Program Analysis and Verification

0368-4479

Noam Rinetzky

Lecture 9: Abstract Interpretation

Slides credit: Roman Manevich, Mooly Sagiv, Eran Yahav

Abstract Interpretation [Cousot'77]

- Mathematical framework for approximating semantics (aka abstraction)
  - Allows designing sound static analysis algorithms
    - Usually compute by iterating to a fixed-point

  - Computes (loop) invariants
    - Can be interpreted as axiomatic verification assertions
    - Generalizes Hoare Logic & WP / SP calculus

Required knowledge

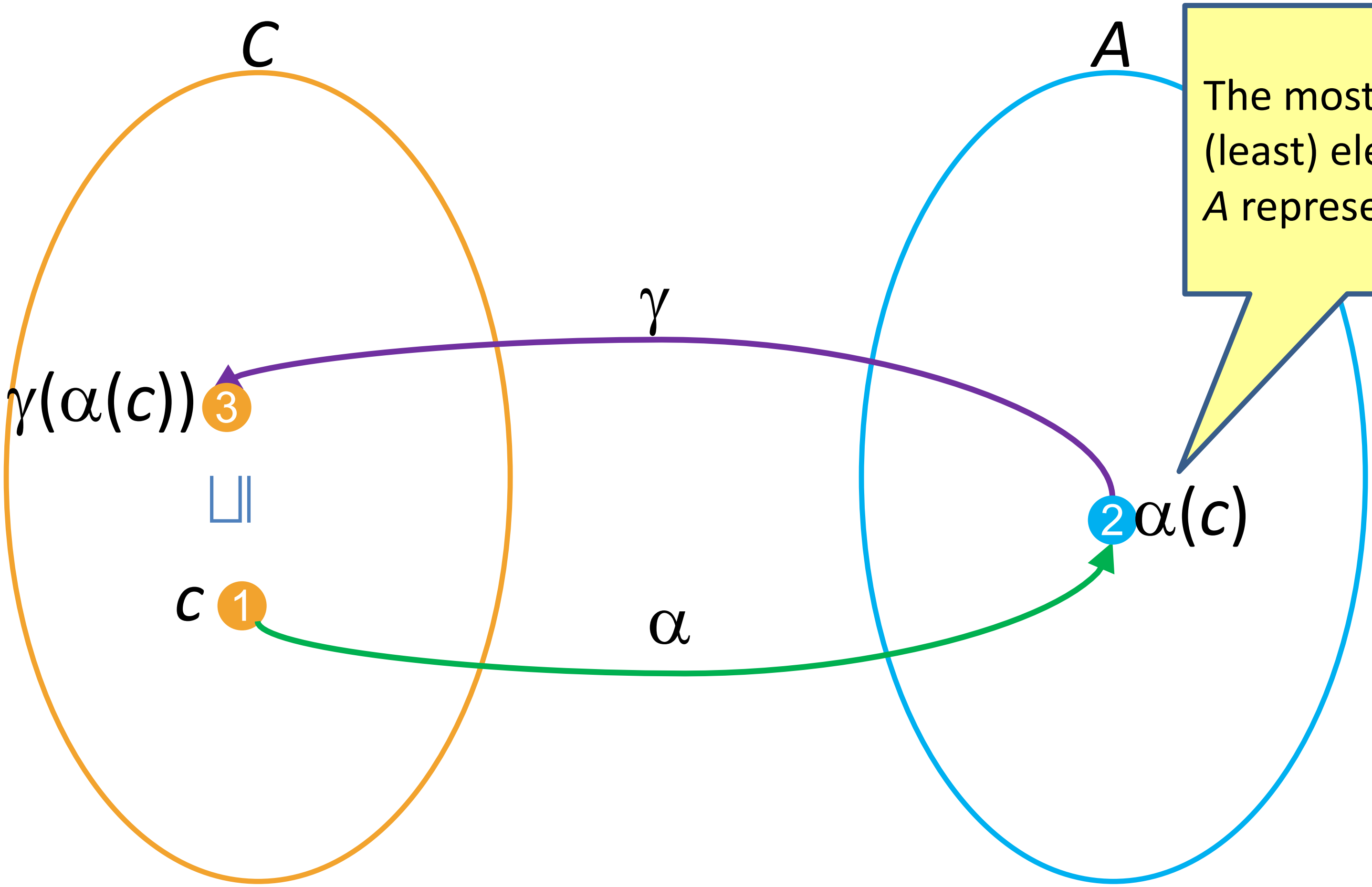
- ✓ Domain theory
- ✓ Collecting semantics
- ✓ Abstract semantics (over lattices)
- ✓ Algorithm to compute abstract semantics (chaotic iteration)
- ✓ Connection between collecting semantics and abstract semantics
- ✓ Abstract transformers

Galois Connection

- Given two complete lattices
  $C = (D^C, \sqsubseteq^C, \sqcup^C, \sqcap^C, \bot^C, \top^C)$ – concrete domain
  $A = (D^A, \sqsubseteq^A, \sqcup^A, \sqcap^A, \bot^A, \top^A)$ – abstract domain
- A Galois Connection (GC) is quadruple (C, α, γ, A) that relates C and A via the monotone functions
  - The abstraction function $\alpha : D^C \to D^A$
  - The concretization function $\gamma : D^A \to D^C$
- for every concrete element $c \in D^C$ and abstract element $a \in D^A$

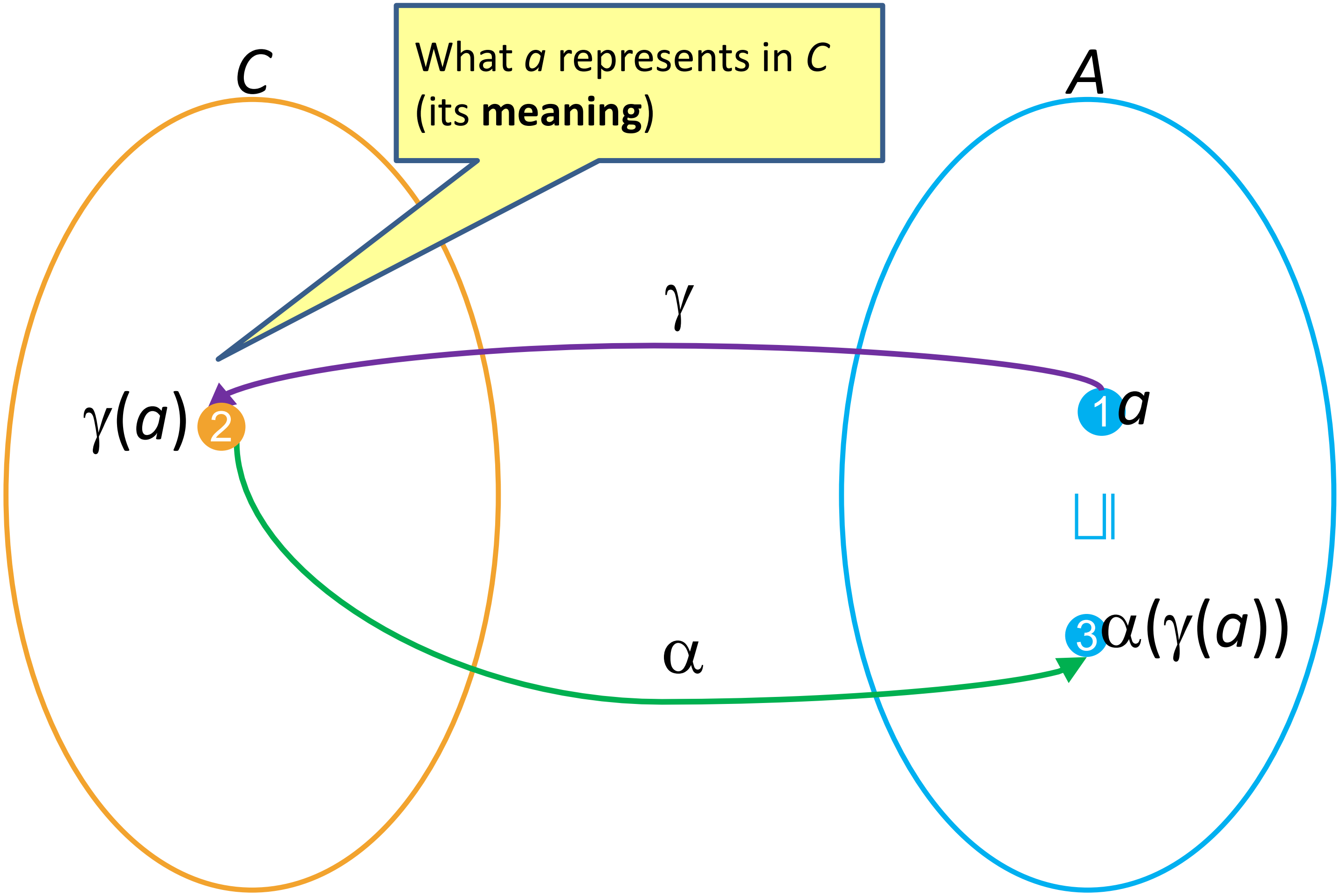
  $$\alpha(\gamma(a)) \sqsubseteq a \text{ and } c \sqsubseteq \gamma(\alpha(c))$$
- Alternatively $\alpha(c) \sqsubseteq a$ iff $c \sqsubseteq \gamma(a)$

