorem gives a constructive method for computing the lfp

Mathematical definition

$$\mathrm{lfp}(f) = \bigsqcup_{n\in N} f^n(\bot)$$

Algorithm

```
$d := \bot$
**while** $f(d) \neq d$ **do**
    $d := d \sqcup f(d)$
**return** $d$
```



Sound abstract transformer

- Given two lattices
  $C = (D^C, \sqsubseteq^C, \sqcup^C, \sqcap^C, \perp^C, \top^C)$
  $A = (D^A, \sqsubseteq^A, \sqcup^A, \sqcap^A, \perp^A, \top^A)$
  and $GC^{C,A}=(C, \alpha, \gamma, A)$ with
- A concrete transformer $f : D^C \longrightarrow D^C$
  an abstract transformer $f^\# : D^A \longrightarrow D^A$
- We say that $f^\#$ is a sound transformer (w.r.t. f) if
  - $\forall c$: $\alpha(f(c)) \sqsubseteq f^\#(\alpha(c))$
  - $\forall a$: $\alpha(f(\gamma(a))) \sqsubseteq^A f^\#(a)$

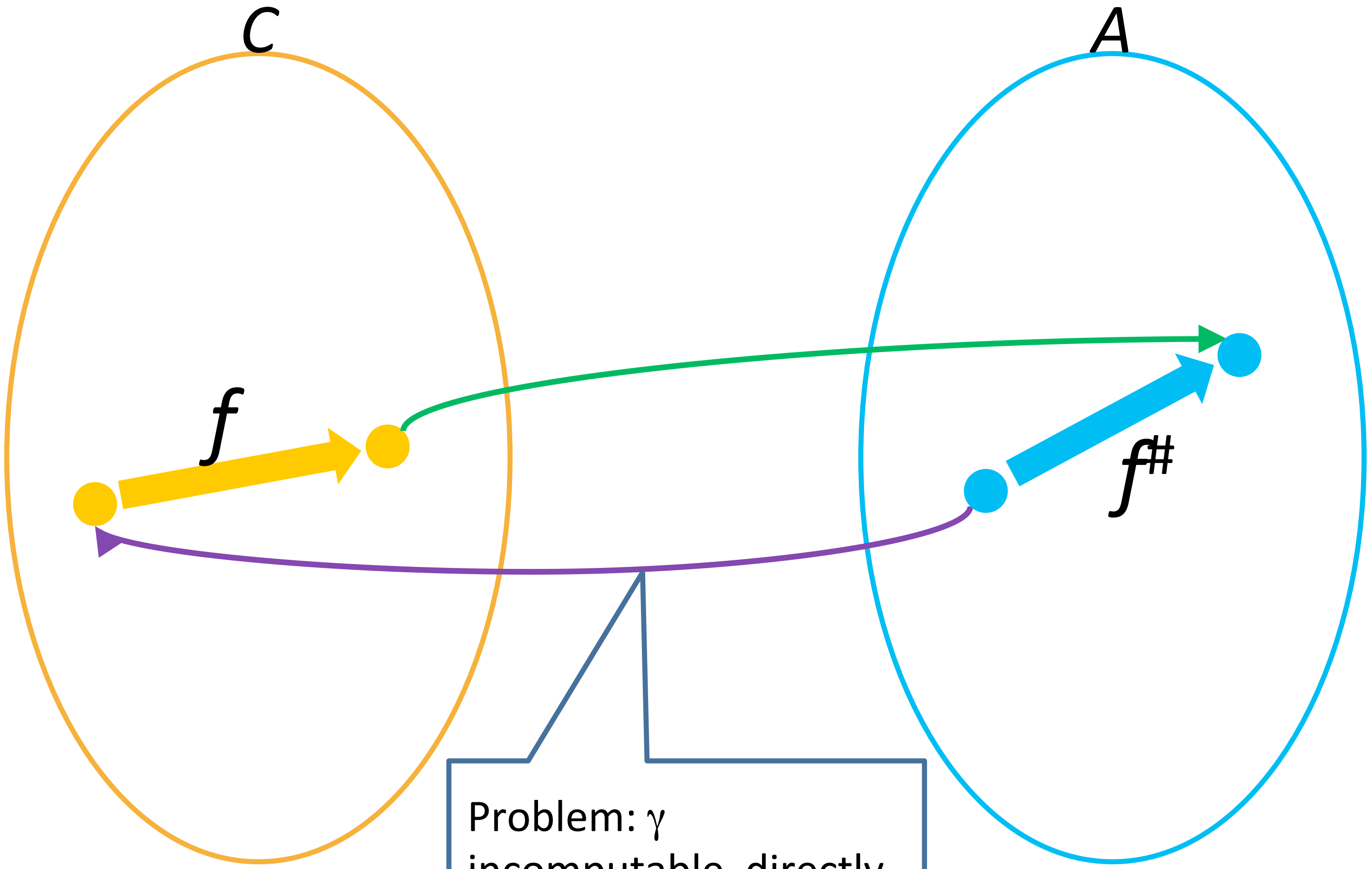
Soundness

1. Given two complete lattices
   $C = (D^C, \sqsubseteq^C, \sqcup^C, \sqcap^C, \bot^C, \top^C)$
   $A = (D^A, \sqsubseteq^A, \sqcup^A, \sqcap^A, \bot^A, \top^A)$
   and $GC^{C,A}=(C, \alpha, \gamma, A)$ with
2. Monotone concrete transformer $f : D^C \longrightarrow D^C$
3. Monotone abstract transformer $f^\# : D^A \longrightarrow D^A$ s
4. Either $\forall a \in D^A : f(\gamma(a)) \sqsubseteq \gamma(f^\#(a))$
   or $\forall c \in D^C : \alpha(f(c)) \sqsubseteq f^\#(\alpha(c))$

Then $\text{lfp}(f) \sqsubseteq \gamma(\text{lfp}(f^\#))$ and $\alpha(\text{lfp}(f)) \sqsubseteq \text{lfp}(f^\#)$

Best (induced) transformer [CC'77]

$$\boxed{f^{\#}(a) = \alpha(f(\gamma(a)))}$$



Problem: γ incomputable directly

Fixed-point theorem [Kleene]

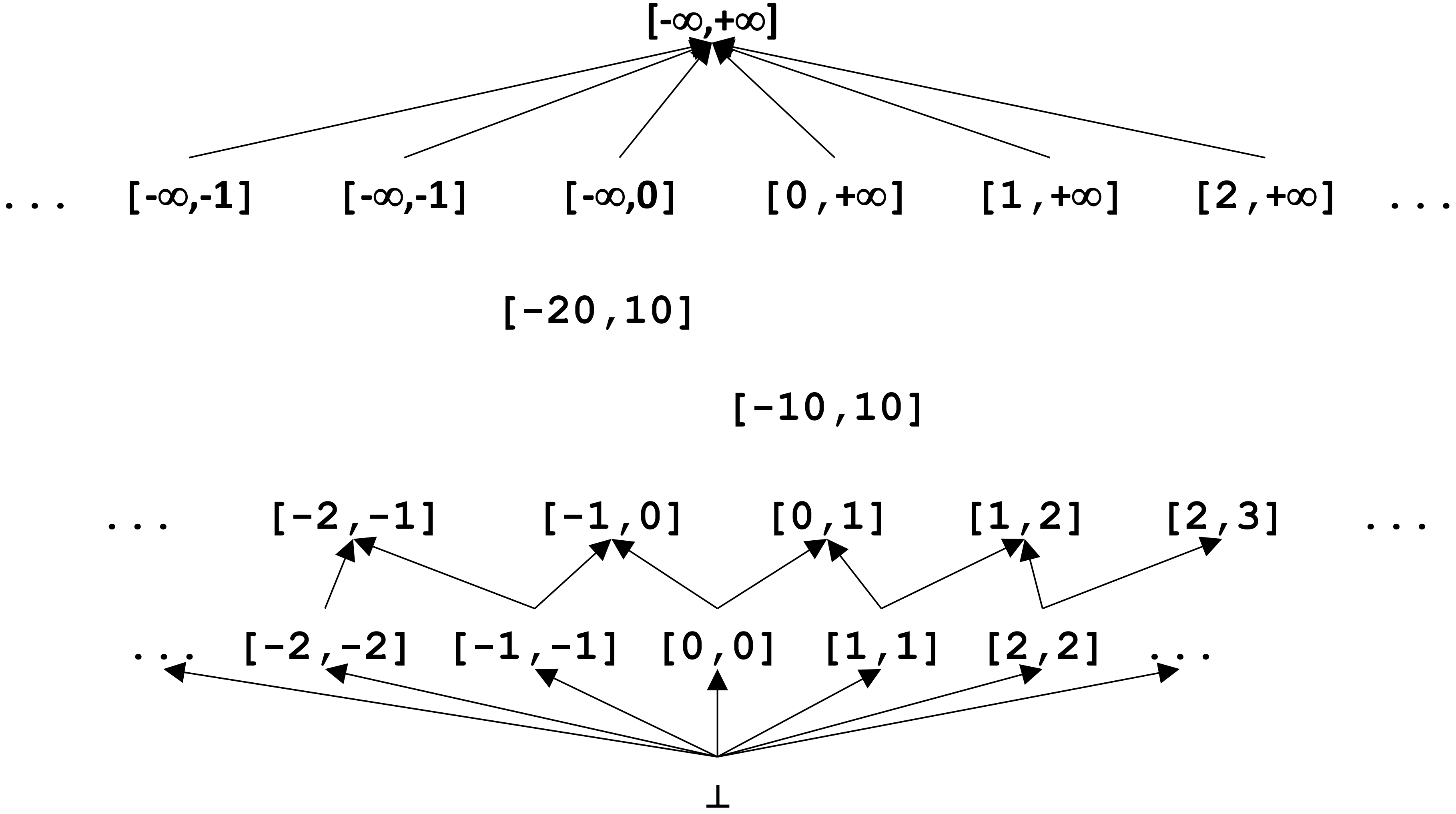
- Let $L = (D, \sqsubseteq, \sqcup, \bot)$ be a complete partial order and a **continuous** function $f: D \rightarrow D$ then

$$\mathrm{lfp}(f) = \bigsqcup_{n \in N} f^n(\bot)$$

- **Lemma:** Monotone functions on posets satisfying ACC are continuous

- **What if ACC does not hold?**

Intervals lattice for variable *x*



[-∞,+∞]

... [-∞,-1] [-∞,-1] [-∞,0] [0,+∞] [1,+∞] [2,+∞] ...

[-20,10]

[-10,10]

... [-2,-1] [-1,0] [0,1] [1,2] [2,3] ...

... [-2,-2] [-1,-1] [0,0] [1,1] [2,2] ...

⊥

Intervals lattice for variable *x*

- $D^{int}[x]$ = { (L,H) | L∈-∞,**Z** and H∈**Z**,+∞ and $L \leq H$}
- ⊥
- ⊤=[-∞,+∞]
- ⊑ = ?
  - [1,2] ⊑ [3,4] ?
  - [1,4] ⊑ [1,3] ?
  - [1,3] ⊑ [1,4] ?
  - [1,3] ⊑ [-∞,+∞] ?
- What is the lattice height?

Joining/meeting intervals

- $[a,b] \sqcup [c,d] = [\min(a,c), \max(b,d)]$
  - $[1,1] \sqcup [2,2] = [1,2]$
  - $[1,1] \sqcup [2,+\infty] = [1,+\infty]$
- $[a,b] \sqcap [c,d] = [\max(a,c), \min(b,d)]$ if a proper interval and otherwise $\perp$
  - $[1,2] \sqcap [3,4] = \perp$
  - $[1,4] \sqcap [3,4] = [3,4]$
  - $[1,1] \sqcap [1,+\infty] = [1,1]$
- Check that indeed $x \sqsubseteq y$ if and only if $x \sqcup y = y$

Interval domain for programs

- $D^{int}[x]$ = { (L,H) | $L \in -\infty,\mathbf{Z}$ and $H \in \mathbf{Z},+\infty$ and $L \leq H$}
- For a program with variables $Var = \{x_1,\ldots,x_k\}$
- $D^{int}[Var] = D^{int}[x_1] \times \ldots \times D^{int}[x_k]$
- How can we represent it in terms of formulas?
  - Two types of factoids $x \geq c$ and $x \leq c$
  - Example: $S = \bigwedge\{x \geq 9, y \geq 5, y \leq 10\}$
  - Helper operations
    - $c + +\infty = +\infty$
    - remove(S, x) = S without any x-constraints
    - lb(S, x) = k if $k \leq x \leq m$
    - ub(S,x) = m if $k \leq x \leq m$

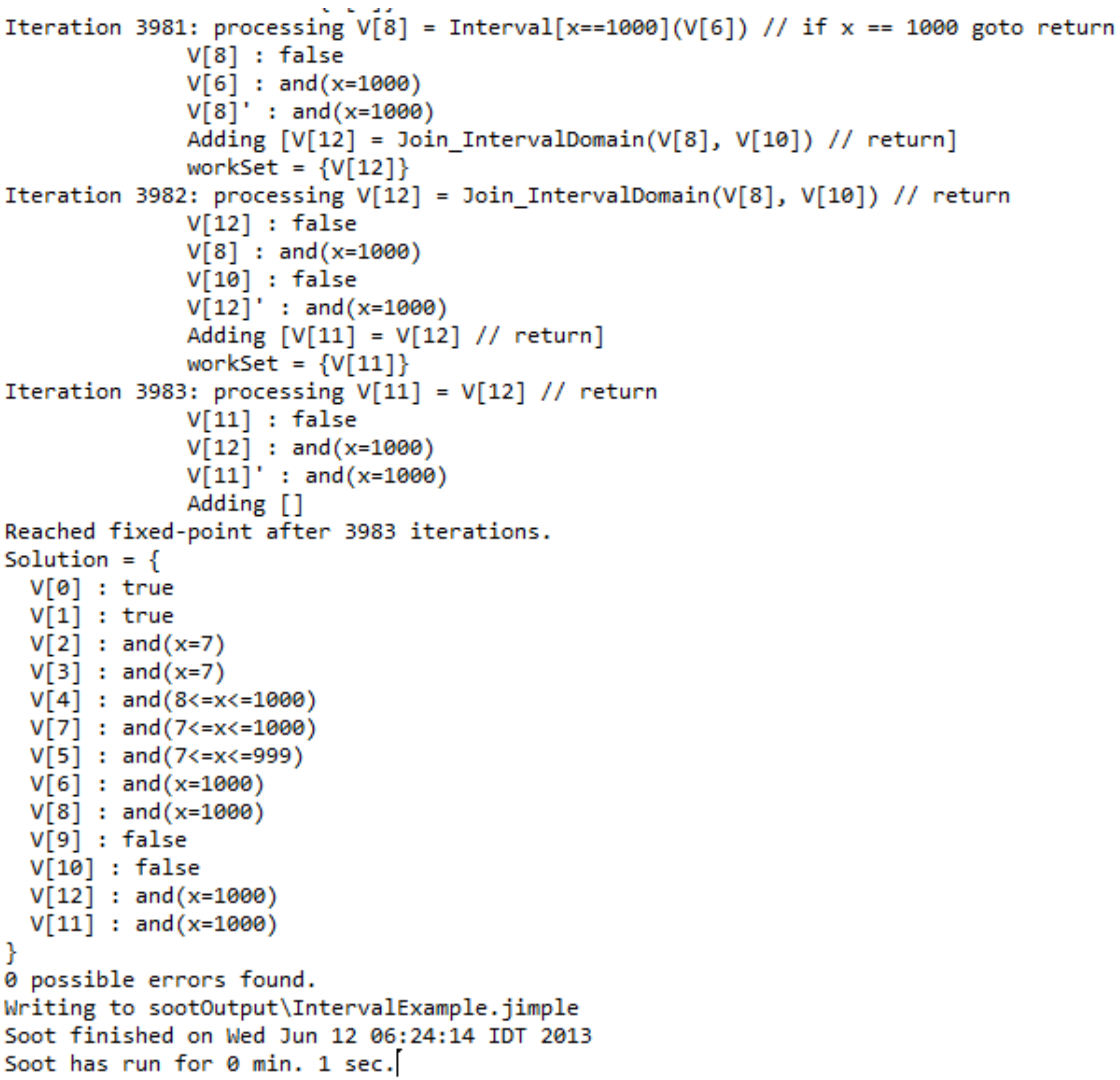
Assignment transformers

- $[\![x := c]\!]^\# S$ = remove(S,x) $\cup$ {$x \geq c$, $x \leq c$}
- $[\![x := y]\!]^\# S$ = remove(S,x) $\cup$ {$x \geq$lb(S,y), $x \leq$ub(S,y)}
- $[\![x := y+c]\!]^\# S$ = remove(S,x) $\cup$ {$x \geq$lb(S,y)+c, $x \leq$ub(S,y)+c}
- $[\![x := y+z]\!]^\# S$ = remove(S,x) $\cup$ {$x \geq$lb(S,y)+lb(S,z),
  $x \leq$ub(S,y)+ub(S,z)}
- $[\![x := y*c]\!]^\# S$ = remove(S,x) $\cup$ if c>0 {$x \geq$lb(S,y)*c, $x \leq$ub(S,y)*c}
  else {$x \geq$ub(S,y)*-c, $x \leq$lb(S,y)*-c}
- $[\![x := y*z]\!]^\# S$ = remove(S,x) $\cup$?

`assume` transformers

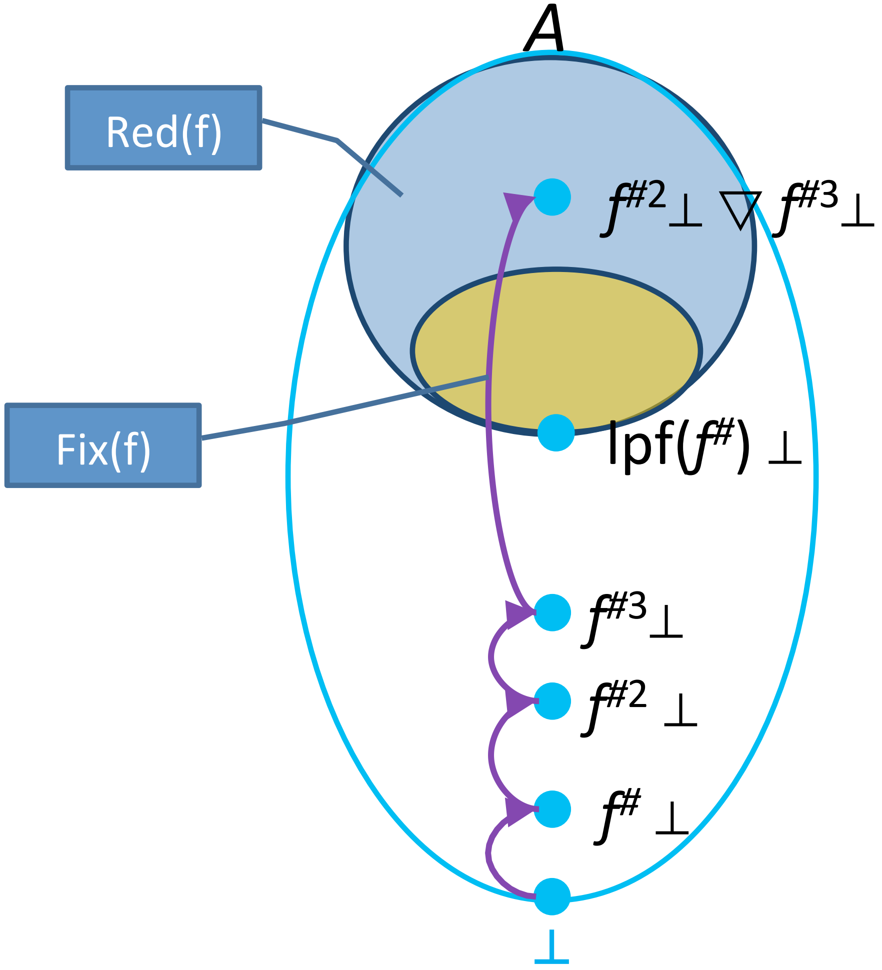
- $[\![$**`assume`** $x=c]\!]^\# S = S \sqcap \{x \geq c, x \leq c\}$
- $[\![$**`assume`** $x<c]\!]^\# S = S \sqcap \{x \leq c\text{-}1\}$
- $[\![$**`assume`** $x=y]\!]^\# S = S \sqcap \{x \geq \mathrm{lb}(S,y), x \leq \mathrm{ub}(S,y)\}$
- $[\![$**`assume`** $x \neq c]\!]^\# S = (S \sqcap \{x \leq c\text{-}1\}) \sqcup (S \sqcap \{x \geq c+1\})$

Too many iterations to converge

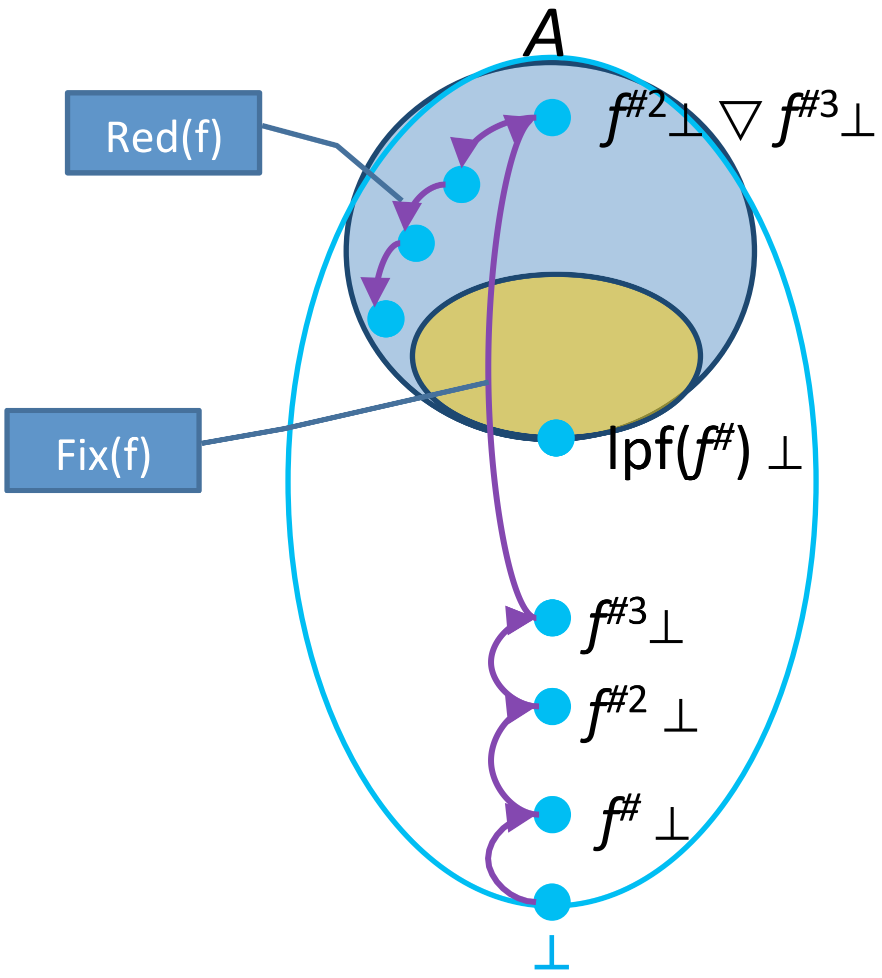


```
Iteration 3981: processing V[8] = Interval[x==1000](V[6]) // if x == 1000 goto return
              V[8] : false
              V[6] : and(x=1000)
              V[8]' : and(x=1000)
              Adding [V[12] = Join_IntervalDomain(V[8], V[10]) // return]
              workSet = {V[12]}
Iteration 3982: processing V[12] = Join_IntervalDomain(V[8], V[10]) // return
              V[12] : false
              V[8] : and(x=1000)
              V[10] : false
              V[12]' : and(x=1000)
              Adding [V[11] = V[12] // return]
              workSet = {V[11]}
Iteration 3983: processing V[11] = V[12] // return
              V[11] : false
              V[12] : and(x=1000)
              V[11]' : and(x=1000)
              Adding []
Reached fixed-point after 3983 iterations.
Solution = {
  V[0] : true
  V[1] : true
  V[2] : and(x=7)
  V[3] : and(x=7)
  V[4] : and(8<=x<=1000)
  V[7] : and(7<=x<=1000)
  V[5] : and(7<=x<=999)
  V[6] : and(x=1000)
  V[8] : and(x=1000)
  V[9] : false
  V[10] : false
  V[12] : and(x=1000)
  V[11] : and(x=1000)
}
0 possible errors found.
Writing to sootOutput\IntervalExample.jimple
Soot finished on Wed Jun 12 06:24:14 IDT 2013
Soot has run for 0 min. 1 sec.
```

Analysis with widening



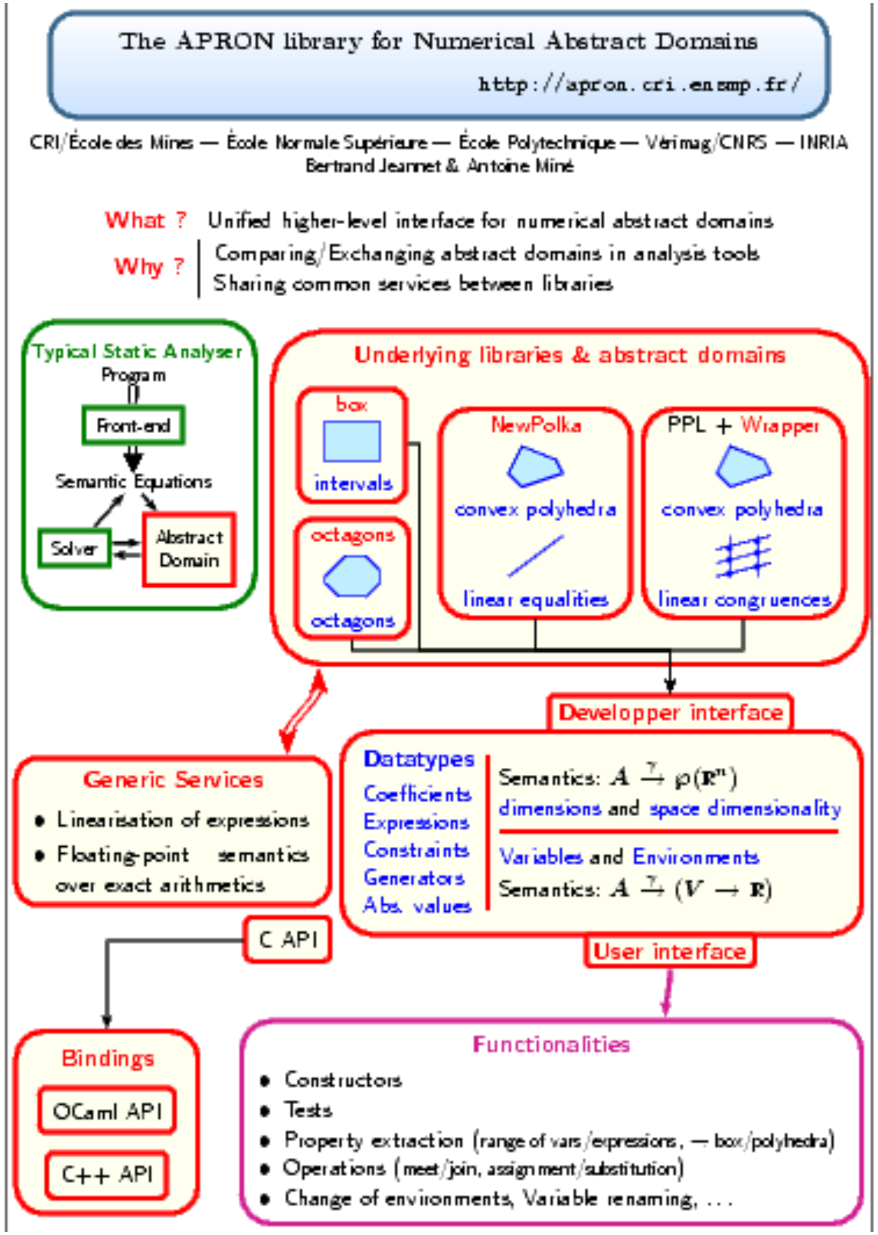
Analysis with narrowing



Overview

- Goal: infer numeric properties of program variables (integers, floating point)
- Applications
  - Detect division by zero, overflow, out-of-bound array access
  - Help non-numerical domains
- Classification
  - Non-relational
  - (Weakly-)relational
  - Equalities / Inequalities
  - Linear / non-linear
  - Exotic

Implementation



The APRON library for Numerical Abstract Domains

http://apron.cri.ensmp.fr/

CRI/École des Mines — École Normale Supérieure — École Polytechnique — Vérimag/CNRS — INRIA
Bertrand Jeannet & Antoine Miné

What ? Unified higher-level interface for numerical abstract domains

Why ? Comparing/Exchanging abstract domains in analysis tools
Sharing common services between libraries

Datatypes

- Coefficients
- Expressions
- Constraints
- Generators
- Abs. values

Semantics: $A \xrightarrow{\gamma} \wp(\mathbb{R}^n)$
dimensions and space dimensionality

Variables and Environments
Semantics: $A \xrightarrow{\gamma} (V \to \mathbb{R})$

Generic Services

- Linearisation of expressions
- Floating-point semantics over exact arithmetics

Bindings

- OCaml API
- C++ API

Functionalities

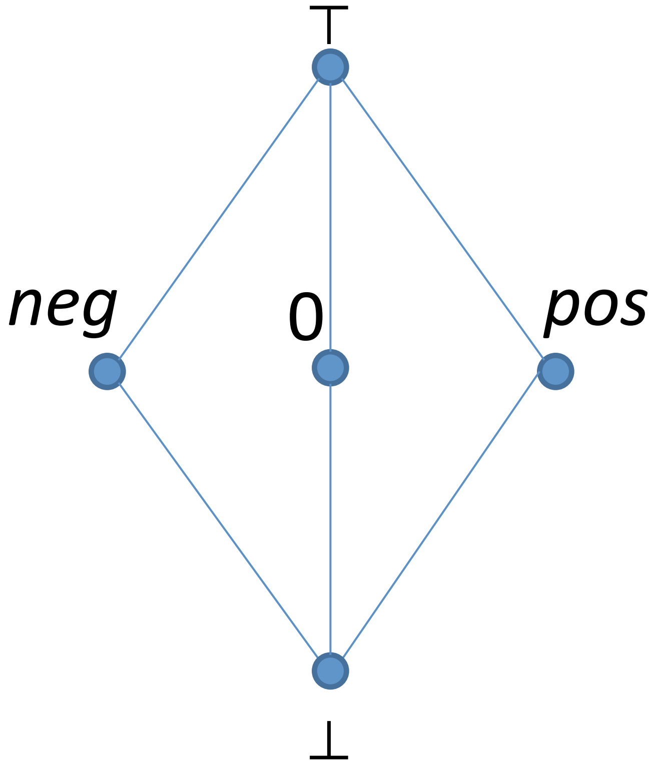
- Constructors
- Tests
- Property extraction (range of vars/expressions, → box/polyhedra)
- Operations (meet/join, assignment/substitution)
- Change of environments, Variable renaming, ...

Non-relational abstractions

- Abstract each variable individually
  - Constant propagation [Kildall'73]
  - Sign
  - Parity (congruences)
  - Intervals (Box)

Sign abstraction for variable *x*

- Concrete lattice: $C = (2^{\mathbf{State}}, \subseteq, \cup, \cap, \emptyset, \mathbf{State})$
- $Sign = \{\bot, neg, 0, pos, \top\}$
- $GC^{C,Sign}=(C, \alpha, \gamma, Sign)$
- $\gamma(\bot)$ = ?
- $\gamma(neg)$ = ?
- $\gamma(0)$ = ?
- $\gamma(pos)$ = ?
- $\gamma(\top)$ = ?
- How can we represent ≥0?