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



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

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



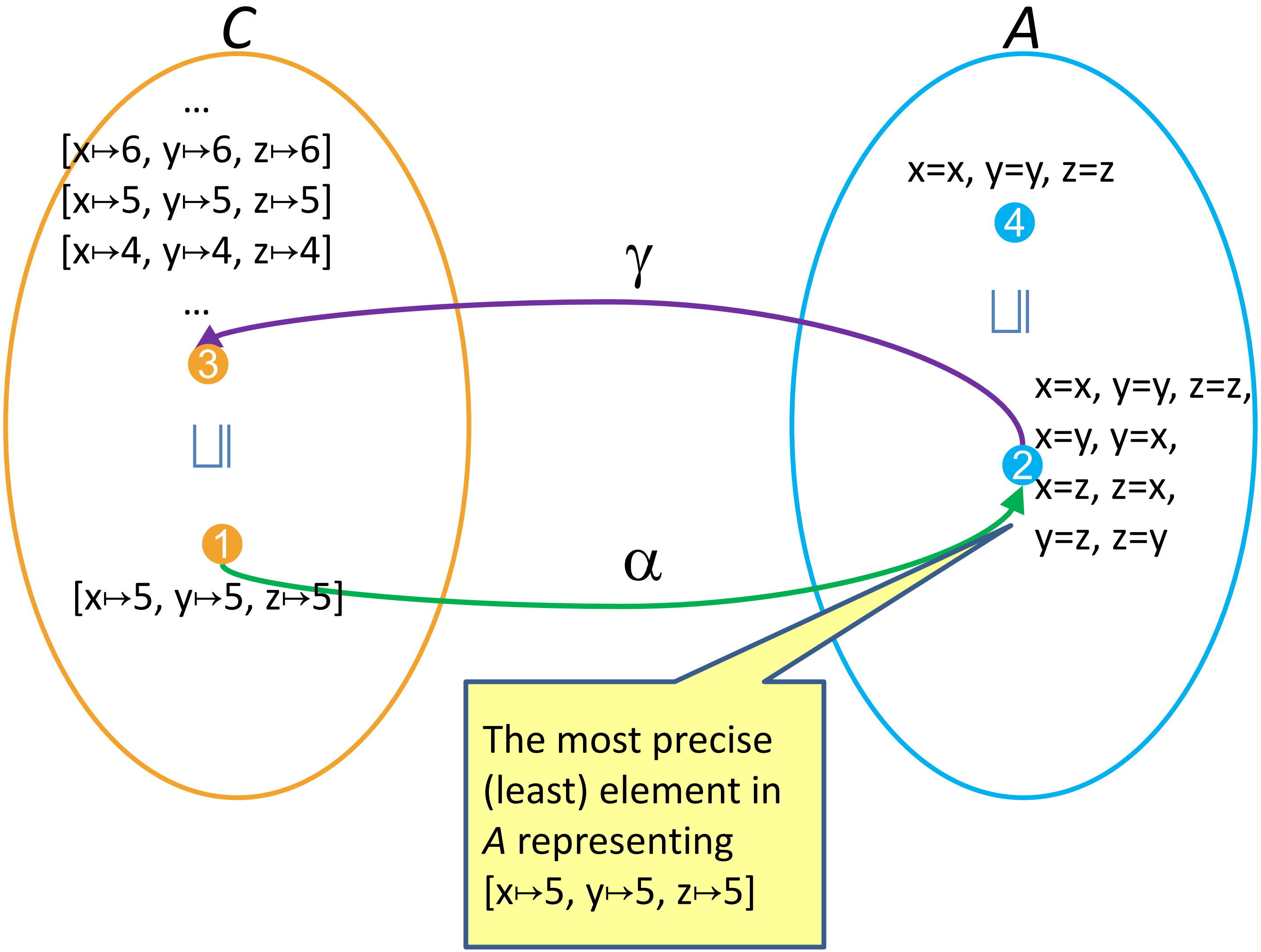
Example: lattice of equalities

- Concrete lattice:
  $C = (2^{\textbf{State}}, \subseteq, \cup, \cap, \emptyset, \textbf{State})$
- Abstract lattice:
  $EQ = \{ x{=}y \mid x, y \in \mathrm{Var}\}$
  $A = (2^{EQ}, \supseteq, \cap, \cup, EQ\ , \emptyset)$
  - Treat elements of *A* as both formulas and sets of constraints
- Useful for copy propagation – a compiler optimization
  - $\alpha(X)$ = ?
  - $\gamma(\mathrm{Y})$ = ?

Example: lattice of equalities

- Concrete lattice:
  $C = (2^{\mathbf{State}}, \subseteq, \cup, \cap, \emptyset, \mathbf{State})$
- Abstract lattice:
  $EQ = \{ x{=}y \mid x, y \in \mathrm{Var}\}$
  $A = (2^{EQ}, \supseteq, \cap, \cup, EQ , \emptyset)$
  - Treat elements of *A* as both formulas and sets of constraints
- Useful for copy propagation – a compiler optimization
- $\beta(s) = \alpha(\{s\}) = \{ x{=}y \mid s\ x = s\ y\}$ that is $s \vDash x{=}y$
  $\alpha(X) = \cap\{\beta(s) \mid s{\in}X\} = \sqcup^{A} \{\beta(s) \mid s{\in}X\}$
  $\gamma(\mathrm{Y}) = \{ s \mid s \vDash \wedge \mathrm{Y} \} = \mathrm{models}(\wedge \mathrm{Y})$

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



C

…

[x↦6, y↦6, z↦6]

[x↦5, y↦5, z↦5]

[x↦4, y↦4, z↦4]

…

③

⊑

①

[x↦5, y↦5, z↦5]

A

x=x, y=y, z=z

④

⊑

②

x=x, y=y, z=z,
x=y, y=x,
x=z, z=x,
y=z, z=y

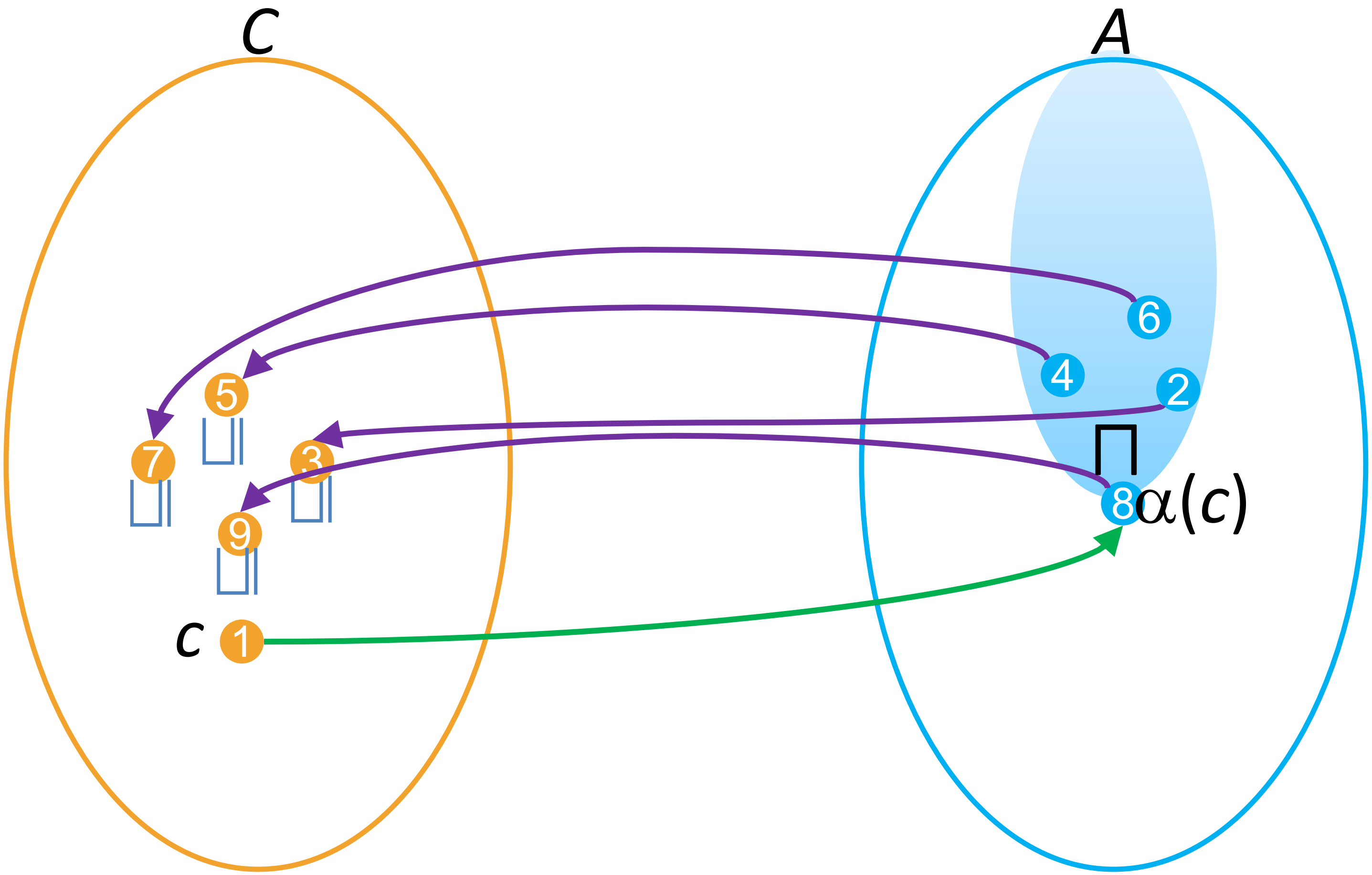
γ

α

The most precise (least) element in *A* representing [x↦5, y↦5, z↦5]

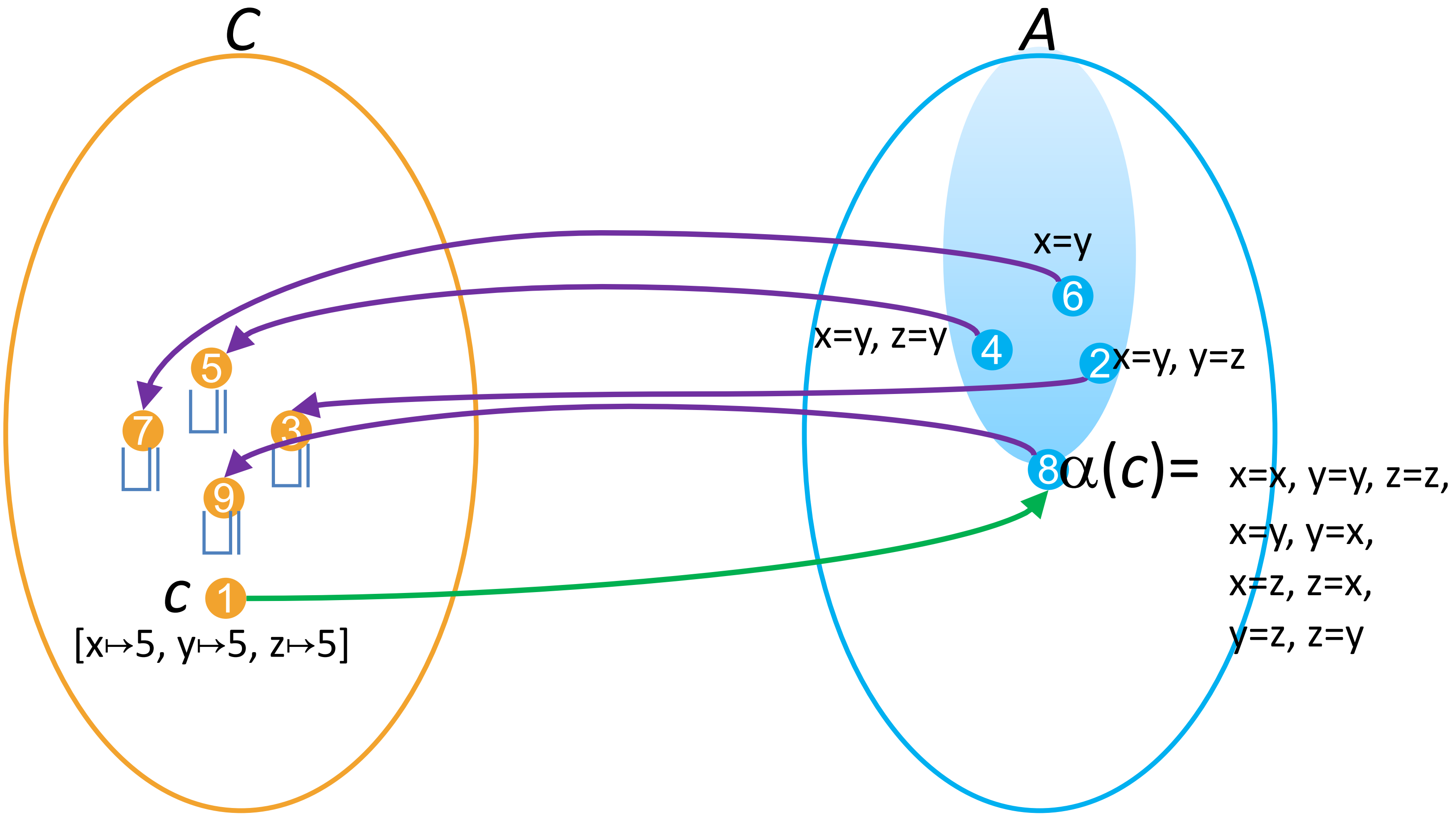
Most precise abstract representation

$$\boxed{\alpha(c) = \sqcap\{a \mid c \sqsubseteq \gamma(a)\}}$$

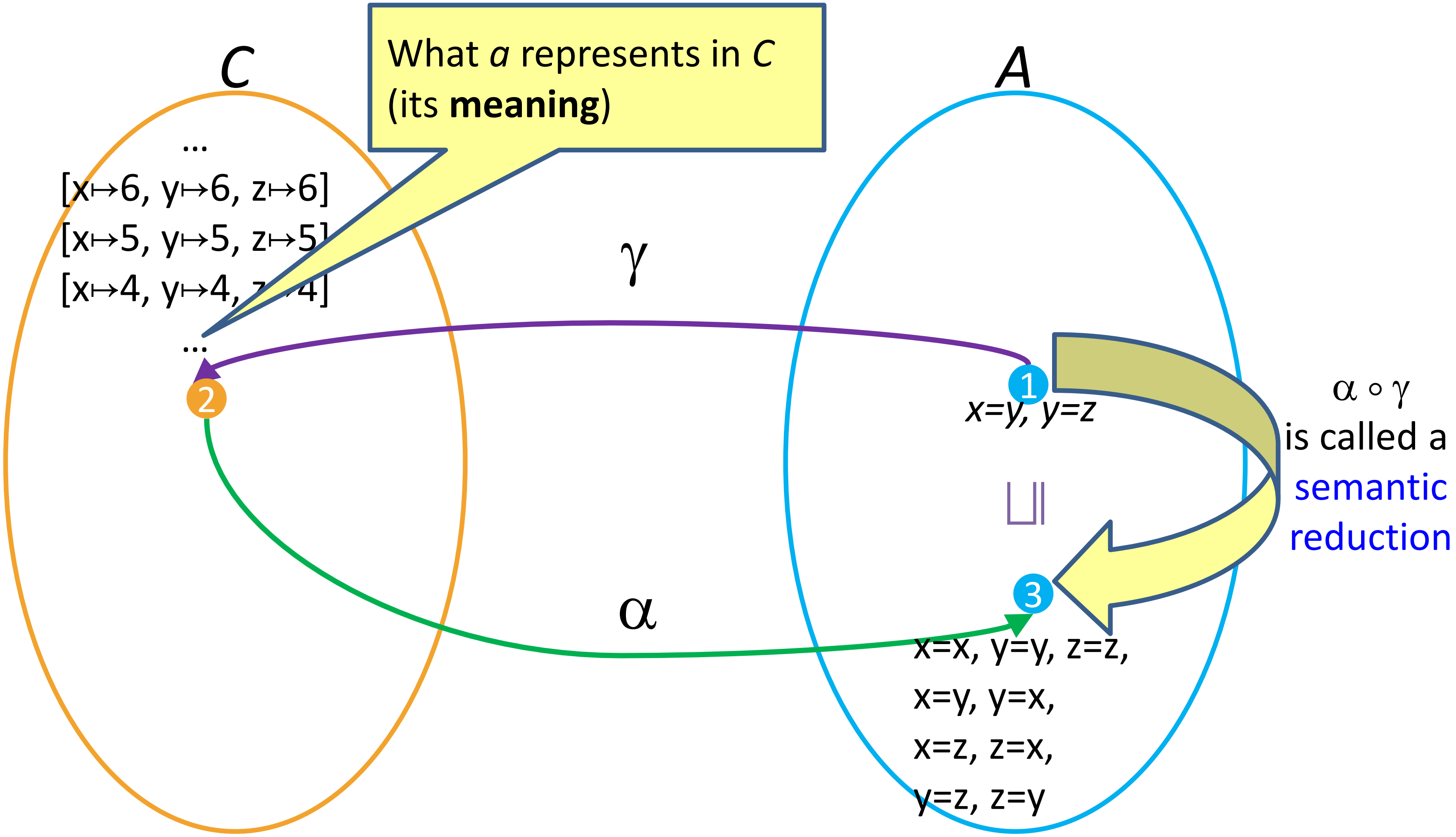


Most precise abstract representation

$$\boxed{\alpha(c) = \sqcap\{a \mid c \sqsubseteq \gamma(a)\}}$$



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



What *a* represents in *C* (its **meaning**)

C

…
[x↦6, y↦6, z↦6]
[x↦5, y↦5, z↦5]
[x↦4, y↦4, z↦4]
…

②

A

① *x=y, y=z*

$\sqsupseteq$

③ x=x, y=y, z=z,
x=y, y=x,
x=z, z=x,
y=z, z=y

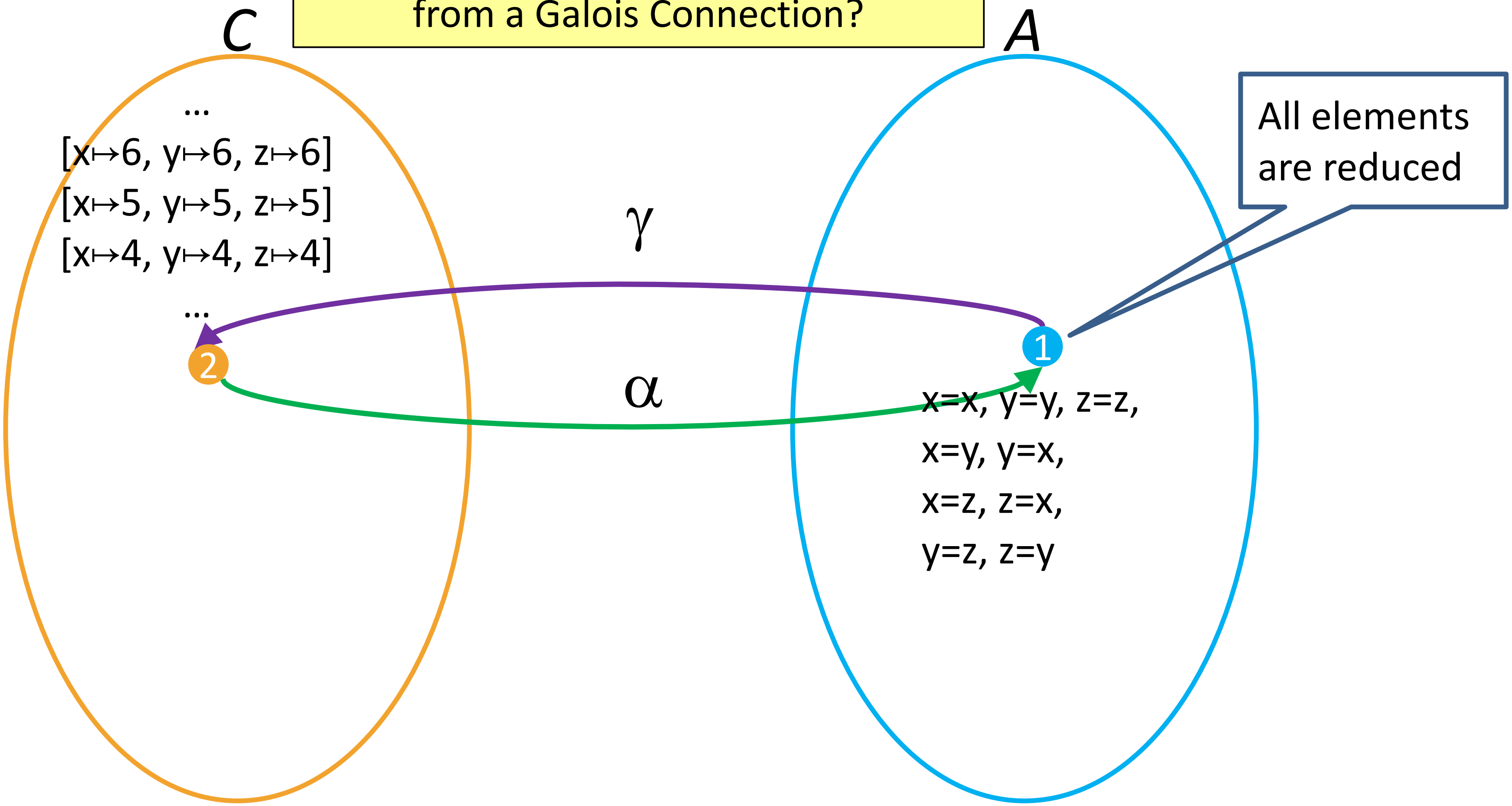
γ

α

$\alpha \circ \gamma$
is called a
semantic
reduction

Galois Insertion $\forall a: \alpha(\gamma(a))=a$

How can we obtain a Galois Insertion from a Galois Connection?



…
[x↦6, y↦6, z↦6]
[x↦5, y↦5, z↦5]
[x↦4, y↦4, z↦4]
…

x=x, y=y, z=z,
x=y, y=x,
x=z, z=x,
y=z, z=y

All elements are reduced

Properties of a Galois Connection

- The abstraction and concretization functions uniquely determine each other:

$$\gamma(a) = \bigsqcup\{c \mid \alpha(c) \sqsubseteq a\}$$

$$\alpha(c) = \bigsqcap\{a \mid c \sqsubseteq \gamma(a)\}$$

Abstracting (disjunctive) sets

- It is usually convenient to first define the abstraction of single elements

  $$\beta(s) = \alpha(\{s\})$$

- Then lift the abstraction to sets of elements

  $$\alpha(X) = \sqcup^{A} \{\beta(s) \mid s \in X\}$$

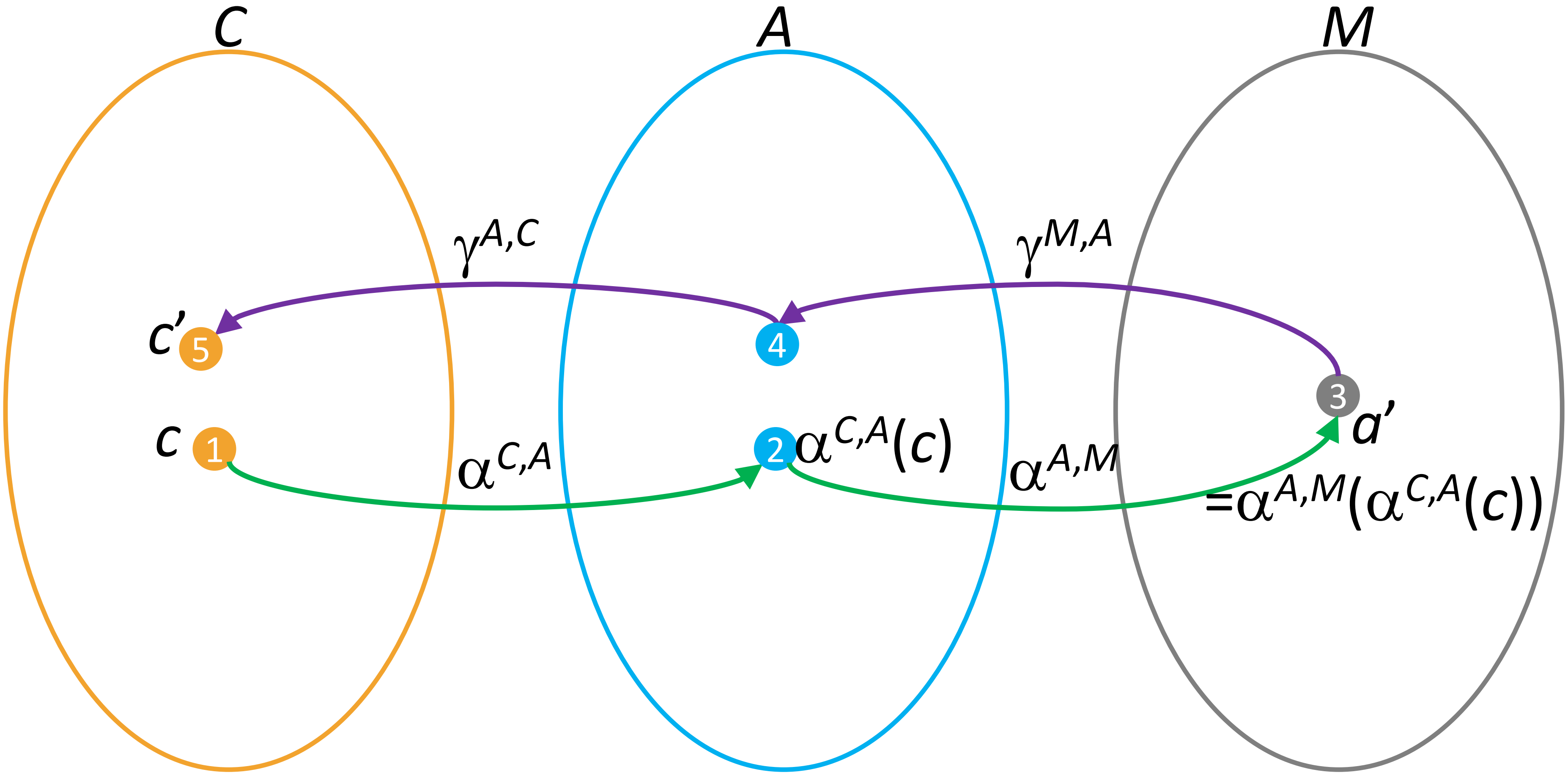
The case of symbolic domains

- An important class of abstract domains are symbolic domains – domains of formulas
- $C = (2^{\textbf{State}}, \subseteq, \cup, \cap, \varnothing, \textbf{State})$
  $A = (D^A, \sqsubseteq^A, \sqcup^A, \sqcap^A, \bot^A, \top^A)$
- If D^A is a set of formulas then the abstraction of a state is defined as
  $$\beta(s) = \alpha(\{s\}) = \sqcap^A\{\varphi \mid s \vDash \varphi\}$$
  the least formula from D^A that s satisfies
- The abstraction of a set of states is
  $$\alpha(X) = \sqcup^A \{\beta(s) \mid s \in X\}$$
- The concretization is
  $$\gamma(\varphi) = \{ s \mid s \vDash \varphi \} = \text{models}(\varphi)$$

Inducing along the connections

- Assume the complete lattices
  $C = (D^C, \sqsubseteq^C, \sqcup^C, \sqcap^C, \bot^C, \top^C)$
  $A = (D^A, \sqsubseteq^A, \sqcup^A, \sqcap^A, \bot^A, \top^A)$
  $M = (D^M, \sqsubseteq^M, \sqcup^M, \sqcap^M, \bot^M, \top^M)$
  and
  Galois connections
  $GC^{C,A}=(C, \alpha^{C,A}, \gamma^{A,C}, A)$ and $GC^{A,M}=(A, \alpha^{A,M}, \gamma^{M,A}, M)$
- **Lemma:** both connections induce the
  $GC^{C,M}= (C, \alpha^{C,M}, \gamma^{M,C}, M)$
  defined by $\alpha^{C,M} = \alpha^{C,A} \circ \alpha^{A,M}$ and $\gamma^{M,C} = \gamma^{M,A} \circ \gamma^{A,C}$

Inducing along the connections

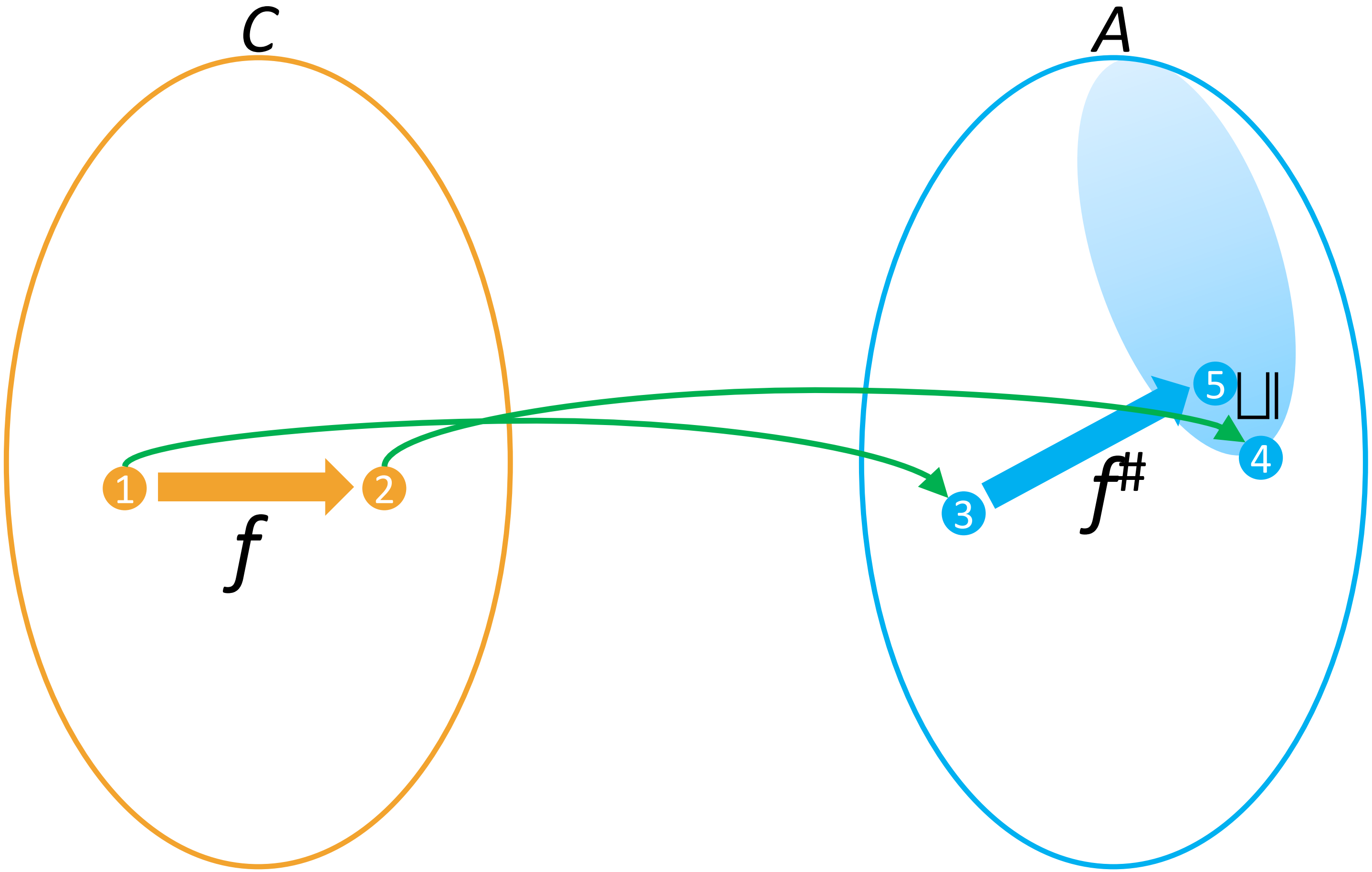


Sound abstract transformer

- Given two 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
- A concrete transformer $f : D^C \rightarrow D^C$
  an abstract transformer $f^\# : D^A \rightarrow D^A$
- We say that $f^\#$ is a sound transformer (w.r.t. f) if
  - $\forall c: f(c)=c' \Rightarrow \alpha(f^\#(c)) \sqsupseteq \alpha(c')$
  - For every a and a' such that
    $$\alpha(f(\gamma(a))) \sqsubseteq^A f^\#(a)$$