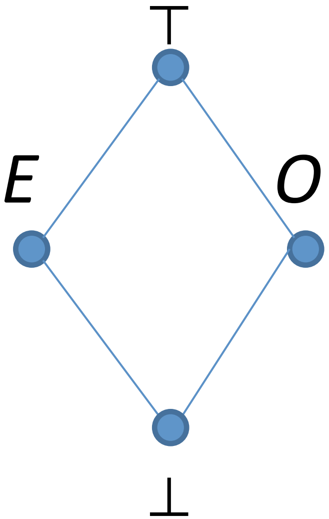


Transformer x:=y+z

	⊥	neg	0	pos	⊤
⊥	⊥	⊥	⊥	⊥	⊥
neg	⊥	neg	neg	⊤	⊤
0	⊥	neg	0	pos	⊤
pos	⊥	⊤	pos	pos	⊤
⊤	⊥	⊤	⊤	⊤	⊤

Parity abstraction for variable *x*

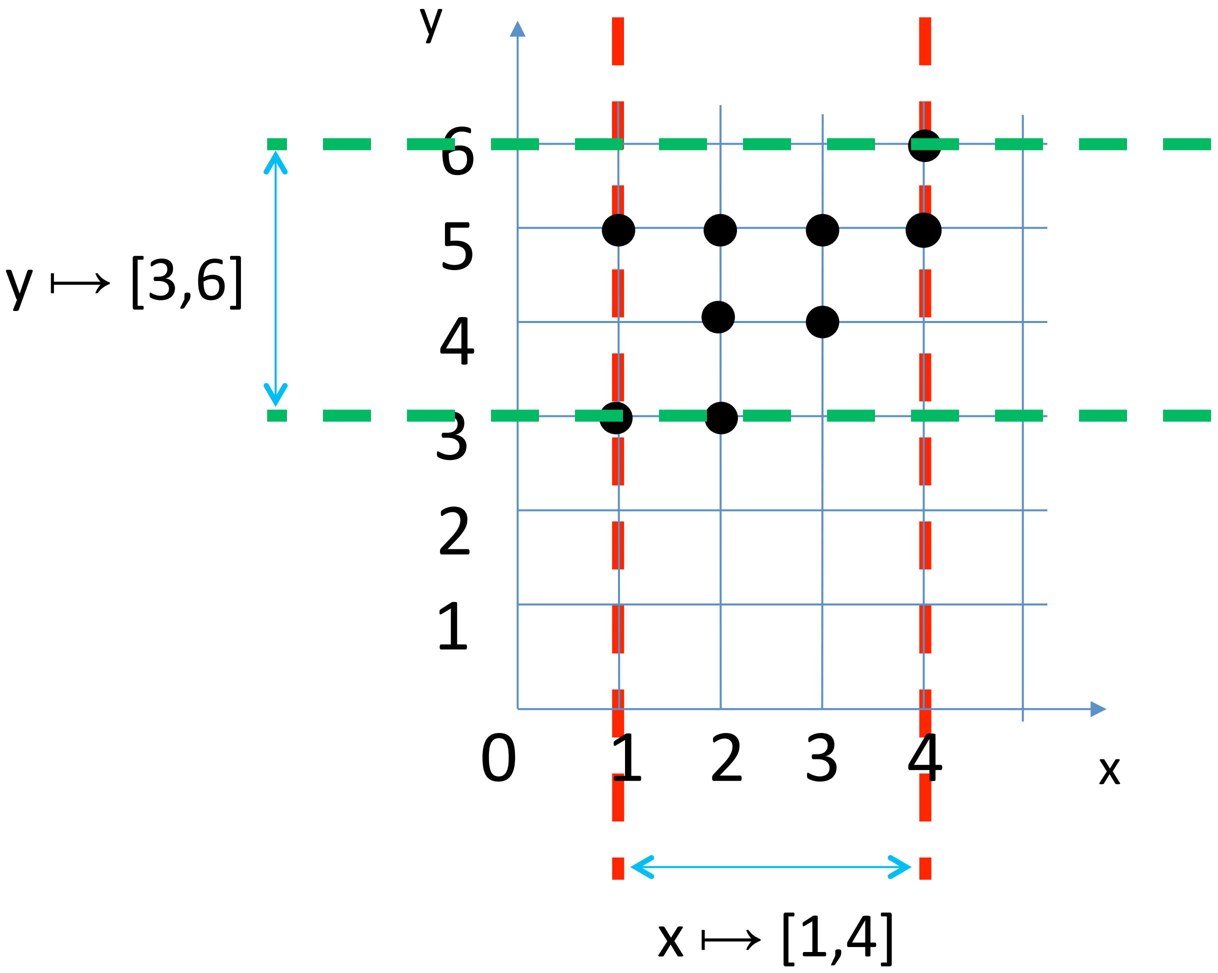
- Concrete lattice: $C = (2^{\textbf{State}}, \subseteq, \cup, \cap, \varnothing, \textbf{State})$
- $Parity = \{\bot, E, O, \top\}$
- $GC^{C,Parity}=(C, \alpha, \gamma, Parity)$
- $\gamma(\bot) = ?$
- $\gamma(E) = ?$
- $\gamma(O) = ?$
- $\gamma(\top) = ?$



Transformer x:=y+z

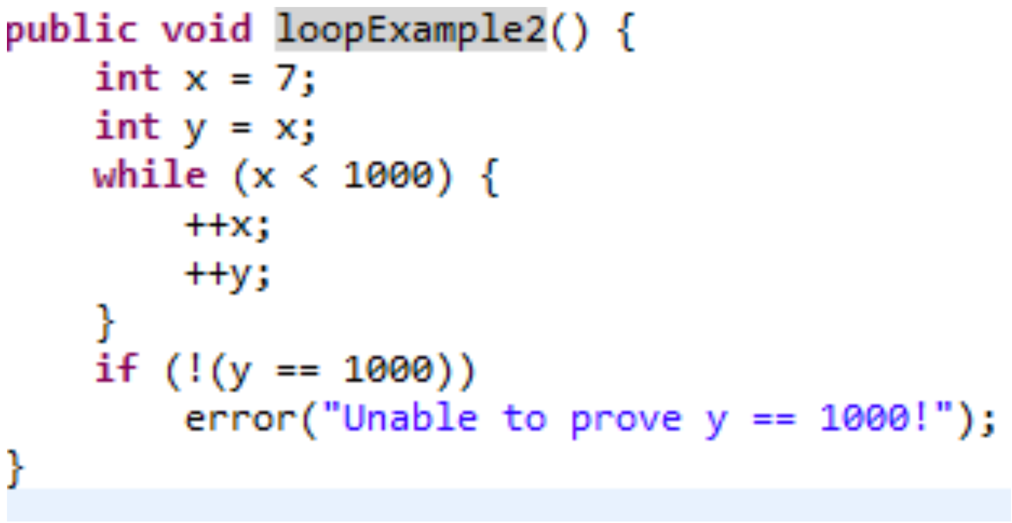
	$\bot$	E	O	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
E	$\bot$	E	O	$\top$
O	$\bot$	O	E	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$

Boxes (intervals)



Non-relational abstractions

- Cannot prove properties that hold simultaneous for several variables
  - x = 2*y
  - x ≤ y

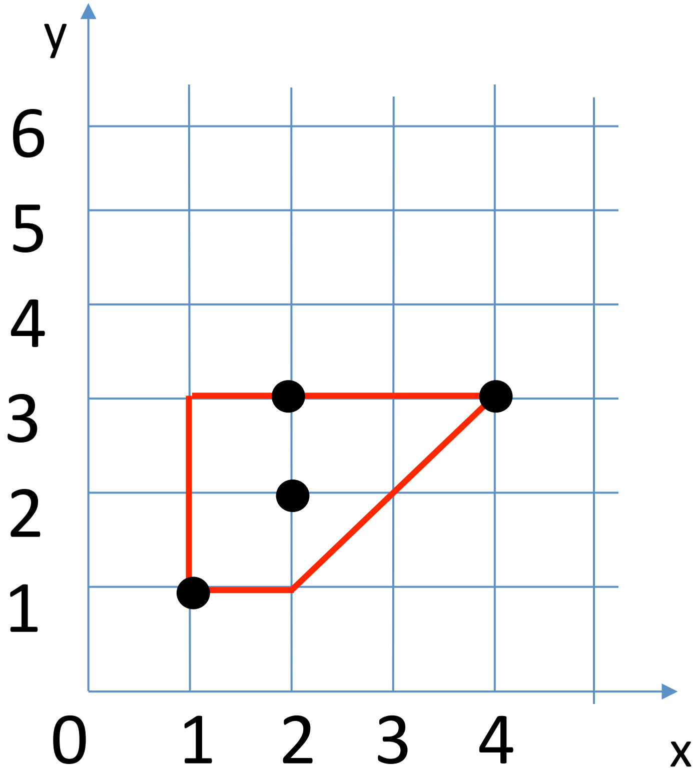


```
public void loopExample2() {
    int x = 7;
    int y = x;
    while (x < 1000) {
        ++x;
        ++y;
    }
    if (!(y == 1000))
        error("Unable to prove y == 1000!");
}
```

Zone abstraction [Mine]

- Maintain bounded differences between a pair of program variables (useful for tracking array accesses)
- Abstract state is a conjunction of linear inequalities of the form x-y≤c

$$\left\{\begin{array}{l} x \le 4 \\ -x \le -1 \\ y \le 3 \\ -y \le -1 \\ x - y \le 1 \end{array}\right.$$



Difference bound matrices

- Add a special V0 variable for the number 0
- Represent non-existent relations between variables by $+\infty$ entries
- Convenient for defining the partial order between two abstract elements… ⊑=?

$$\left\{\begin{array}{l} x \le 4 \\ -x \le -1 \\ y \le 3 \\ -y \le -1 \\ x - y \le 1 \end{array}\right.$$

	V0	x	y
V0	+∞	4	3
x	-1	+∞	+∞
y	-1	1	+∞

Difference bound matrices

- Add a special V0 variable for the number 0
- Represent non-existent relations between variables by +∞ entries
- Convenient for defining the partial order between two abstract elements… $\sqsubseteq$=?

$$\left\{\begin{array}{l} x \leq 4 \\ -x \leq -1 \\ y \leq 3 \\ -y \leq -1 \\ x - y \leq 1 \end{array}\right.$$

	V0	x	y
V0	+∞	4	3
x	-1	+∞	+∞
y	-1	1	+∞

Ordering DBMs

- How should we order $M_1 \sqsubseteq M_2$?

$$M_1 = \left\{ \begin{array}{l} x \leq 4 \\ -x \leq -1 \\ y \leq 3 \\ -y \leq -1 \\ x - y \leq 1 \end{array} \right.$$

	V0	x	y
V0	+∞	4	3
x	-1	+∞	+∞
y	-1	1	+∞

$$M_2 = \left\{ \begin{array}{l} x \leq 5 \\ -x \leq -1 \\ y \leq 3 \\ x - y \leq 1 \end{array} \right.$$

	V0	x	y
V0	+∞	5	3
x	-1	+∞	+∞
y	+∞	1	+∞

Widening DBMs

- How should we join $M_1 \sqcup M_2$?

$$M_1 = \left\{ \begin{array}{l} x \leq 4 \\ -x \leq -1 \\ y \leq 3 \\ -y \leq -1 \\ x - y \leq 1 \end{array} \right.$$

	V0	x	y
V0	+∞	4	3
x	-1	+∞	+∞
y	-1	1	+∞

$$M_2 = \left\{ \begin{array}{l} x \leq 2 \\ -x \leq -1 \\ y \leq 0 \\ x - y \leq 1 \end{array} \right.$$

	V0	x	y
V0	+∞	2	0
x	-1	+∞	+∞
y	+∞	1	+∞

Widening DBMs

- How should we widen $M_1 \bigtriangledown M_2$?

$$M_1 = \begin{cases} x \leq 4 \\ -x \leq -1 \\ y \leq 3 \\ -y \leq -1 \\ x - y \leq 1 \end{cases}$$

	V0	x	y
V0	+∞	4	3
x	-1	+∞	+∞
y	-1	1	+∞

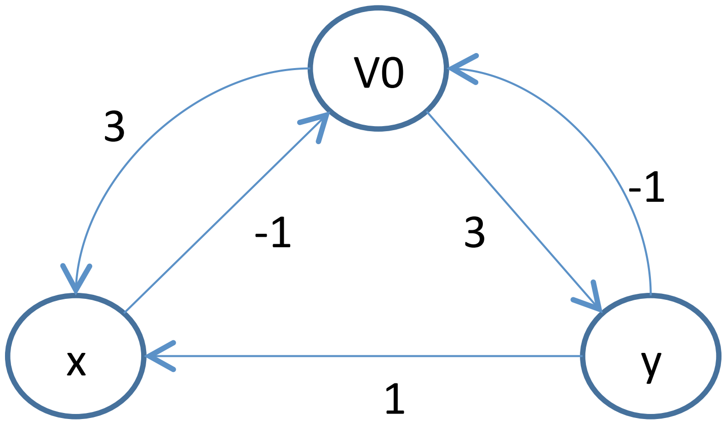
$$M_2 = \begin{cases} x \leq 5 \\ -x \leq -1 \\ y \leq 3 \\ x - y \leq 1 \end{cases}$$

	V0	x	y
V0	+∞	5	3
x	-1	+∞	+∞
y	+∞	1	+∞

Potential graph

- A vertex per variable
- A directed edge with the weight of the inequality
- Enables computing semantic reduction by shortest-path algorithms

$$\begin{cases} x \le 4 \\ -x \le -1 \\ y \le 3 \\ -y \le -1 \\ x - y \le 1 \end{cases}$$



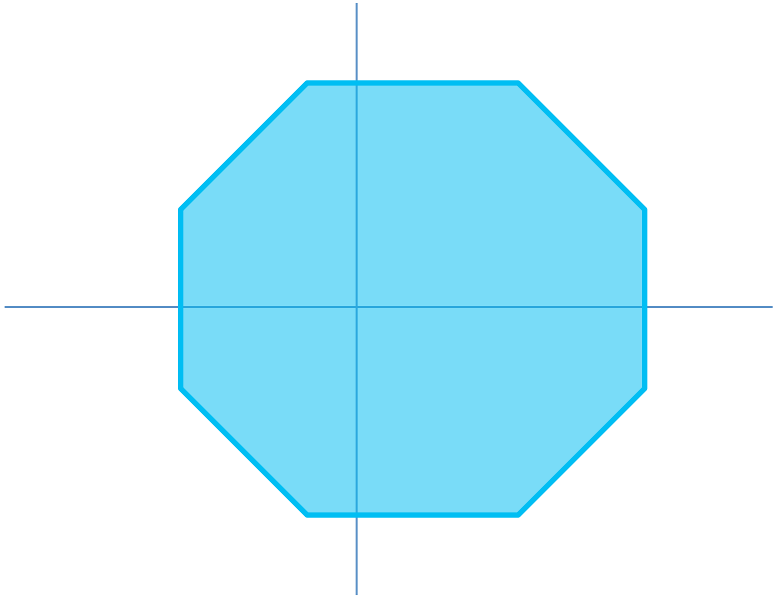
Can we tell whether a system of constraints is satisfiable?