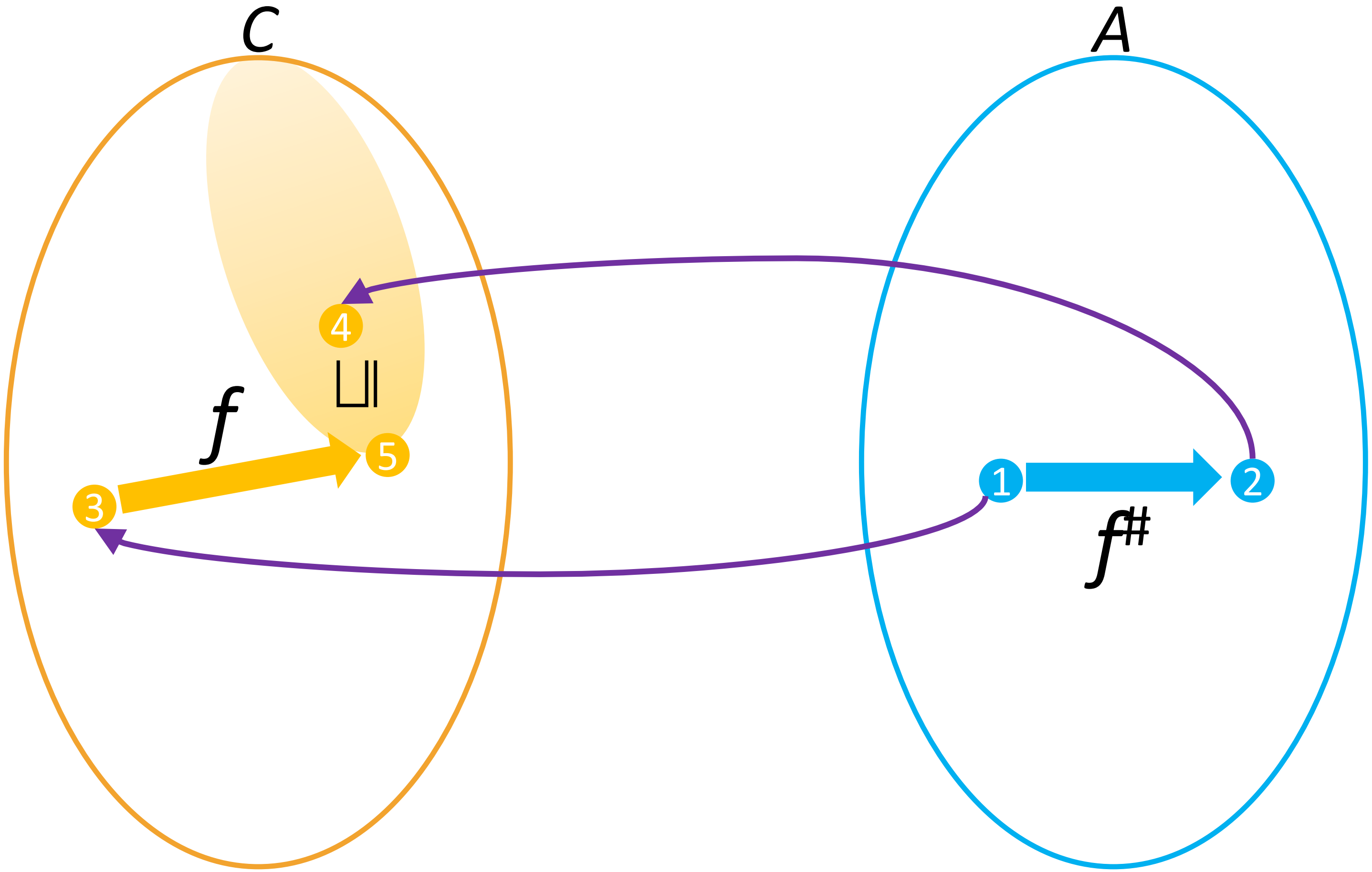
Transformer soundness condition 1

$$\forall c: f(c)=c' \Rightarrow \alpha(f^{\#}(c)) \sqsupseteq \alpha(c')$$



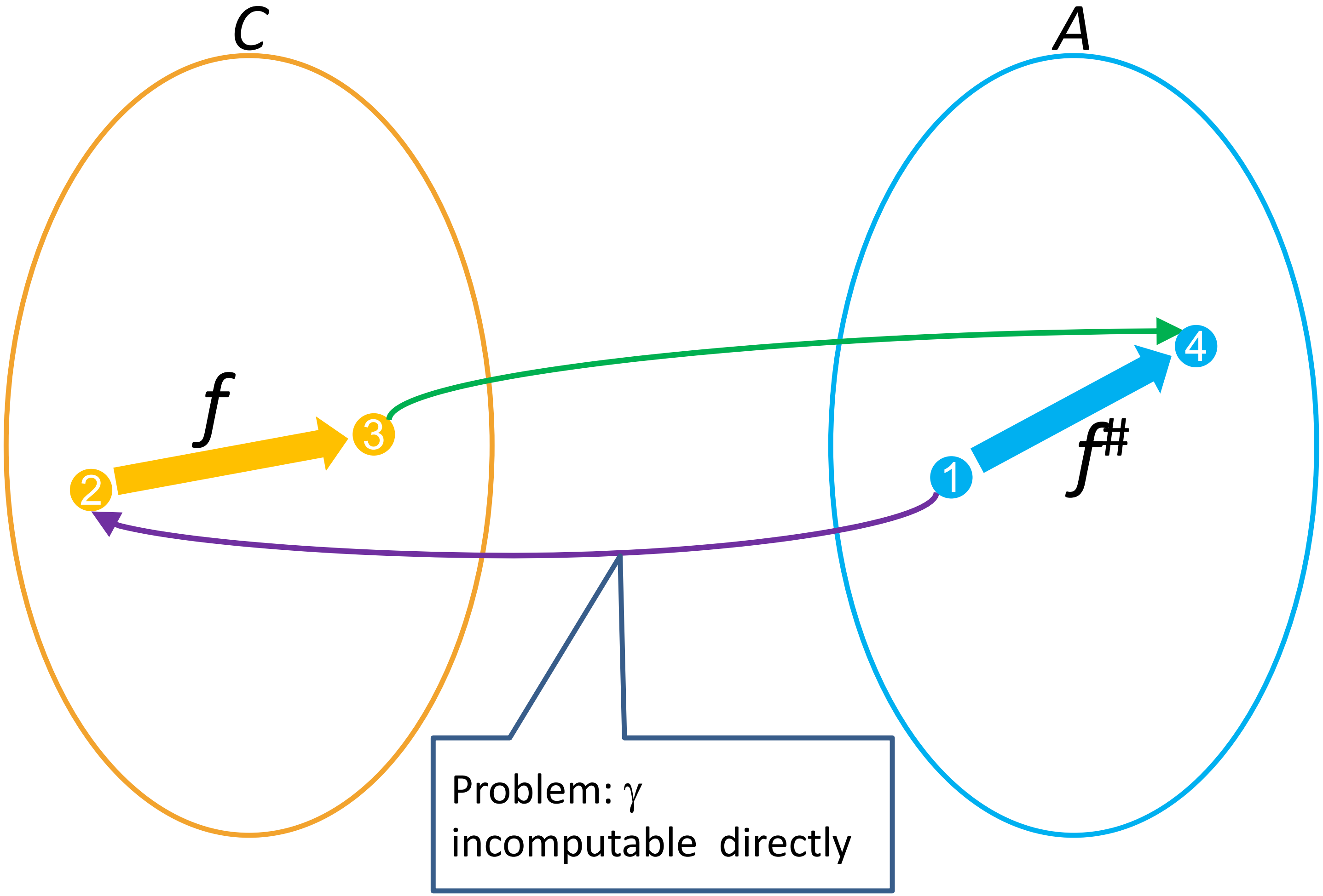
Transformer soundness condition 2

$\forall a: f^{\#}(a)=a' \Rightarrow f(\gamma(a)) \sqsubseteq \gamma(a')$



Best (induced) transformer

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



Problem: γ incomputable directly

Best abstract transformer [CC'77]

- Best in terms of **precision**
  - Most precise abstract transformer
  - May be too expensive to compute
- Constructively defined as

$$f^{\#} = \alpha \circ f \circ \gamma$$

  - Induced by the GC
- Not directly computable because first step is concretization
- We often compromise for a "good enough" transformer
  - Useful tool: partial concretization

Transformer example

- $C = (2^{\mathbf{State}}, \subseteq, \cup, \cap, \emptyset, \mathbf{State})$
- $EQ = \{\, x{=}y \mid x, y \in \mathrm{Var}\}$
  $A = (2^{EQ}, \supseteq, \cap, \cup, EQ\ , \emptyset)$
- $\beta(s) = \alpha(\{s\}) = \{\, x{=}y \mid s\ x = s\ y\}$ that is $s \vDash x{=}y$
  $\alpha(X) = \cap\{\beta(s) \mid s{\in}X\} = \sqcup^{A}\ \{\beta(s) \mid s{\in}X\}$
  $\gamma(\varphi) = \{\ s \mid s \vDash \varphi\ \} = \mathrm{models}(\varphi)$
- Concrete: $[\![x{:=}y]\!]\ X = \{\ s[\mathrm{x}{\mapsto}s\ \mathrm{y}] \mid s{\in}X\}$
- Abstract: $[\![x{:=}y]\!]^{\#}\ X = ?$

Developing a transformer for *EQ* - 1

- Input has the form $X = \bigwedge\{a=b\}$
- $\text{sp}(x{:=}expr,\ \varphi) = \exists v.\ x{=}\text{expr}[v/\text{x}] \wedge \varphi[v/\text{x}]$
- $\text{sp}(x{:=}y,\ X) = \exists v.\ x{=}y[v/\text{x}] \wedge \bigwedge\{a{=}b\}[v/\text{x}] = \ldots$
- Let's define helper notations:
  - $\text{EQ}(X, y) = \{y{=}a,\ b{=}y \in X\}$
    - Subset of equalities containing *y*
  - $\text{EQc}(X, y) = X \setminus \text{EQ}(X, y)$
    - Subset of equalities **not** containing *y*

Developing a transformer for *EQ* - 2

- $\mathrm{sp}(x{:=}y,\ X) = \exists v.\ x=y[v/x] \wedge \bigwedge\{a=b\}[v/x] = \ldots$
- Two cases
  - x is y: $\mathrm{sp}(x{:=}y,\ X) = X$
  - x is different from y:
    $\mathrm{sp}(x{:=}y,\ X) = \exists v.\ x=y \wedge \mathrm{EQ}(X, x)[v/x] \wedge \mathrm{EQc}(X, x)[v/x]$
    $= x=y \wedge\ \mathrm{EQc}(X, x)\ \ \wedge \exists v.\ \mathrm{EQ}(X, x)[v/x]$
    $\Rightarrow x=y \wedge\ \mathrm{EQc}(X, x)$
- Vanilla transformer: $[\![x{:=}y]\!]^{\#1}\ X = x=y \wedge\ \mathrm{EQc}(X, x)$
- Example: $[\![x{:=}y]\!]^{\#1} \bigwedge\{x=p,\ q=x,\ m=n\} = \bigwedge\{x=y,\ m=n\}$
  Is this the most precise result?

Developing a transformer for *EQ* - 3

- $[\![x:=y]\!]^{\#1} \bigwedge\{x=p, x=q, m=n\} = \bigwedge\{x=y, m=n\} \sqsupseteq \bigwedge\{x=y, m=n, p=q\}$
  - Where does the information p=q come from?
- $\mathrm{sp}(x:=y,\ X) =$

  $$x=y \wedge \mathrm{EQc}(X, x) \wedge \exists v.\ \mathrm{EQ}(X, x)[v/x]$$

- $\exists v.\ \mathrm{EQ}(X, x)[v/x]$ holds possible equalities between different *a*'s and *b*'s – how can we account for that?

Developing a transformer for *EQ* - 4

- Define a reduction operator:

  ```
  Explicate(X) = if exist $\{a=b, b=c\} \subseteq X$
          but not $\{a=c\} \subseteq X$ then
    Explicate($X \cup \{a=c\}$)
          else
              X
  ```

- Define $[\![x{:=}y]\!]^{\#2} = [\![x{:=}y]\!]^{\#1} \circ$ Explicate
- $[\![x{:=}y]\!]^{\#2} \wedge(\{x=p, x=q, m=n\}) = \bigwedge\{x=y, m=n, p=q\}$ is this the best transformer?