Semantic reduction for zones

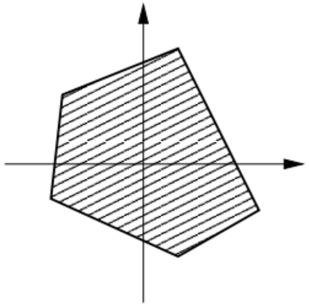
- Apply the following rule repeatedly
  $$\frac{x - y \leq c \qquad y - z \leq d}{x - z \leq c+d}$$
- When should we stop?
- Theorem 3.3.4. Best abstraction of potential sets and zones
  $m* = (\alpha^{Pot} \circ \gamma^{Pot})(m)$
- A word of caution: do not apply widening on top of semantic reduction (see 3.7.2)

Octagon abstraction [Mine-01]

- Abstract state is an intersection of linear inequalities of the form $\pm x \pm y \leq c$



Some inequality-based relational domains

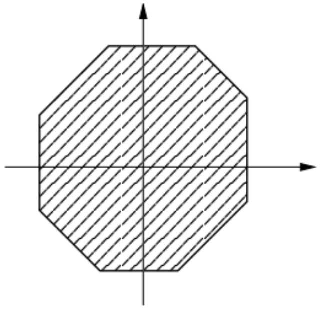


Polyhedra

$\sum_i \alpha_i X_i \geq \beta$

[Cousot-Halbwachs-78]

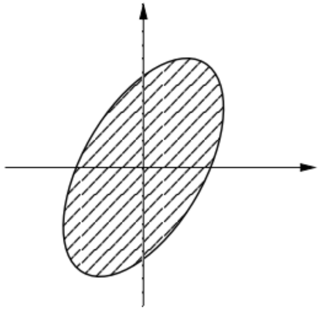
policy iteration



Octagons

$\pm X_i \pm X_j \leq \beta$

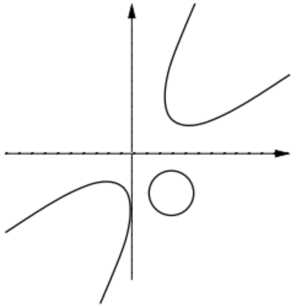
[Miné-01]



Ellipsoids

$X^2 + \beta Y^2 + \gamma XY \leq \delta$

[Feret-04]



Varieties

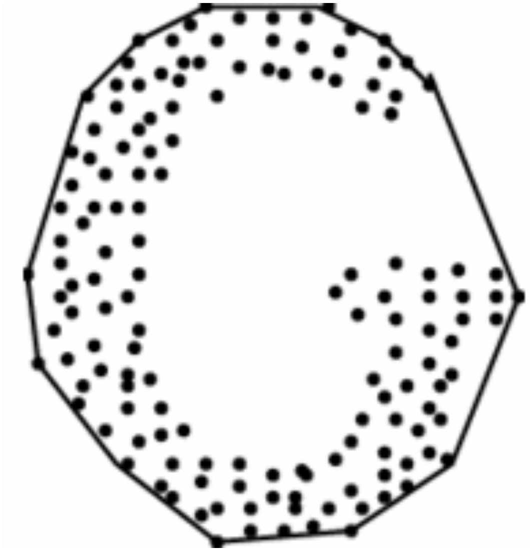
$P(\vec{X}) = 0,\ P \in \mathbb{R}[\mathrm{Var}]$

[Sankaranarayanan-Sipma-Man

Polyhedral Abstraction

- abstract state is an intersection of linear inequalities of the form $a_1x_2+a_2x_2+\ldots a_nx_n \leq c$

- represent a set of points by their convex hull



(image from http://www.cs.sunysb.edu/~algorith/files/convex-hull.shtml)

Operations on Polyhedra

Equality-based domains

- Simple congruences [Granger’89]: $y=a \bmod k$
- **Linear relations:** $y=a*x+b$
  - Join operator a little tricky
- Linear equalities [Karr’76]: $a_1*x_1+\ldots+a_k*x_k = c$
- Polynomial equalities:
  $a_1*x_1^{d1}*\ldots*x_k^{dk} + b_1*y_1^{z1}*\ldots*y_k^{zk} + \ldots = c$
  - Some good results are obtainable when $d_1+\ldots+d_k < n$ for some small n

Pointer Analysis

Constant propagation example

```
x = 3;

y = 4;

z = x + 5;
```

Constant propagation example with pointers

```
x = 3;

*p = 4;

z = x + 5;
```

Is x always 3 here?

Constant propagation example with pointers

pointers affect
most program analyses

```
p = &y;
x = 3;
*p = 4;
z = x + 5;
```

x is always 3

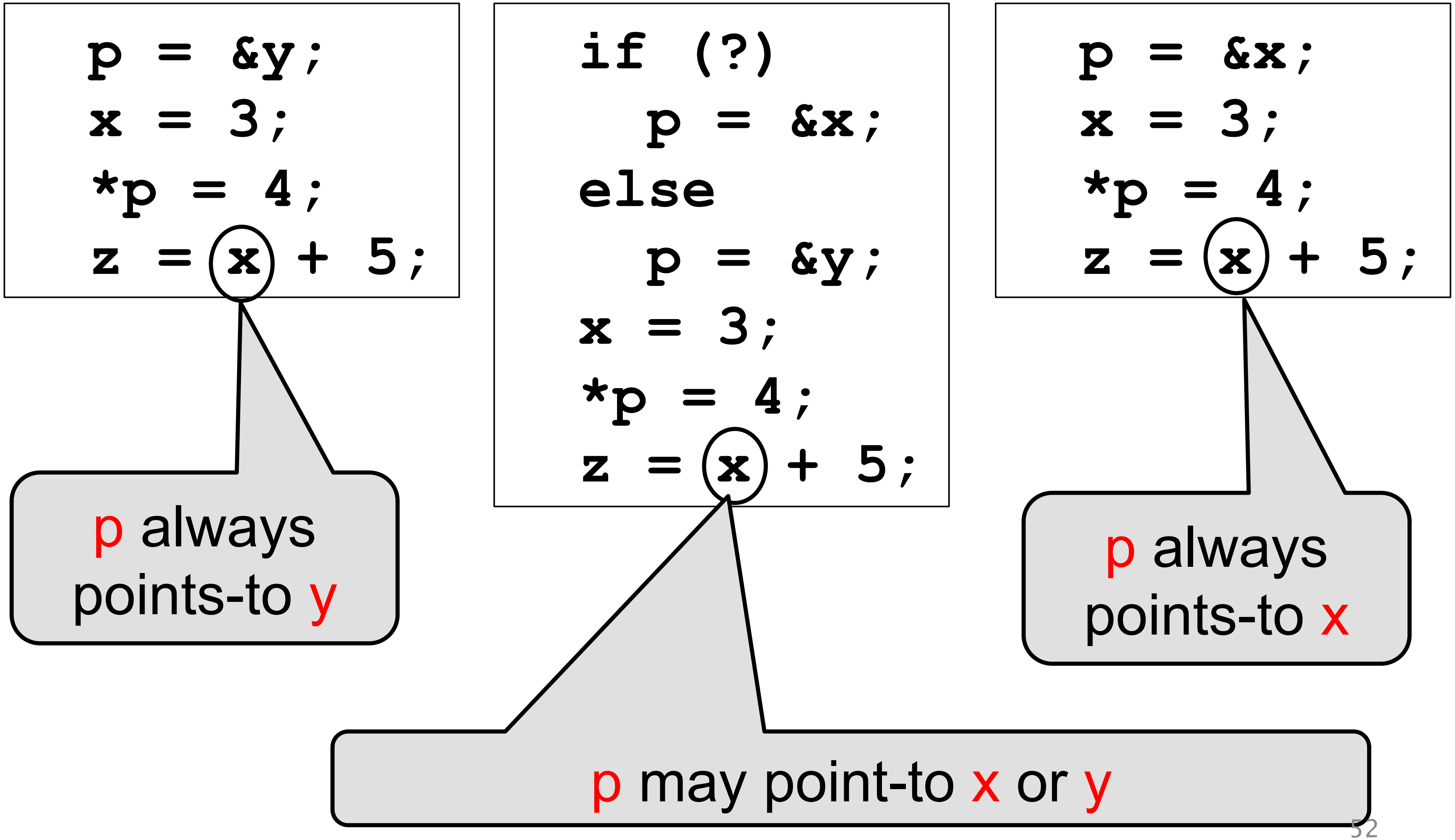
```
else
   p = &y;
x = 3;
*p = 4;
z = x + 5;
```

x may be 3 or 4
(i.e., x is unknown in our lattice)

```
= &x;
= 3;
*p = 4;
z = x + 5;
```

x is always 4

Constant propagation example with pointers



```
p = &y;
x = 3;
*p = 4;
z = x + 5;
```

p always points-to y

```
if (?)
  p = &x;
else
  p = &y;
x = 3;
*p = 4;
z = x + 5;
```

p may point-to x or y

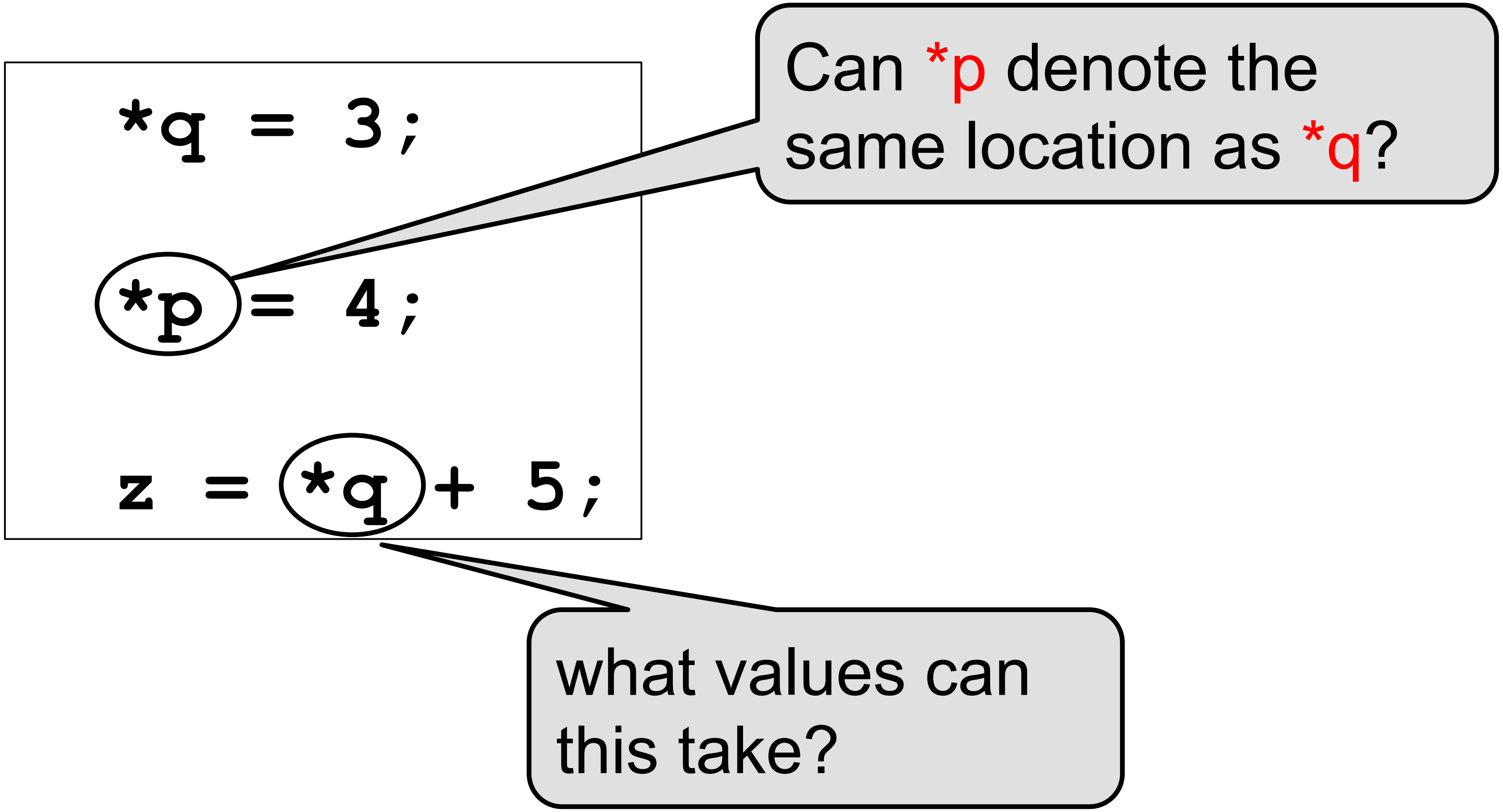
```
p = &x;
x = 3;
*p = 4;
z = x + 5;
```

p always points-to x

Points-to Analysis

- Determine the set of targets a pointer variable could point-to (at different points in the program)
  - "p points-to x"
    - "p stores the value &x"
    - "*p denotes the location x"
  - targets could be variables or locations in the heap (dynamic memory allocation)
    - p = &x;
    - p = new Foo(); or p = malloc (…);
  - must-point-to vs. may-point-to

Constant propagation example with pointers



```
*q = 3;

*p = 4;

z = *q + 5;
```

Can *p denote the same location as *q?

what values can this take?

More terminology

- *p and *q are said to be aliases (in a given concrete state) if they represent the same location
- Alias analysis
  - Determine if a given pair of references could be aliases at a given program point
  - *p may-alias *q
  - *p must-alias *q

Pointer Analysis

- Points-To Analysis
  - may-point-to
  - must-point-to

- Alias Analysis
  - may-alias
  - must-alias

Applications

- Compiler optimizations
  - Method de-virtualization
  - Call graph construction
  - Allocating objects on stack via escape analysis

- Verification & Bug Finding
  - Datarace detection
  - Use in preliminary phases
  - Use in verification itself

Points-to analysis: a simple example

```
p = &x;      {p=&x}
q = &y;      {p=&x ∧ q=&y}
if (?) {
  q = p;     {p=&x ∧ q=&x}
}            {p=&x ∧ (q=&y ∨ q=&x)}
x = &a;      {p=&x ∧ (q=&y ∨ q=&x) ∧ x=&a}
y = &b;      {p=&x ∧ (q=&y ∨ q=&x) ∧ x=&a ∧ y=&b}
z = *q;      {p=&x ∧ (q=&y∨q=&x) ∧ x=&a ∧ y=&b ∧ (z=x∨z=y)}
```

We will usually drop variable-equality information

How would you construct an abstract domain to represent these abstract states?

Points-to lattice

- **Points-to**
  - *PT-factoids*[x] = { x=&y | $y \in$ Var} $\cup$ false
    PT[x] = ($2^{\textit{PT-factoids}}$, $\subseteq$, $\cup$, $\cap$, *false*, *PT-factoids*[x])
    (interpreted disjunctively)
- How should combine them to get the abstract states in the example?

  `{p=&x ∧ (q=&y∨q=&x) ∧ x=&a ∧ y=&b}`

Points-to lattice

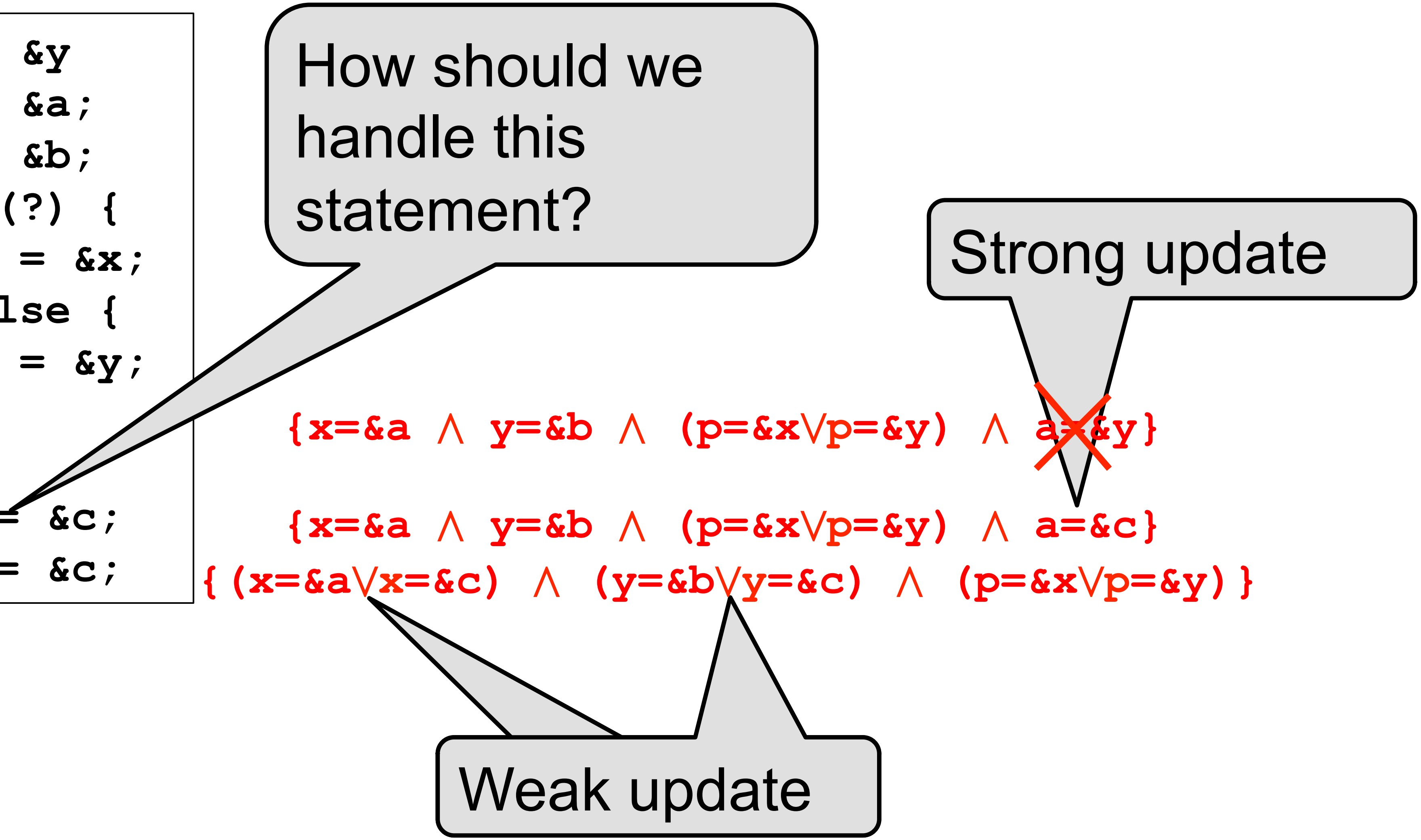
- **Points-to**
  - $\mathit{PT\text{-}factoids}[\mathrm{x}] = \{\, x{=}\&y \mid y \in \mathrm{Var}\} \cup \mathrm{false}$
    $\mathit{PT}[\mathrm{x}] = (2^{\mathit{PT\text{-}factoids}}, \subseteq, \cup, \cap, \mathit{false}, \mathit{PT\text{-}factoids}[\mathrm{x}])$
    (interpreted disjunctively)
- How should combine them to get the abstract states in the example?