Developing a transformer for *EQ* - 5

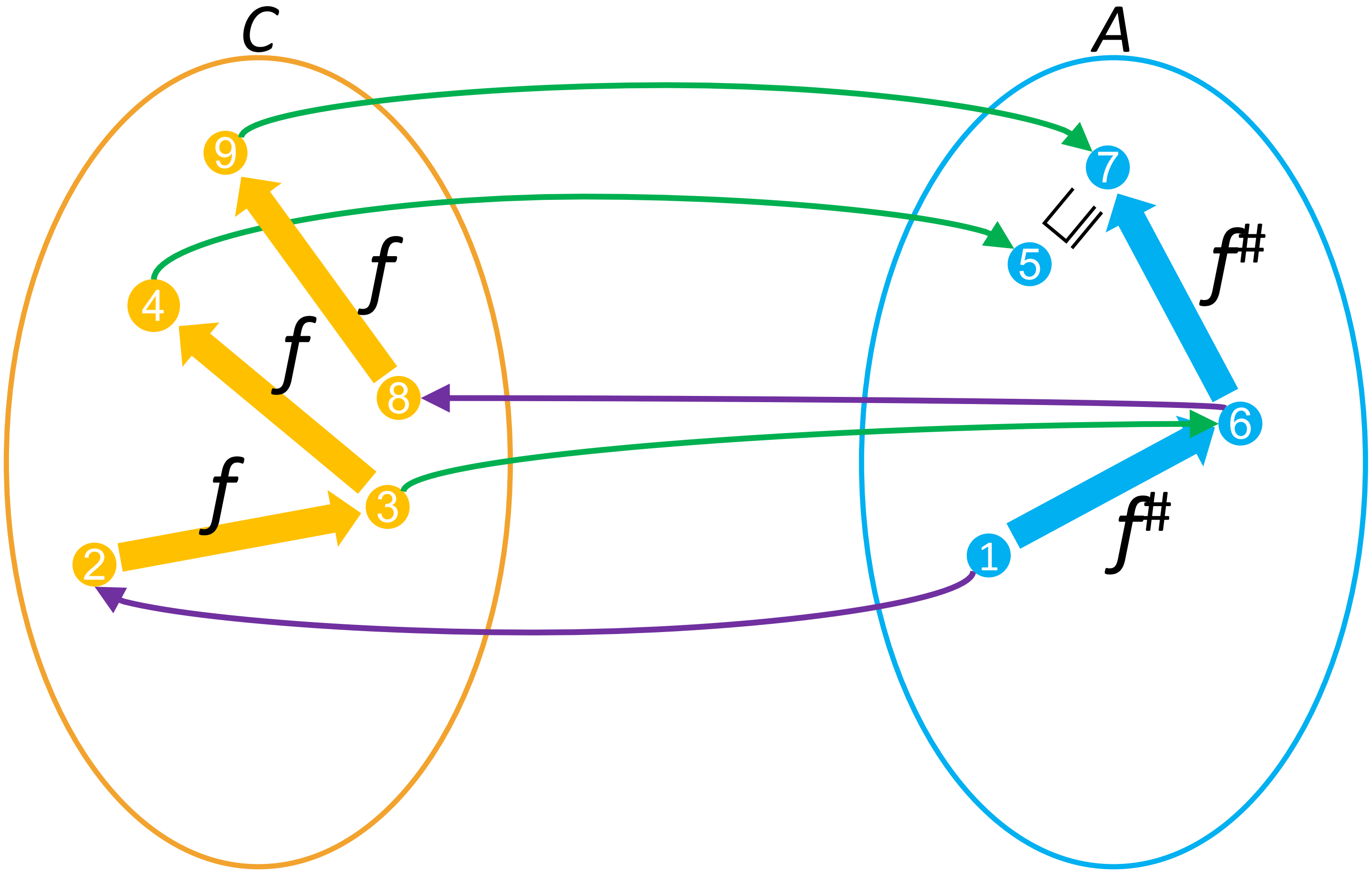
- $[\![x{:=}y]\!]^{\#2} \wedge(\{y{=}z\}) = \{x{=}y, y{=}z\} \sqsupseteq \{x{=}y, y{=}z, x{=}z\}$
- Idea: apply reduction operator again after the vanilla transformer
- $[\![x{:=}y]\!]^{\#3} = \text{Explicate} \circ [\![x{:=}y]\!]^{\#1} \circ \text{Explicate}$
- Observation : after the first time we apply Explicate, all subsequent values will be in the image of the abstraction so really we only need to apply it once to the input
- Finally: $[\![x{:=}y]\!]^{\#}(X) = \text{Explicate} \circ [\![x{:=}y]\!]^{\#1}$
  - Best transformer for reduced elements (elements in the image of the abstraction)

Negative property of best transformers

- Let $f^\# = \alpha \circ f \circ \gamma$
- Best transformer does not compose

$$\alpha(f(f(\gamma(a)))) \sqsubseteq f^\#(f^\#(a))$$

$$\alpha(f(f(\gamma(a)))) \sqsubseteq f^\#(f^\#(a))$$



Soundness theorem 1

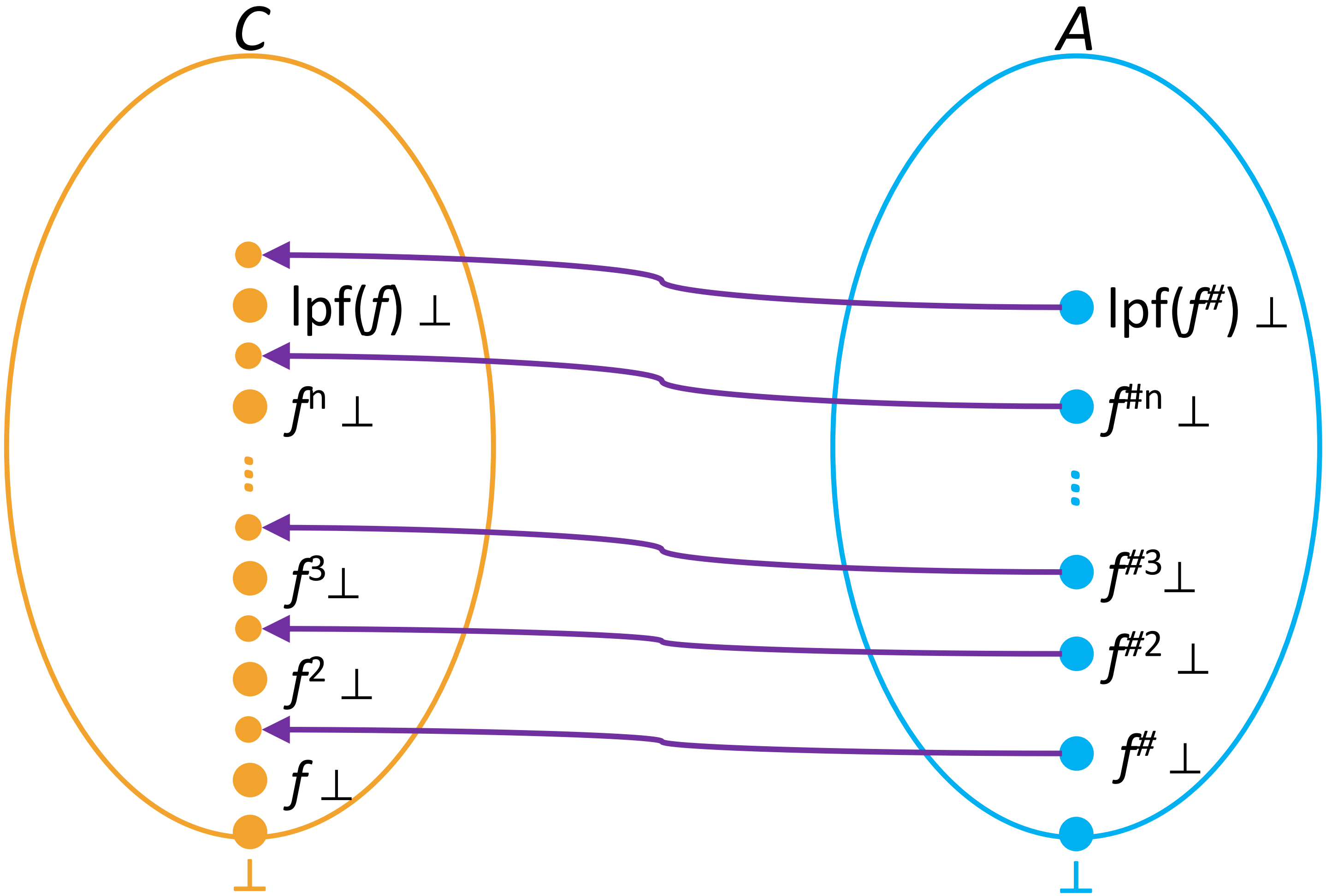
1. Given two complete lattices
   $C = (D^C, \sqsubseteq^C, \sqcup^C, \sqcap^C, \bot^C, \top^C)$
   $A = (D^A, \sqsubseteq^A, \sqcup^A, \sqcap^A, \bot^A, \top^A)$
   and $GC^{C,A}=(C, \alpha, \gamma, A)$ with
2. Monotone concrete transformer $f : D^C \rightarrow D^C$
3. Monotone abstract transformer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

Then

$$\mathrm{lfp}(f) \sqsubseteq \gamma(\mathrm{lfp}(f^\#))$$
$$\alpha(\mathrm{lfp}(f)) \sqsubseteq \mathrm{lfp}(f^\#)$$

Soundness theorem 1

$$
\begin{aligned}
\forall a \in D^A : f(\gamma(a)) \sqsubseteq \gamma(f^{\#}(a)) \quad & \Rightarrow \forall a \in D^A : f^n(\gamma(a)) \sqsubseteq \gamma(f^{\#n}(a)) \\
& \Rightarrow \forall a \in D^A : \mathrm{lfp}(f^n)(\gamma(a)) \sqsubseteq \gamma(\mathrm{lfp}(f^{\#n})(a)) \\
& \Rightarrow \mathrm{lfp}(f)\ \bot \sqsubseteq \mathrm{lfp}(f^{\#})\ \bot
\end{aligned}
$$