  `{p=&x ∧ (q=&y∨q=&x) ∧ x=&a ∧ y=&b}`
- $D[\mathrm{x}] = \mathrm{Disj}(\mathit{VE}[\mathrm{x}]) \times \mathrm{Disj}(\mathit{PT}[\mathrm{x}])$
- For all program variables: $D = D[\mathrm{x}_1] \times \ldots \times D[\mathrm{x}_k]$

Points-to analysis

```
a = &y
x = &a;
y = &b;
if (?) {
  p = &x;
} else {
  p = &y;
}

*x = &c;
*p = &c;
```



How should we handle this statement?

Strong update

{x=&a ∧ y=&b ∧ (p=&x∨p=&y) ∧ ~~a=&y~~}

{x=&a ∧ y=&b ∧ (p=&x∨p=&y) ∧ a=&c}

{(x=&a∨x=&c) ∧ (y=&b∨y=&c) ∧ (p=&x∨p=&y)}

Weak update

Questions

- When is it correct to use a strong update? A weak update?
- Is this points-to analysis precise?
- What does it mean to say
  - p must-point-to x at program point u
  - p may-point-to x at program point u
  - p must-not-point-to x at program u
  - p may-not-point-to x at program u

Points-to analysis, formally

- We must formally define what we want to compute before we can answer many such questions

PWhile syntax

- A primitive statement is of the form
  - x := null
  - x := y
  - x := *y
  - x := &y;
  - *x := y
  - skip

  (where x and y are variables in Var)

Omitted (for now)
- Dynamic memory allocation
- Pointer arithmetic
- Structures and fields
- Procedures

PWhile operational semantics

- **State** : (Var⟶Z) ∪ (Var⟶Var∪{null})
- ⟦ x = y ⟧ s =
- ⟦ x = *y ⟧ s =
- ⟦ *x = y ⟧ s =
- ⟦ x = null ⟧ s =
- ⟦ x = &y ⟧ s =

PWhile operational semantics

- **State** : $(\mathrm{Var} \rightarrow Z) \cup (\mathrm{Var} \rightarrow \mathrm{Var} \cup \{\mathrm{null}\})$
- $\llbracket\, x = y \,\rrbracket\, s \quad = s[x \mapsto s(y)]$
- $\llbracket\, x = {*y} \,\rrbracket\, s = s[x \mapsto s(s(y))]$
- $\llbracket\, {*x} = y \,\rrbracket\, s = s[s(x) \mapsto s(y)]$
- $\llbracket\, x = \mathrm{null} \,\rrbracket\, s \quad = s[x \mapsto \mathrm{null}]$
- $\llbracket\, x = \&y \,\rrbracket\, s = s[x \mapsto y]$

must say what happens if null is dereferenced

PWhile collecting semantics

- *CS*[*u*] = set of concrete states that can reach program point *u* (CFG node)

Ideal PT Analysis: formal definition

- Let *u* denote a node in the CFG

- Define IdealMustPT(*u*) to be

  { (*p*,*x*) | **forall** *s* in *CS*[*u*]. *s*(*p*) = *x* }

- Define IdealMayPT(*u*) to be

  { (*p*,*x*) | **exists** *s* in *CS*[*u*]. *s*(*p*) = x }

May-point-to analysis: formal Requirement specification

May Point-To Analysis

Compute R: V -> $2^{Vars'}$ such that
R(u) $\sim$ IdealMayPT(u)
(where Var' = Var U {null})

For every vertex u in the CFG,
compute a set R(u) such that
R(u) $\sim$ $\{ (p,x) \mid \exists s \in CS[u].\ s(p) = x \}$

May-point-to analysis: formal Requirement specification

Compute R: V -> $2^{Vars'}$ such that
R(u) ~ IdealMayPT(u)

- An algorithm is said to be correct if the solution R it computes satisfies

$$\forall u \in V.\ R(u) \sim \text{IdealMayPT}(u)$$

- An algorithm is said to be precise if the solution R it computes satisfies

$$\forall u \in V.\ R(u) = \text{IdealMayPT}(u)$$

- An algorithm that computes a solution R_1 is said to be more precise than one that computes a solution R_2 if

$$\forall u \in V.\ R_1(u) \subseteq R_2(u)$$

(May-point-to analysis) *Algorithm A*

- Is this algorithm correct?
- Is this algorithm precise?

- Let’s first completely and formally define the algorithm

Points-to graphs

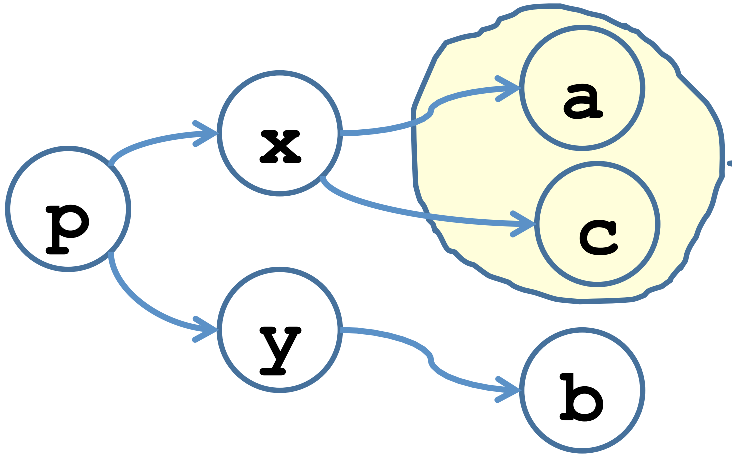
```
x = &a;
y = &b;
if (?) {
  p = &x;
} else {
  p = &y;
}

*x = &c;
*p = &c;
```

{x=&a ∧ y=&b ∧ (p=&x∨p=&y)}

{x=&a ∧ y=&b ∧ (p=&x∨p=&y) ∧ a=&c}

{(x=&a∨x=&c) ∧ (y=&b∨y=&c) ∧ (p=&x∨p=&y)}



The points-to set of **x**

Algorithm A: A formal definition the "Data Flow Analysis" Recipe

- Define join-semilattice of abstract-values
  - PTGraph ::= (Var, Var$\times$Var')
  - $g_1 \sqcup g_2$ = ?
  - $\bot$ = ?
  - $\top$ = ?
- Define transformers for primitive statements
  - $[\![\text{stmt}]\!]^\#$: PTGraph $\rightarrow$ PTGraph

Algorithm A: A formal definition the "Data Flow Analysis" Recipe

- Define join-semilattice of abstract-values
  - PTGraph ::= (Var, Var$\times$Var')
  - $g_1 \sqcup g_2$ = (Var, $E_1 \cup E_2$)
  - $\perp$ = (Var, {})
  - $\top$ = (Var, Var$\times$Var')
- Define transformers for primitive statements
  - $[\![\text{stmt}]\!]^{\#}$: PTGraph $\rightarrow$ PTGraph

Algorithm A: transformers

- Abstract transformers for primitive statements
  - $\llbracket$ stmt $\rrbracket^{\#}$: PTGraph → PTGraph
- $\llbracket$ x := y $\rrbracket^{\#}$ (Var, E) = ?
- $\llbracket$ x := null $\rrbracket^{\#}$ (Var, E) = ?
- $\llbracket$ x := &y $\rrbracket^{\#}$ (Var, E) = ?
- $\llbracket$ x := *y $\rrbracket^{\#}$ (Var, E) = ?
- $\llbracket$ *x := &y $\rrbracket^{\#}$ (Var, E) = ?

Algorithm A: transformers

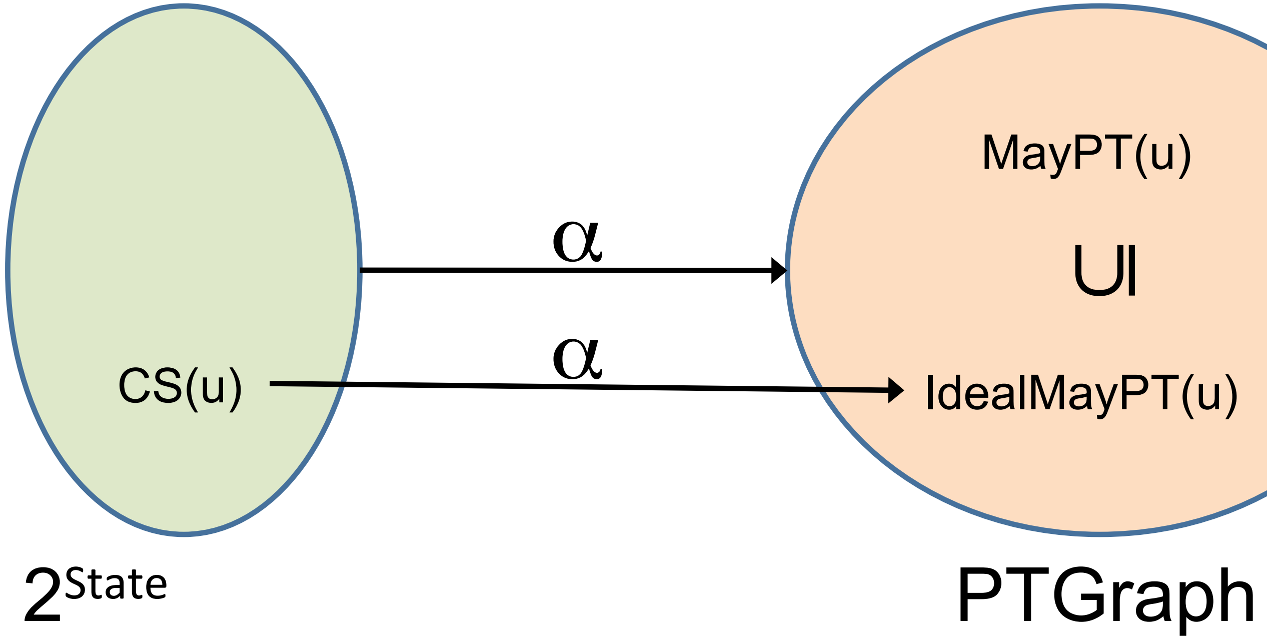
- Abstract transformers for primitive statements
  - $[\![\text{stmt}]\!]^\#$: PTGraph → PTGraph
- $[\![x := y]\!]^\#$ (Var, E) = (Var, E[succ(x)=succ(y)]
- $[\![x := \text{null}]\!]^\#$ (Var, E) = (Var, E[succ(x)={null}]
- $[\![x := \&y]\!]^\#$ (Var, E) = (Var, E[succ(x)={y}]
- $[\![x := *y]\!]^\#$ (Var, E) = (Var, E[succ(x)=succ(succ(y))]
- $[\![*x := \&y]\!]^\#$ (Var, E) = ???

Correctness & precision

- We have a complete & formal definition of the problem
- We have a complete & formal definition of a proposed solution

- How do we reason about the correctness & precision of the proposed solution?

Points-to analysis (abstract interpretation)



α(Y) = { (p,x) | exists s in Y. s(p) = x }

IdealMayPT (u) = α (CS(u))

Concrete transformers

- CS[stmt] : State → State
- $[\![x = y]\!] s = s[x \mapsto s(y)]$
- $[\![x = *y]\!] s = s[x \mapsto s(s(y))]$
- $[\![*x = y]\!] s = s[s(x) \mapsto s(y)]$
- $[\![x = \text{null}]\!] s = s[x \mapsto \text{null}]$
- $[\![x = \&y]\!] s = s[x \mapsto y]$

- CS*[stmt] : $2^{\text{State}} \rightarrow 2^{\text{State}}$
- CS*[st] X = { CS[st]s | s ∈ X }

Shape Analysis

Shape Analysis

Automatically verify properties of programs manipulating dynamically allocated storage

Identify all possible shapes (layout) of the heap

Sequential Stack

```
void push (int v) {
 Node *x = malloc(sizeof(Node));
 x->d = v;
 x->n = Top;
 Top = x;
}

 int pop() {
  if (Top == NULL) return EMPTY;
  Node *s = Top->n;
  int r = Top->d;
  Top = s;
  return r;
 }
```

Want to Verify

No Null Dereference

Underlying list remains acyclic after each operation

Shape Analysis via 3-valued Logic

1) Abstraction

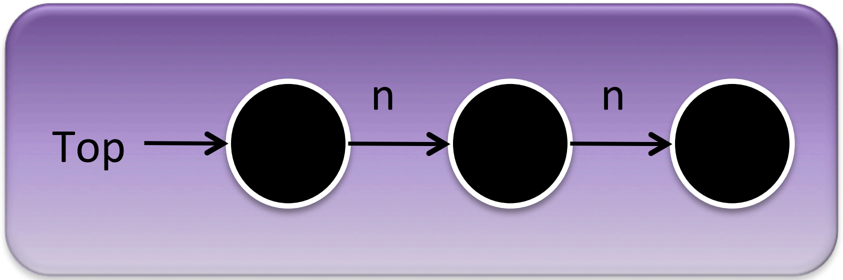
- 3-valued logical structure
- canonical abstraction

2) Transformers

- via logical formulae
- soundness by construction
  - embedding theorem, [SRW02]

Concrete State

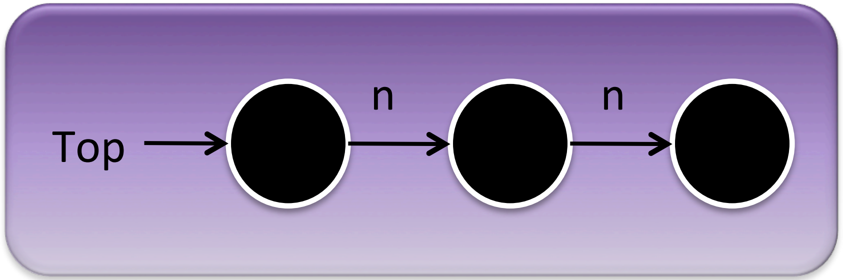
- represent a concrete state as a two-valued logical structure
  - Individuals = heap allocated objects
  - Unary predicates = object properties
  - Binary predicates = relations
- parametric vocabulary



(storeless, no heap addresses)

Concrete State

- S = <U, ι > over a vocabulary P
- U – universe
- ι - interpretation, mapping each predicate from p to its truth value in S



- U = { u1, u2, u3}
- P = { Top, n }
- ι(n)(u1,u2) = 1, ι(n)(u1,u3)=0, ι(n)(u2,u1)=0,…
- ι(Top)(u1)=1, ι(Top)(u2)=0, ι(Top)(u3)=0

Formulae for Observing Properties

```
void push (int v) {
 Node *x =
   malloc(sizeof(Node));
```

$\exists w: x(w)$

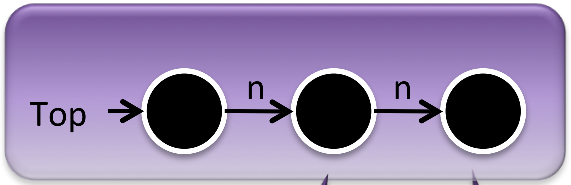
$\exists w: x(w)$

```
 Top = x;

}
```

$\neg\exists v1,v2: n(v1, v2) \land n^*(v2, v1)$

$\neg\exists v1,v2: n(v1, v2) \land Top(v2)$



Top != null

$\exists w:Top(w)$ 1

No node precedes Top

$\neg\exists v1,v2: n(v1, v2) \land Top(v2)$ 1

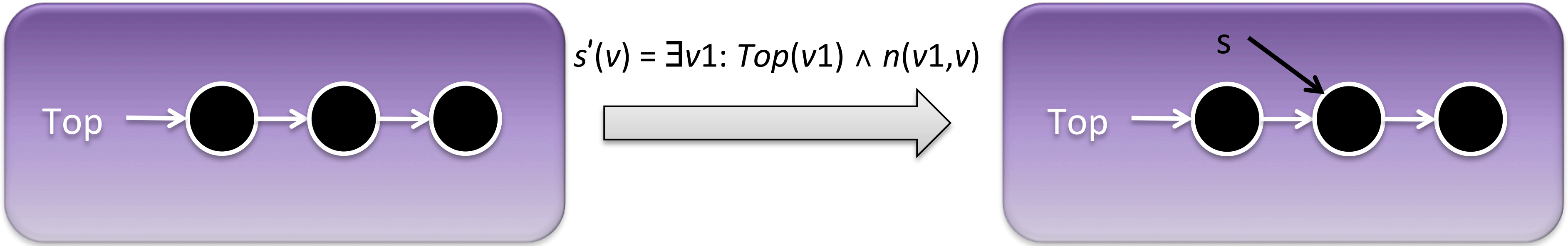
No Cycles

$\neg\exists v1,v2: n(v1, v2) \land n^*(v2, v1)$ 1

Concrete Interpretation Rules

Statement	Update formula
x =NULL	$x'(v)= 0$
x= malloc()	$x'(v) = \mathit{IsNew}(v)$
x=y	$x'(v)= y(v)$
x=y →next	$x'(v)= \exists w: y(w) \land n(w, v)$
x →next=y	$n'(v, w) = (\neg x(v) \land n(v, w)) \lor (x(v) \land y(w))$

Example: $s = Top \rightarrow n$



$s'(v) = \exists v1: Top(v1) \wedge n(v1,v)$

Top	
u1	1
u2	0
u3	0

n	u1	u2	U3
u1	0	1	0
u2	0	0	1
u3	0	0	0

s	
u1	0
u2	0
u3	0

Top	
u1	1
u2	0
u3	0

n	u1	u2	U3
u1	0	1	0
u2	0	0	1
u3	0	0	0

s	
u1	0
u2	1
u3	0

Collecting Semantics

$$
\mathrm{CSS}\,[v] = \begin{cases}
\{ <\emptyset,\emptyset> \} & \text{if } v = \text{entry} \\[2ex]
\displaystyle\bigcup_{\substack{(w,v) \in E(G), \\ w \in \mathrm{Assignments}(G)}} \{ \llbracket \mathrm{st}(w) \rrbracket (S) \mid S \in \mathrm{CSS}[w] \} \cup & \\[2ex]
\displaystyle\bigcup_{\substack{(w,v) \in E(G), \\ w \in \mathrm{Skip}(G)}} \{ S \mid S \in \mathrm{CSS}[w] \} \cup & \\[2ex]
\displaystyle\bigcup_{(w,v) \in \text{True-Branches}(G)} \{ S \mid S \in \mathrm{CSS}[w] \text{ and } S \vDash \mathrm{cond}(w) \} \cup & \text{othrewise} \\[2ex]
\displaystyle\bigcup_{(w,v) \in \text{False-Branches}(G)} \{ S \mid S \in \mathrm{CSS}[w] \text{ and } S \vDash \neg \mathrm{cond}(w) \} &
\end{cases}
$$

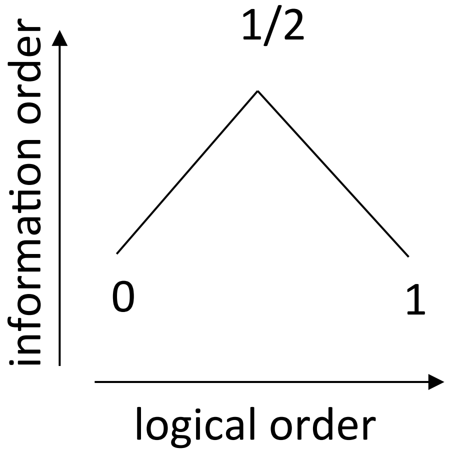
Collecting Semantics

- At every program point – a potentially infinite set of two-valued logical structures
- Representing (at least) all possible heaps that can arise at the program point

- Next step:
  find a bounded abstract representation

3-Valued Logic

- 1 = true
- 0 = false
- 1/2 = unknown



- A join semi-lattice, $0 \sqcup 1 = 1/2$

3-Valued Logical Structures

- A set of individuals (nodes) *U*
- Relation meaning
  - Interpretation of relation symbols in *P*
    $p^0() \rightarrow \{0,1, \mathit{1/2}\}$
    $p^1(v) \rightarrow \{0,1, \mathit{1/2}\}$
    $p^2(u,v) \rightarrow \{0,1, \mathit{1/2}\}$
- A join semi-lattice: $0 \sqcup 1 = 1/2$

Boolean Connectives [Kleene]

∧	0	1/2	1
0	0	0	0
1/2	0	1/2	1/2
1	0	1/2	1

∨	0	1/2	1
0	0	1/2	1
1/2	1/2	1/2	1
1	1	1	1

Property Space

- 3-struct[P] = the set of 3-valued logical structures over a vocabulary (set of predicates) P

- Abstract domain
  - $\wp$(3-Struct[P])
  - $\sqsubseteq$ is $\subseteq$
    - We will see alternatives later (maybe)

Embedding Order

- Given two structures S = <U,ι >, S' = <U', ι'> and an onto function f : U $\rightarrow$ U' mapping individuals in U to individuals in U'
- We say that f embeds S in S' (denoted by S $\sqsubseteq$ S') if
  - for every predicate symbol p $\in$ P of arity k: u1, …, uk $\in$ U, ι(p)(u1, ..., uk) $\sqsubseteq$ ι'(p)(f(u1), ..., f (uk))
  - and for all u' $\in$ U'
    (| { u | f (u) = u' } | > 1) $\sqsubseteq$ ι'(sm)(u')
- We say that S can be embedded in S' (denoted by S $\sqsubseteq$ S') if there exists a function f such that S $\sqsubseteq^{f}$ S'

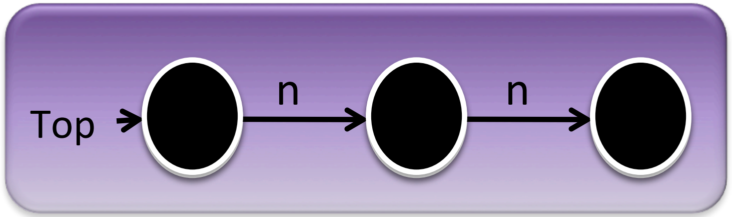
Tight Embedding

- S’ = <U’, ι’> is a tight embedding of S=< U, ι > with respect to a function f if:
  - S’ does not lose unnecessary information

$$\iota'(u'_1,\ldots, u'_k) = \sqcup\{\iota(u_1 \ldots, u_k) \mid f(u_1)=u'_1,\ldots, f(u_k)=u'_k\}$$

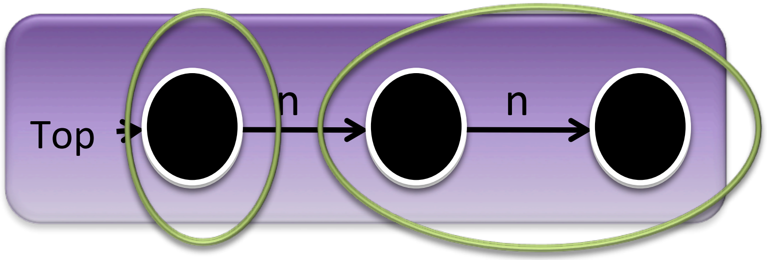
- One way to get tight embedding is canonical abstraction

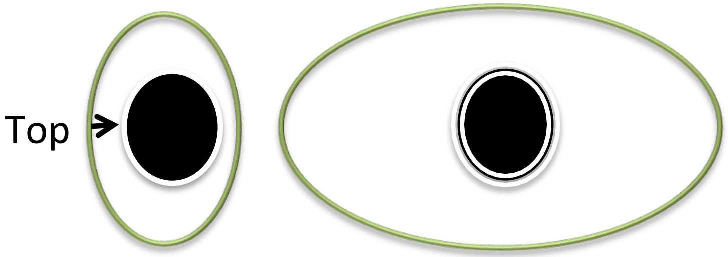
Canonical Abstraction



[Sagiv, Reps, Wilhelm, TOPLAS02]

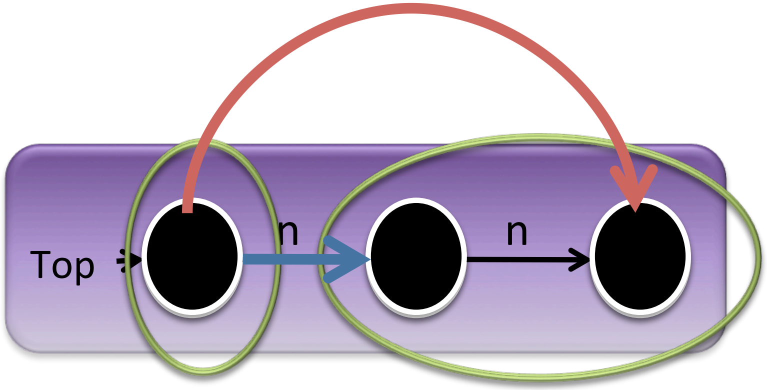
Canonical Abstraction

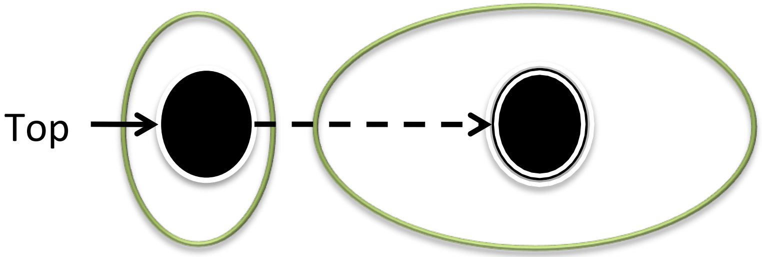




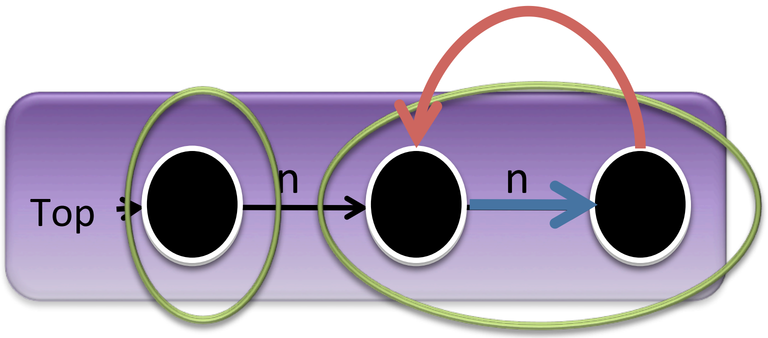
[Sagiv, Reps, Wilhelm, TOPLAS02]

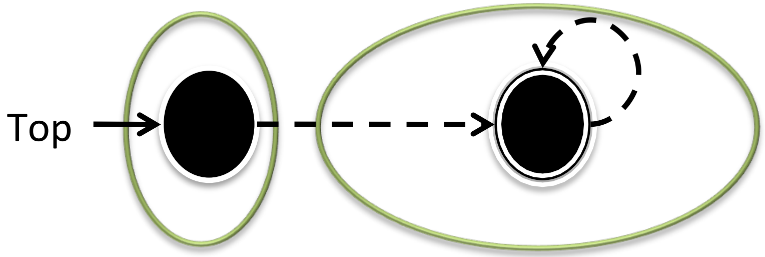
Canonical Abstraction



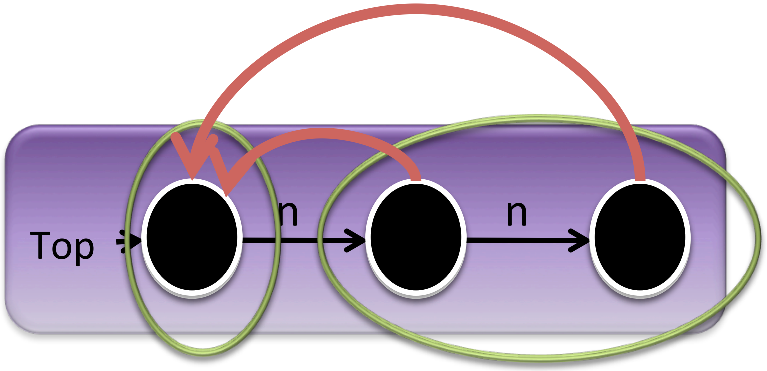


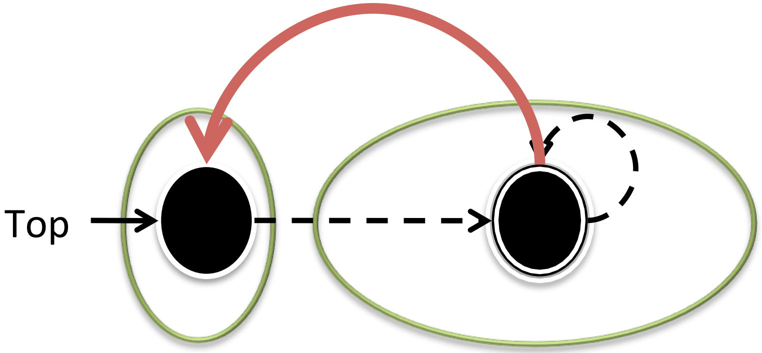
Canonical Abstraction



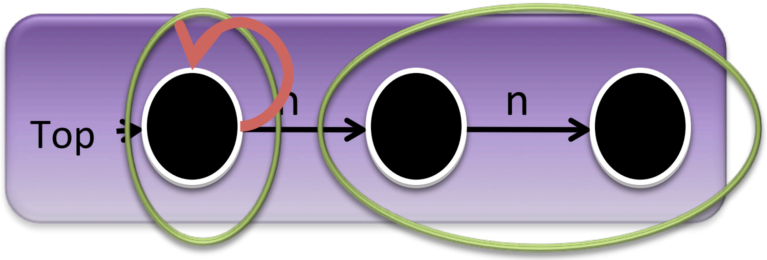


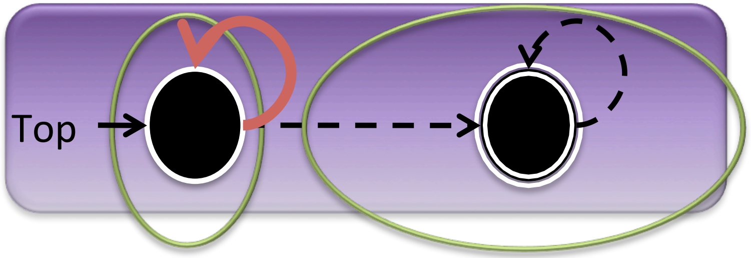
Canonical Abstraction





Canonical Abstraction





Canonical Abstraction (β)