Soundness theorem 2

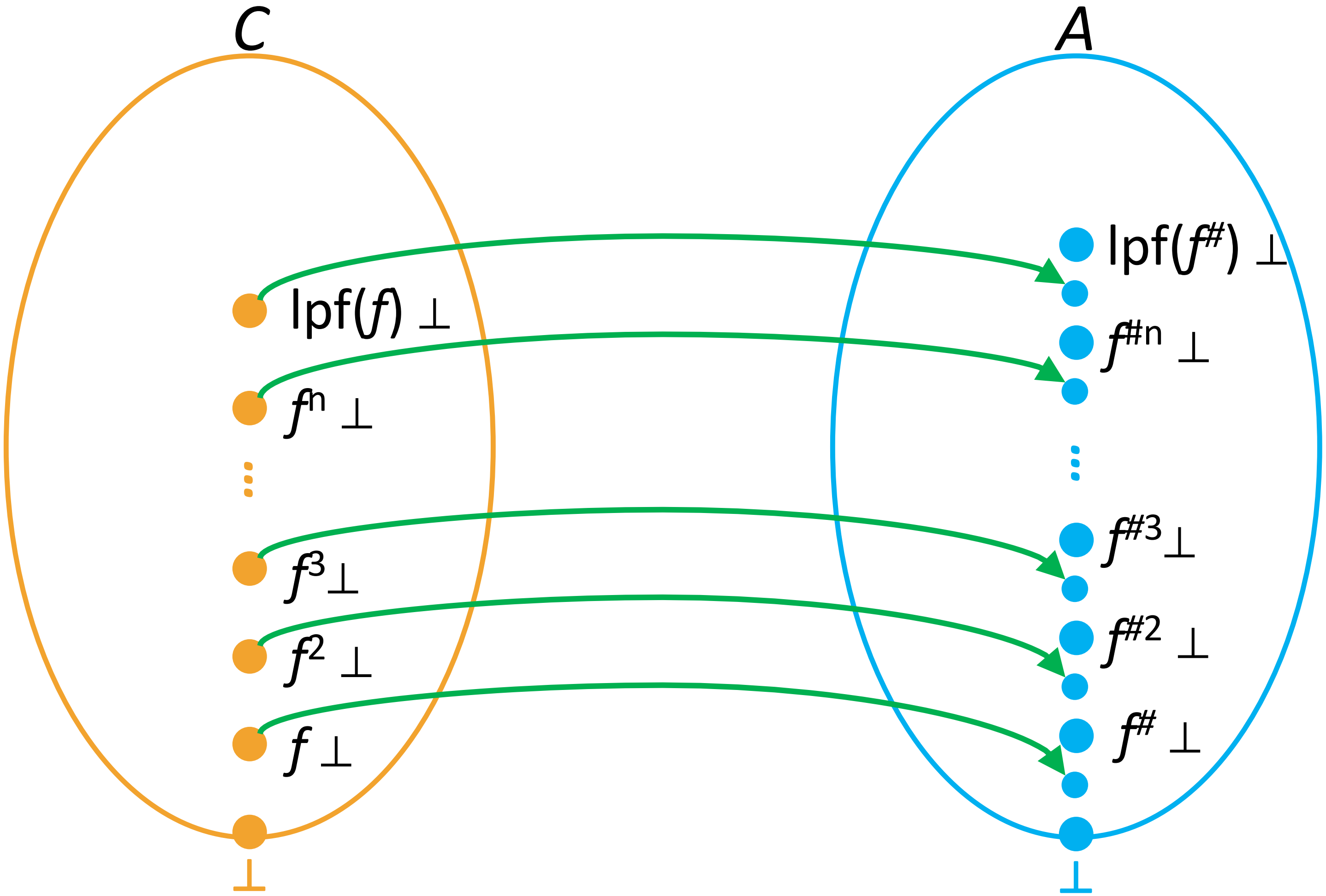
1. Given two complete lattices
   $C = (D^C, \sqsubseteq^C, \sqcup^C, \sqcap^C, \bot^C, \top^C)$
   $A = (D^A, \sqsubseteq^A, \sqcup^A, \sqcap^A, \bot^A, \top^A)$
   and $GC^{C,A}=(C, \alpha, \gamma, A)$ with
2. Monotone concrete transformer $f : D^C \rightarrow D^C$
3. Monotone abstract transformer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

Then

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

Soundness theorem 2

$$\begin{aligned}
\forall c\in D^C : \alpha(f(c)) \sqsubseteq f^{\#}(\alpha(c)) &\Rightarrow \forall c\in D^C : \alpha(f^{n}(c)) \sqsubseteq f^{\#n}(\alpha(c)) \\
&\Rightarrow \forall c\in D^C : \alpha(\mathrm{lfp}(f)(c)) \sqsubseteq \mathrm{lfp}(f^{\#})(\alpha(c)) \\
&\Rightarrow \mathrm{lfp}(f)\perp \sqsubseteq \mathrm{lfp}(f^{\#})\perp
\end{aligned}$$



A recipe for a sound static analysis

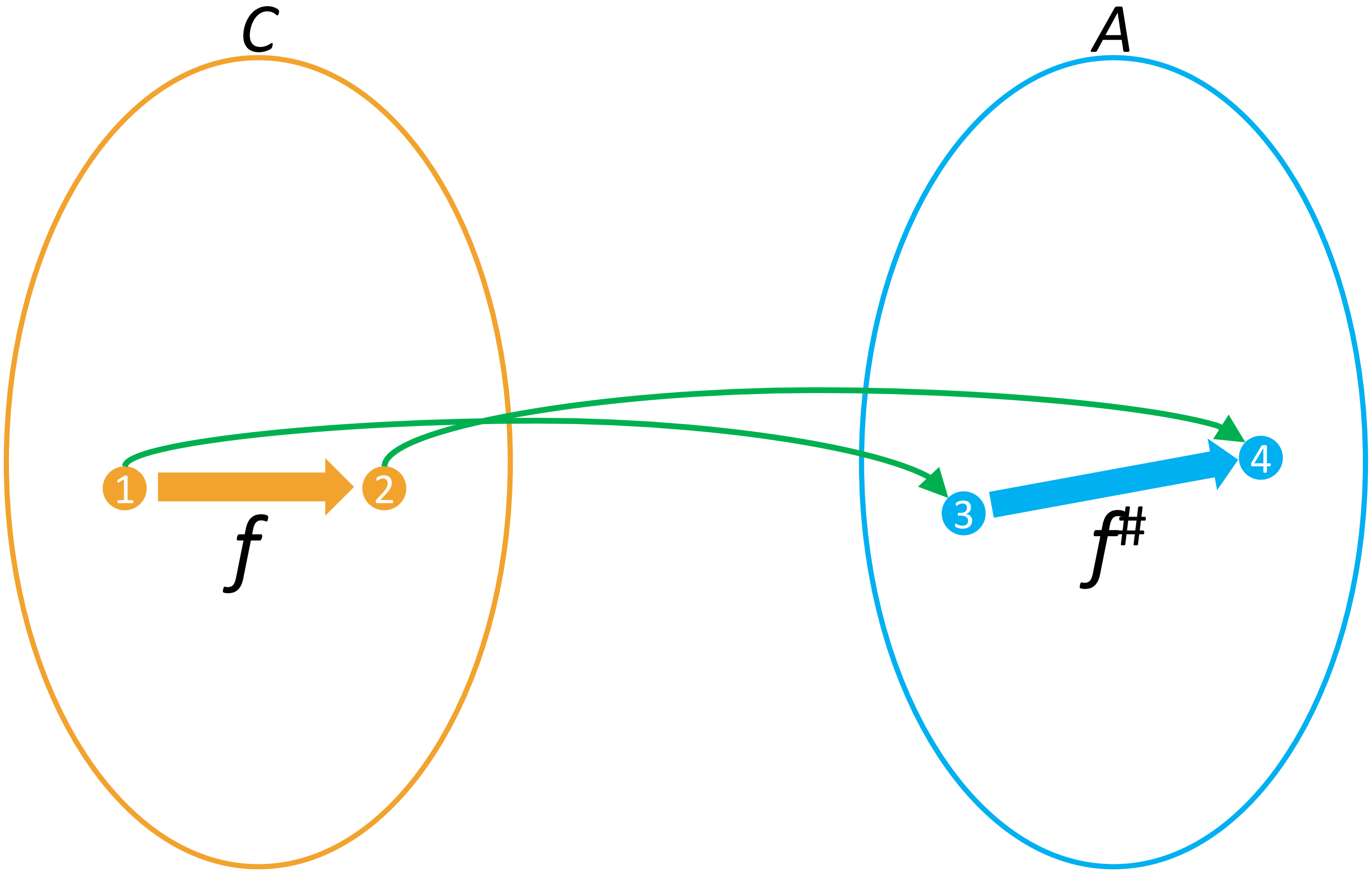
- Define an “appropriate” operational semantics
- Define “collecting” structural operational semantics
- Establish a Galois connection between collecting states and abstract states
- **Local correctness:** show that the abstract interpretation of every atomic statement is sound w.r.t. the collecting semantics
- **Global correctness:** conclude that the analysis is sound

Completeness

- Local property:
  - forward complete: $\forall c$: $\alpha(f^{\#}(c)) = \alpha(f(c))$
  - backward complete: $\forall a$: $f(\gamma(a)) = \gamma(f^{\#}(a))$
- A property of domain and the (best) transformer
- Global property:
  - $\alpha(\mathrm{lfp}(f)) = \mathrm{lfp}(f^{\#})$
  - $\mathrm{lfp}(f) = \gamma(\mathrm{lfp}(f^{\#}))$
- Very ideal but usually not possible unless we change the program model (apply strong abstraction) and/or aim for very simple properties