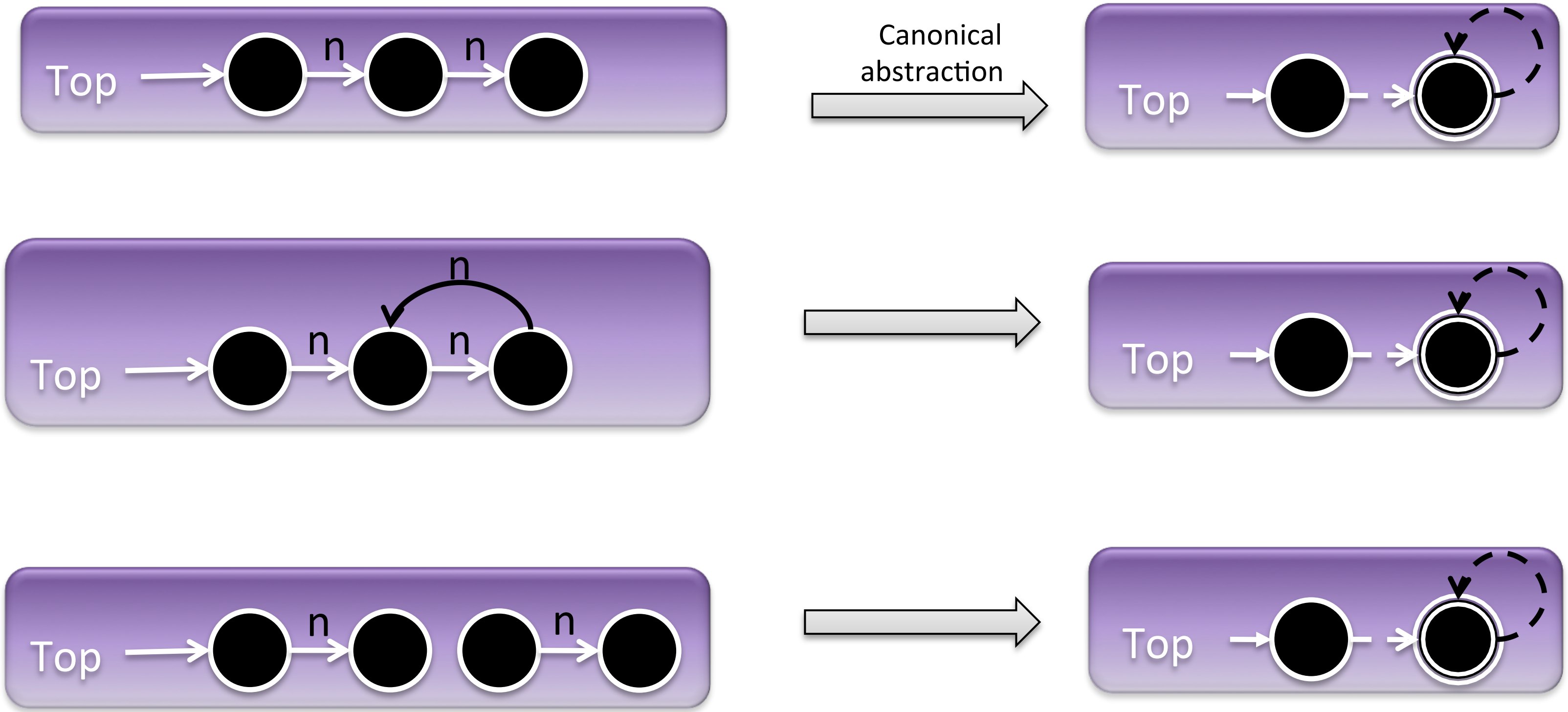
- Merge all nodes with the same unary predicate values into a single summary node
- Join predicate values

$$\iota'(u'_1,\ldots,u'_k) = \sqcup \{\iota(u_1,\ldots,u_k) \mid f(u_1)=u'_1,\ldots,f(u_k)=u'_k\}$$

- Converts a state of arbitrary size into a 3-valued abstract state of bounded size

- $\alpha(C) = \sqcup \{ \beta(c) \mid c \in C \}$

Information Loss



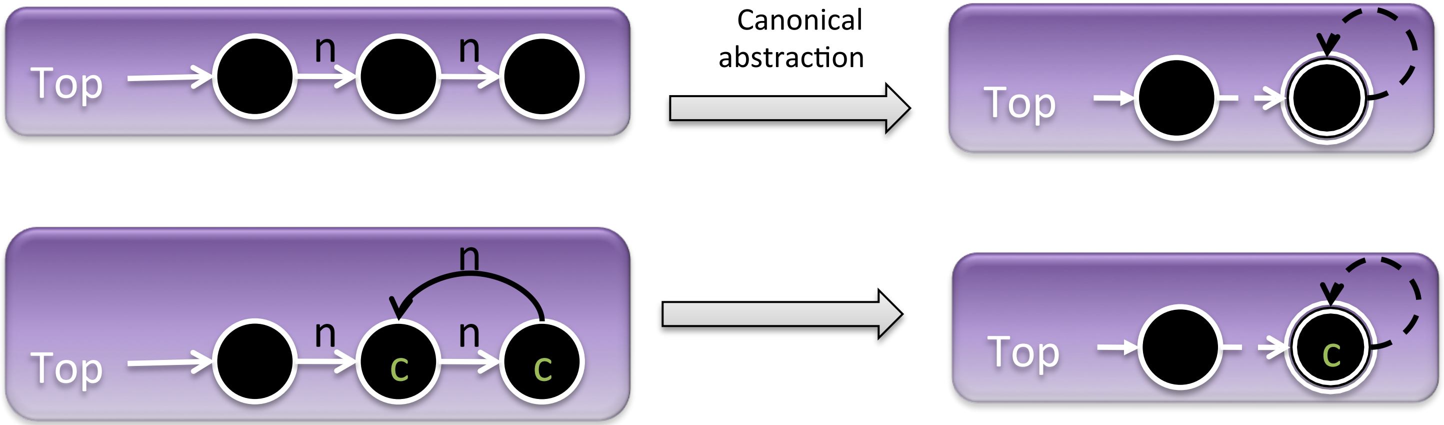
Canonical abstraction

Instrumentation Predicates

- Record additional derived information via predicates

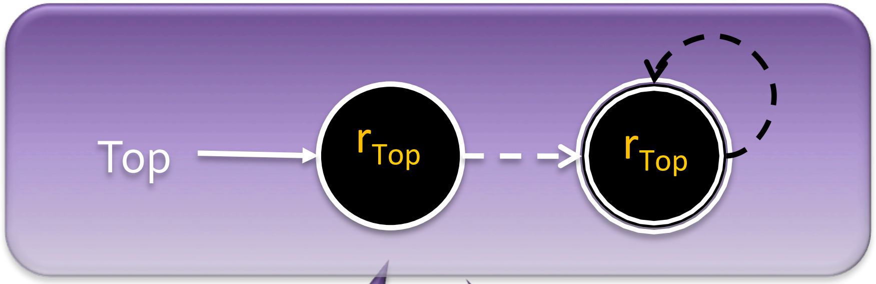
$$r_x(v) = \exists v1: x(v1) \wedge n^*(v1,v)$$

$$c(v) = \exists v1: n(v1, v) \wedge n^*(v, v1)$$



Canonical abstraction

Embedding Theorem: **Conservatively** Observing Properties



No Cycles

$\neg\exists v1,v2: n(v1, v2) \land n^*(v2, v1)$ 1/2

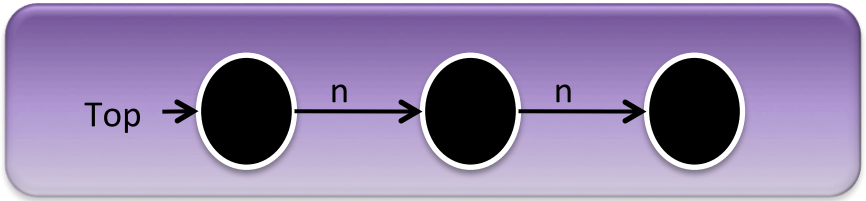
No cycles (derived)

$\forall v:\neg c(v)$ 1

Operational Semantics

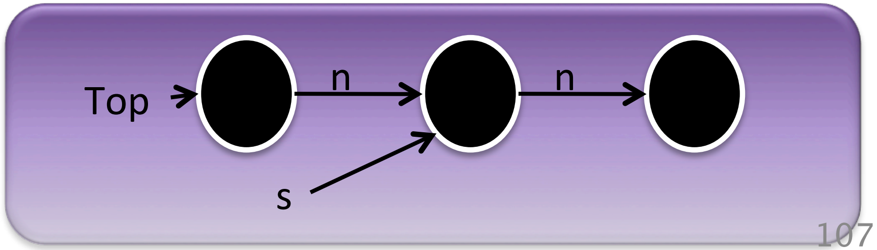
```
void push (int v) {
 Node *x = malloc(sizeof(Node));
 x->d = v;
 x->n = Top;
 Top = x;
}


 int pop() {
  if (Top == NULL) return EMPTY;
  Node *s = Top->n;
  int r = Top->d;
  Top = s;
  return r;
 }
```

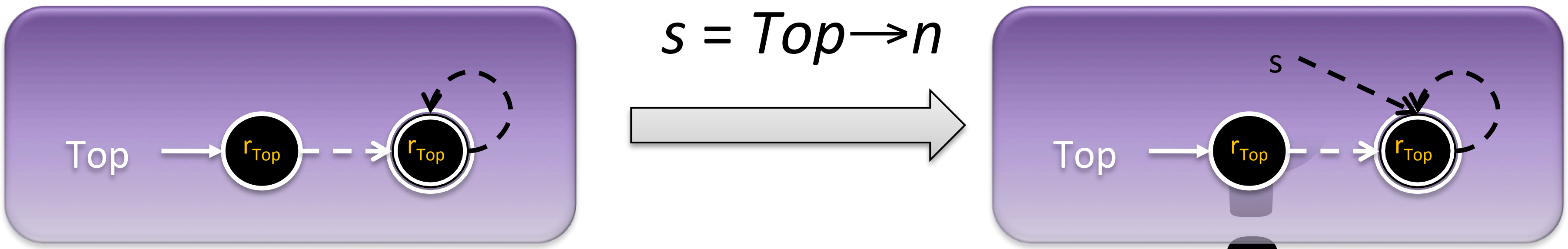


⟦s = Top->n⟧

$$s'(v) = \exists v1 \colon Top(v1) \wedge n(v1,v)$$



Abstract Semantics

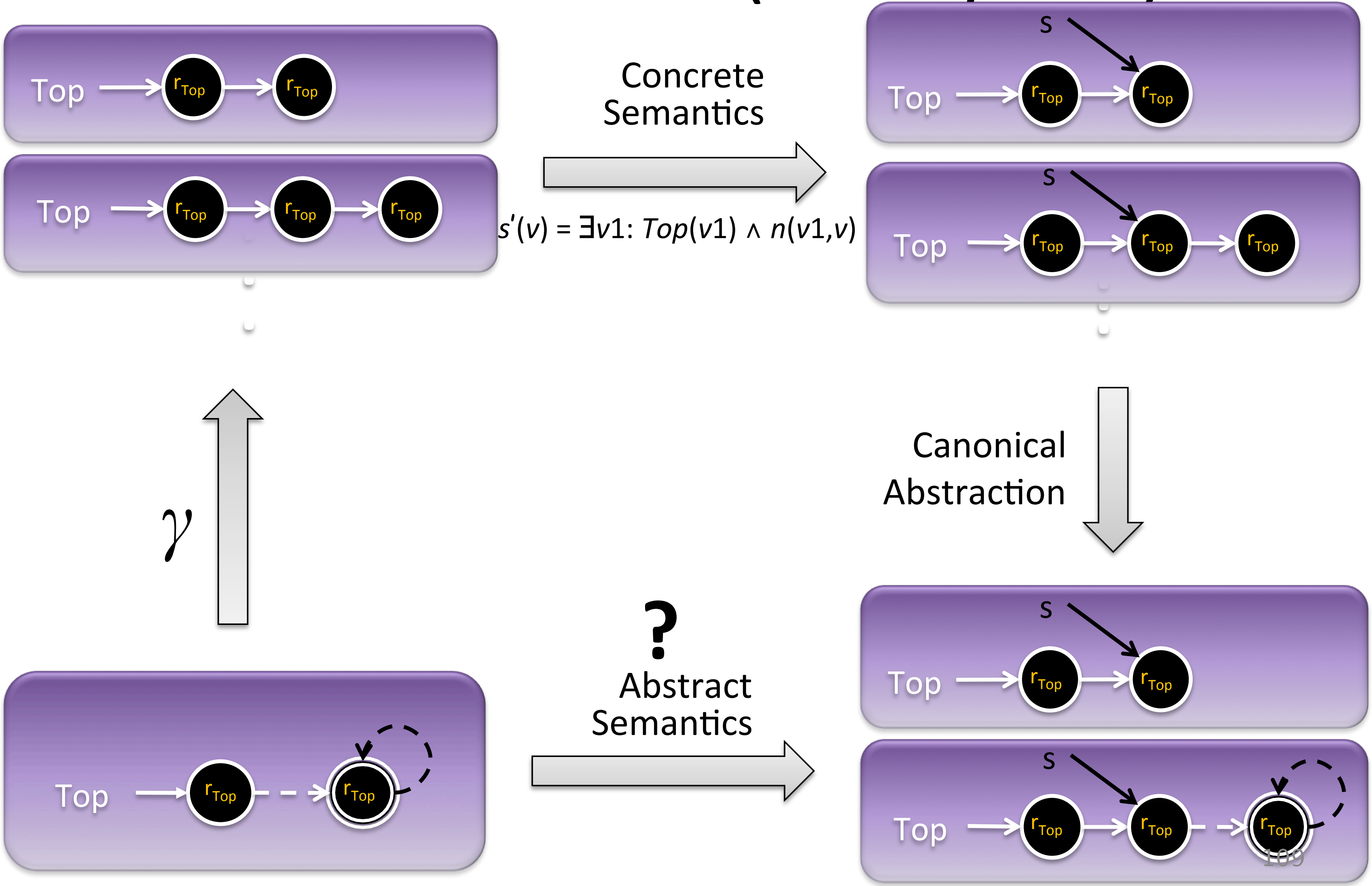


$s = Top \rightarrow n$

$\llbracket$`s = Top->n`$\rrbracket$

$s'(v) = \exists v1: Top(v1) \wedge n(v1,v)$

Best Transformer ($s = Top \rightarrow n$)



Concrete Semantics

$s'(v) = \exists v1{:}\ Top(v1) \wedge n(v1,v)$

Canonical Abstraction

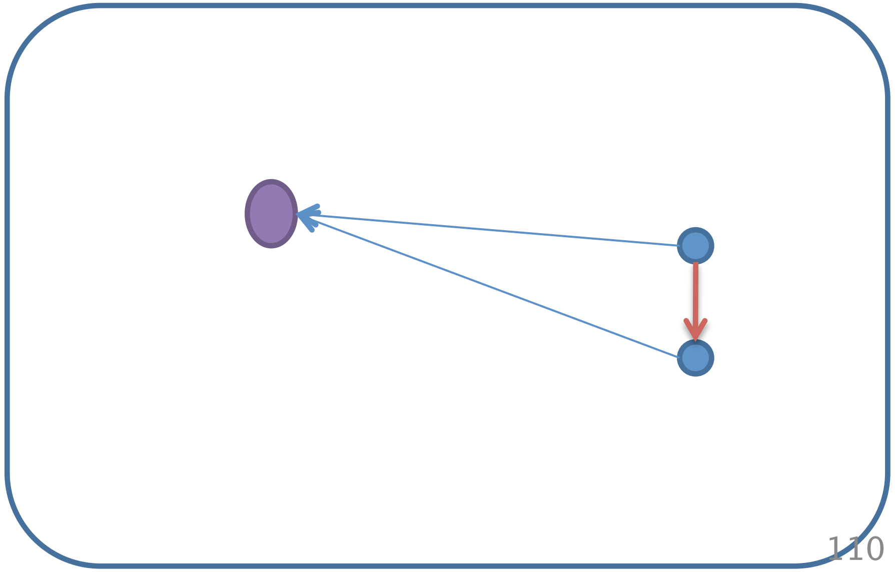
γ

?

Abstract Semantics

Semantic Reduction

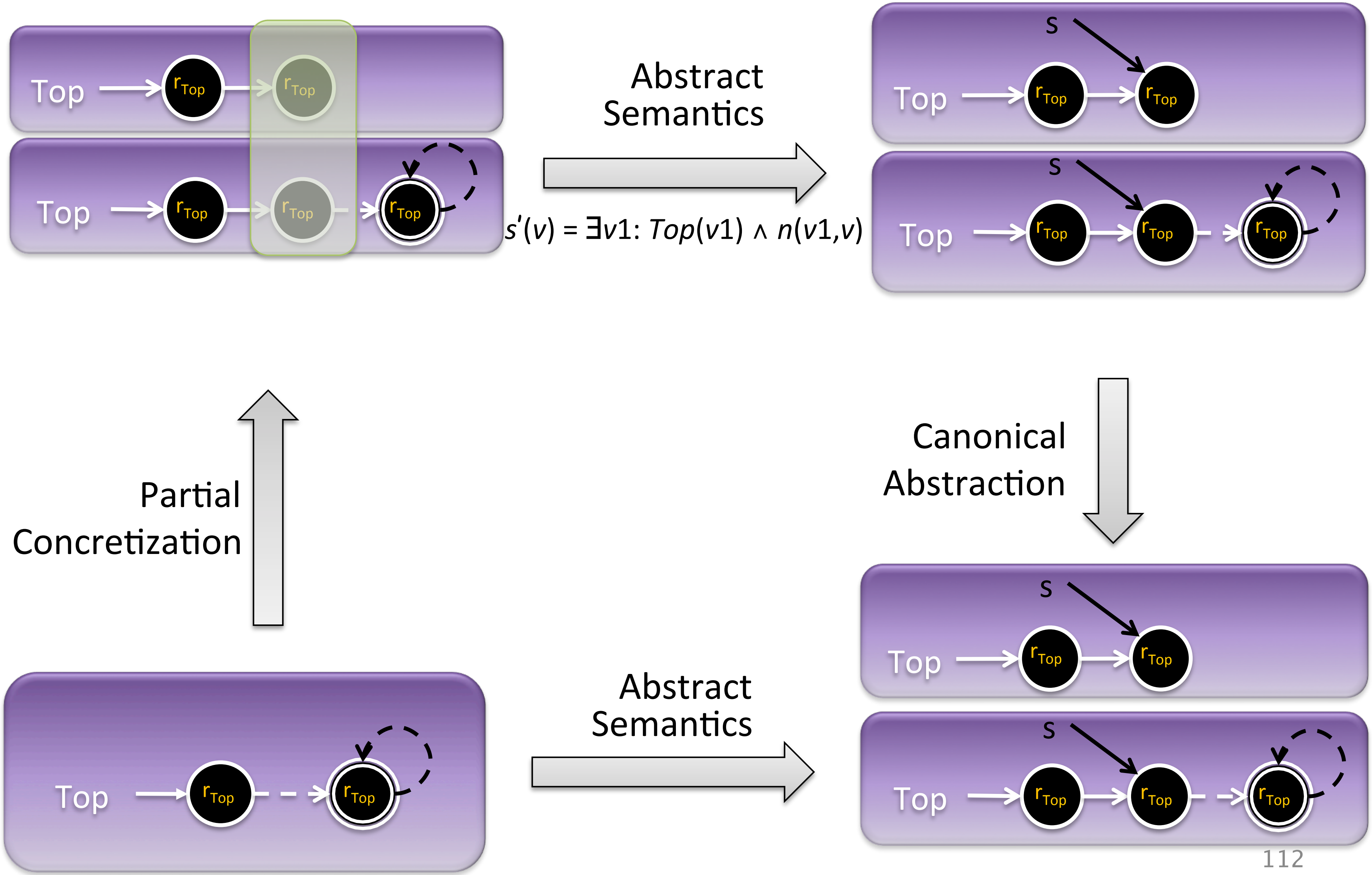
- Improve the precision of the analysis by recovering properties of the program semantics
- A Galois connection (C, α, γ, A)
- An operation op:A→A is a semantic reduction when
  - $\forall l \in L_2$ op(l)$\sqsubseteq$l and
  - γ(op(l)) = γ(l)



The Focus Operation

- Focus: Formula→($\wp$(3-Struct) $\hookrightarrow$ $\wp$(3-Struct))
- Generalizes materialization
- For every formula φ
  - Focus(φ)(X) yields structure in which φ evaluates to a definite values in all assignments
  - Only maximal in terms of embedding
  - Focus(φ) is a semantic reduction
  - But Focus(φ)(X) may be undefined for some X

Partial Concretization Based on Transformer (*s=Top→n*)



Abstract Semantics

$s'(v) = \exists v1\colon Top(v1) \wedge n(v1,v)$

Canonical Abstraction

Partial Concretization

Abstract Semantics

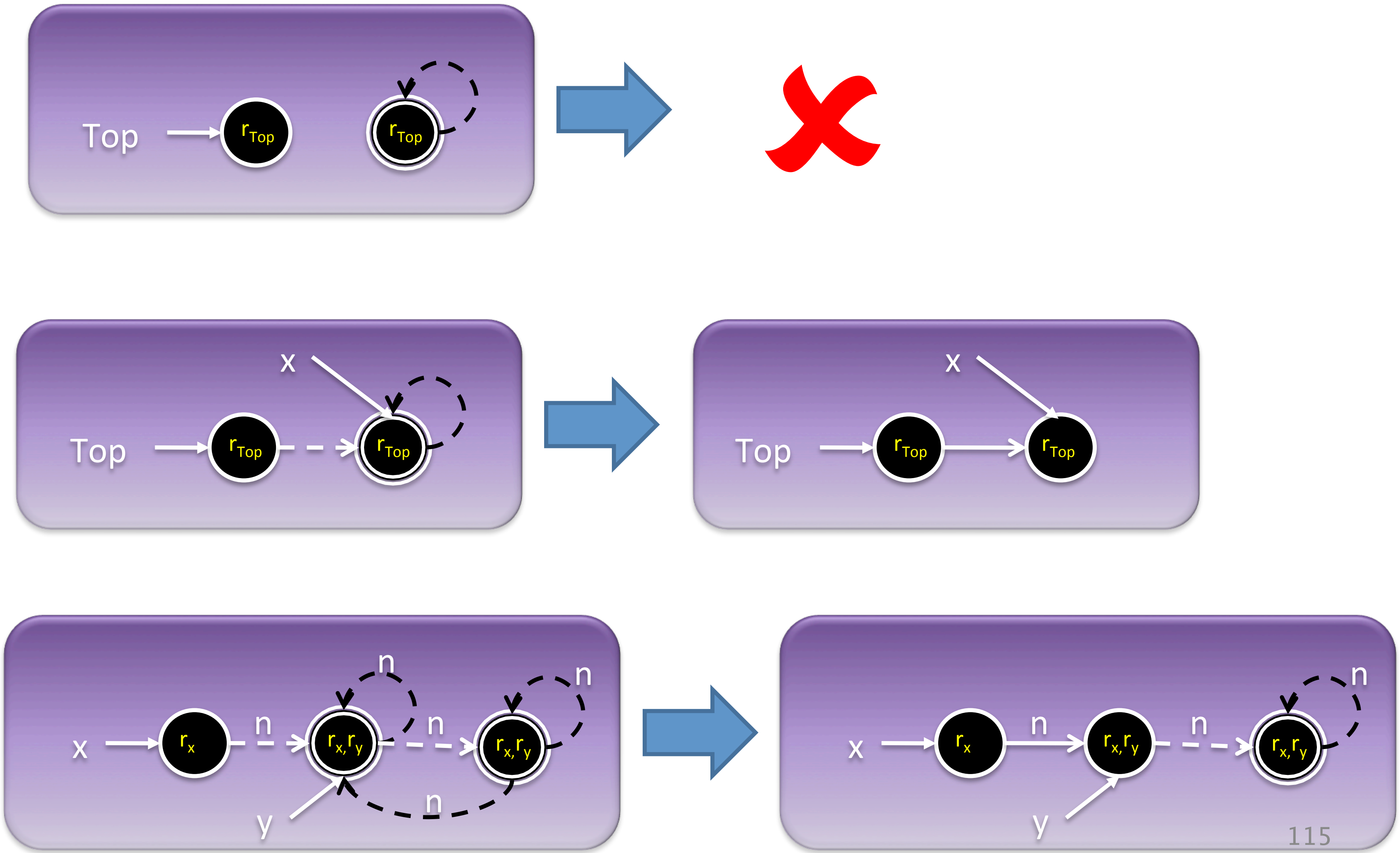
Partial Concretization

- Locally refine the abstract domain per statement
- Soundness is immediate
- Employed in other shape analysis algorithms [Distefano et.al., TACAS'06, Evan et.al., SAS'07, POPL'08]

The Coercion Principle

- Another Semantic Reduction
- Can be applied after Focus or after Update or both
- Increase precision by exploiting some structural properties possessed by all stores (Global invariants)
- Structural properties captured by constraints
- Apply a constraint solver

Apply Constraint Solver



Sources of Constraints

- Properties of the operational semantics
- Domain specific knowledge
  - Instrumentation predicates
- User supplied

Example Constraints

$x(v1) \wedge x(v2) \rightarrow eq(v1, v2)$

$n(v, v1) \wedge n(v,v2) \rightarrow eq(v1, v2)$

$n(v1, v) \wedge n(v2,v) \wedge \neg eq(v1, v2) \leftrightarrow is(v)$

$n^*(v3, v4) \leftrightarrow t[n](v1, v2)$

Abstract Transformers: Summary

- Kleene evaluation yields sound solution
- Focus is a statement-specific partial concretization
- Coerce applies global constraints

Abstract Semantics

$$
SS\ [v] = \begin{cases}
\{ <\emptyset,\emptyset> \} & \text{if } v = \text{entry} \\[2ex]
\displaystyle\bigcup_{\substack{(w,v)\, \in\, E(G), \\ w\, \in\, \mathrm{Assignments}(G)}} \{ \mathrm{t\_embed}(\mathrm{coerce}([\![\mathrm{st}(w)]\!]3(\mathrm{focus}_{F(w)}(SS[w]\ )))) \cup & \\[2ex]
\displaystyle\bigcup_{\substack{(w,v)\, \in\, E(G), \\ w\, \in\, \mathrm{Skip}(G)}} \{ S \mid S \in SS[w]\ \} \cup & \text{othrewise} \\[2ex]
\displaystyle\bigcup_{(w,v)\, \in\, \text{True-Branches}(G)} \{ \mathrm{t\_embed}(S) \mid S \in \mathrm{coerce}([\![\mathrm{st}(w)]\!]3(\mathrm{focus}_{F(w)}(SS[w]\ ))) & \\
\qquad\qquad\qquad \text{and } S \vDash 3\ \mathrm{cond}(w) \} \cup & \\[2ex]
\displaystyle\underset{(w,v)\, \in\, \text{False-Branches}(G)}{\cup} \{ \mathrm{t\_embed}(S) \mid S \in \mathrm{coerce}([\![\mathrm{st}(w)]\!]3(\mathrm{focus}_{F(w)}(SS[w]\ ))) & \\
\qquad\qquad\qquad \text{and } S \vDash 3\ \neg\mathrm{cond}(w) \} \cup &
\end{cases}
$$

Recap

- Abstraction
  - canonical abstraction
  - recording derived information

- Transformers
  - partial concretization (focus)
  - constraint solver (coerce)
  - sound information extraction

Stack Push

```
void push (int v) {
 Node *x =
   alloc(sizeof(Node));
```

∃v: x(v)

```
x→d    v;
```

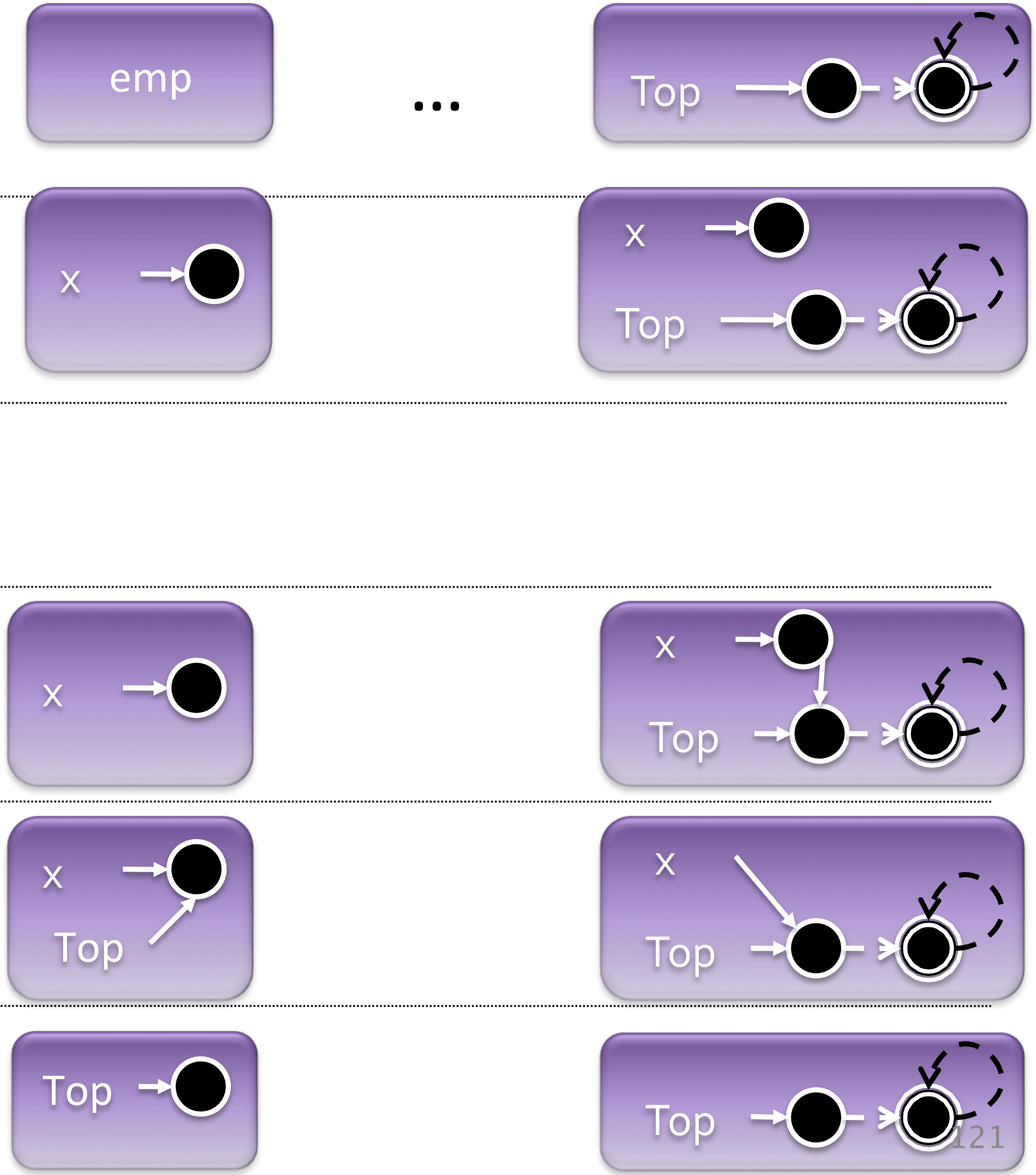
∃v: x(v)

```
x→
```

```
Top = x;
```

¬∃v1,v2: n(v1, v2) ∧Top(v2)

∀v:¬c(v)


emp
…
Top
x
x
Top
x
x
Top
x
Top
x
Top
x
Top
Top
Top