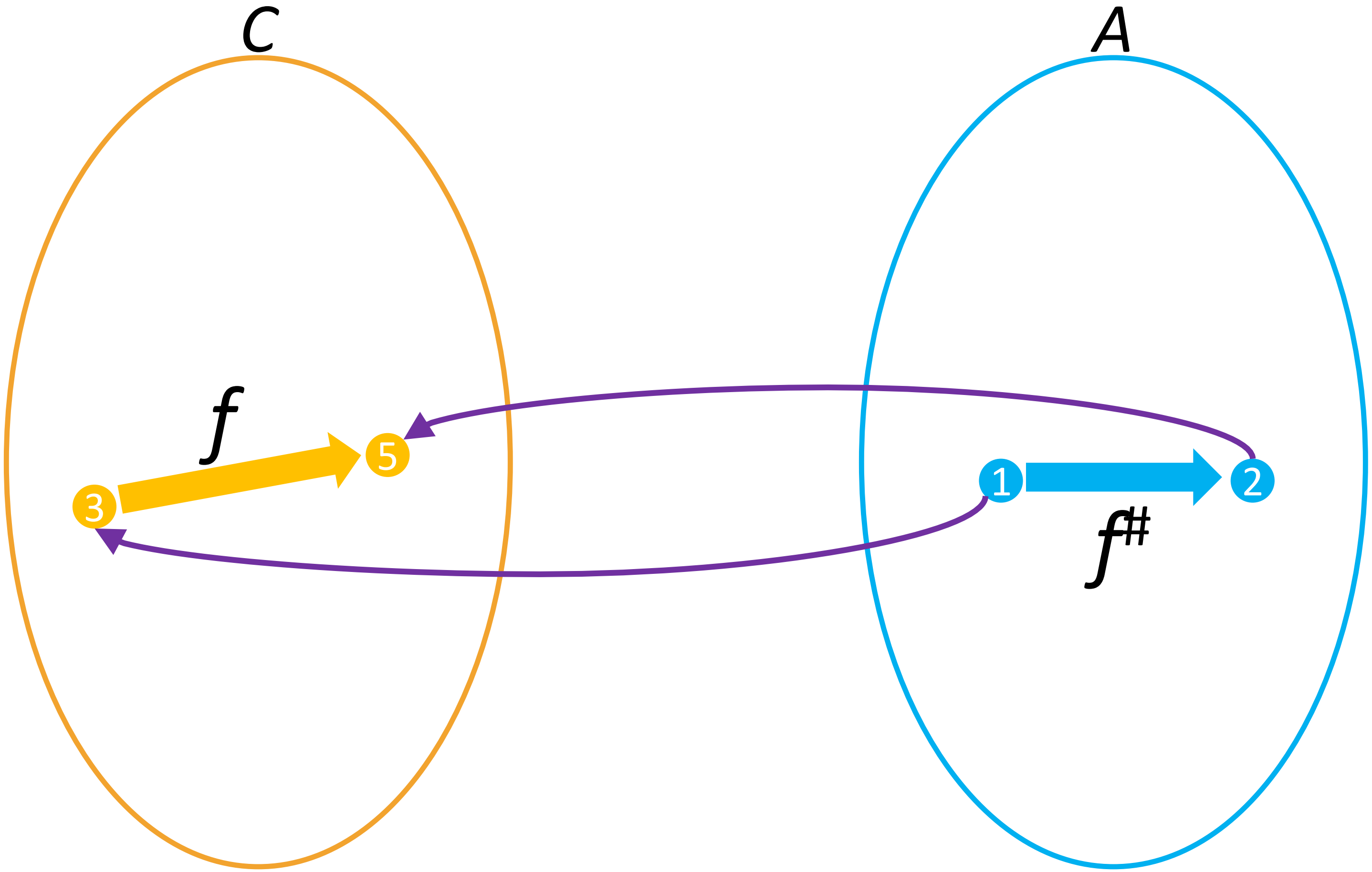
Forward complete transformer

$$\boxed{\forall c\colon \alpha(f^{\#}(c)) = \alpha(f(c))}$$



Backward complete transformer

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

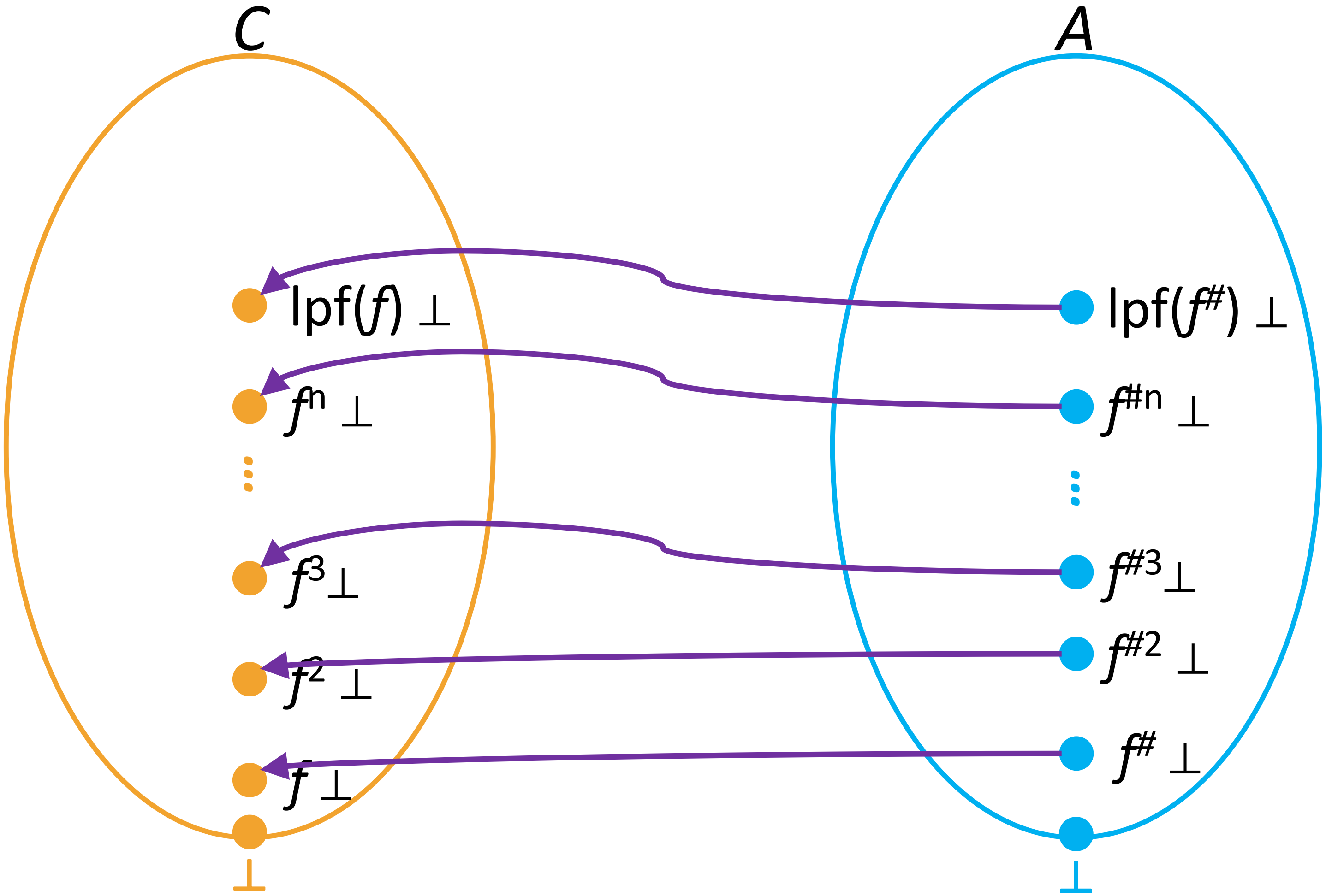


Global (backward) completeness

$\forall a: f(\gamma(a)) = \gamma(f^{\#}(a)) \quad \Rightarrow \forall a: f^{n}(\gamma(a)) = \gamma(f^{\#n}(a))$

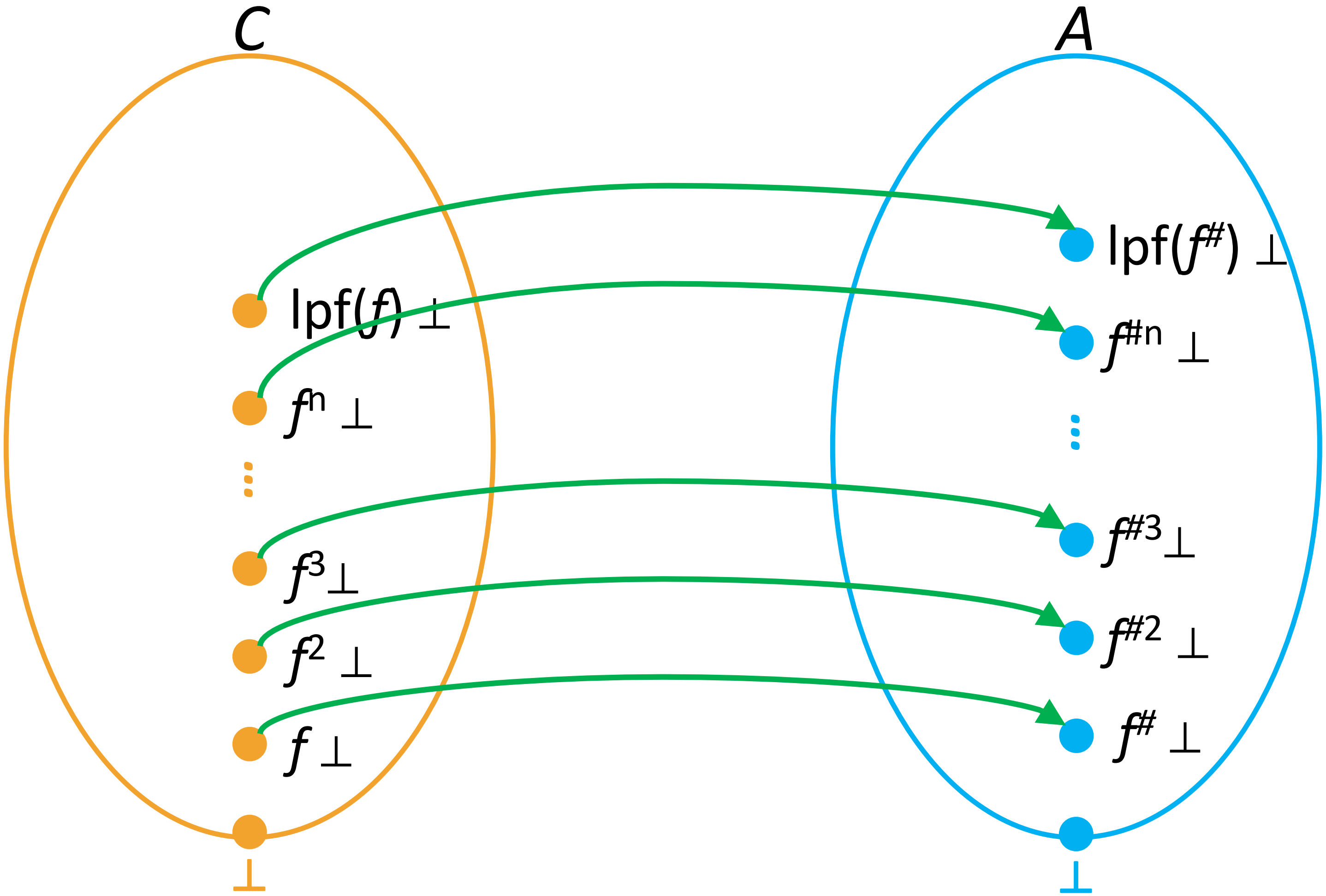
$\Rightarrow \forall a \in D^{A}: \mathrm{lfp}(f^{n})(\gamma(a)) = \gamma(\mathrm{lfp}(f^{\#n})(a))$

$\Rightarrow \mathrm{lfp}(f)\perp = \mathrm{lfp}(f^{\#})\perp$



Global (forward) completeness

$$\forall c \in D^C : \alpha(f(c)) = f^\#(\alpha(c)) \Rightarrow \forall c \in D^C : \alpha(f^n(c)) = f^{\#n}(\alpha(c))$$
$$\Rightarrow \forall c \in D^C : \alpha(\mathrm{lfp}(f)(c)) = \mathrm{lfp}(f^\#)(\alpha(c))$$
$$\Rightarrow \mathrm{lfp}(f)\perp = \mathrm{lfp}(f^\#)\perp$$



Example: Pointer Analysis

Plan

- Understand the problem
- Mention some applications
- Simplified problem
  - Only variables (no object allocation)
- Reference analysis
- Andersen’s analysis
- Steensgaard’s analysis
- Generalize to handle object allocation

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

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

x is always 3

pointers affect
most program analyses

```
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**
  - $PT\text{-}factoids[\mathrm{x}] = \{\, x{=}\&y \mid y \in \mathrm{Var}\} \cup \mathrm{false}$
    $PT[\mathrm{x}] = (2^{PT\text{-}factoids}, \subseteq, \cup, \cap, false, PT\text{-}factoids[\mathrm{x}])$
    (interpreted disjunctively)
- How should combine them to get the abstract states in the example?
  `{p=&x ∧ (q=&y∨q=&x) ∧ x=&a ∧ y=&b}`
- $D[\mathrm{x}] = \mathrm{Disj}(VE[\mathrm{x}]) \times \mathrm{Disj}(PT[\mathrm{x}])$
- 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** : $(Var \rightarrow Z) \cup (Var \rightarrow Var \cup \{null\})$
- $[\![x = y]\!]\, s =$
- $[\![x = {*y}]\!]\, s =$
- $[\![{*x} = y]\!]\, s =$
- $[\![x = null]\!]\, s =$
- $[\![x = \&y]\!]\, s =$

PWhile operational semantics

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

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/Must Point-To Analysis

may

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

must

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

Var' = Var U {null}

May-point-to analysis: formal Requirement specification

Compute $R: V \to 2^{Vars'}$ such that
$R(u) \supseteq IdealMayPT(u)$

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

$$\forall u \in V.\ R(u) \supseteq IdealMayPT(u)$$

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

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

- 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

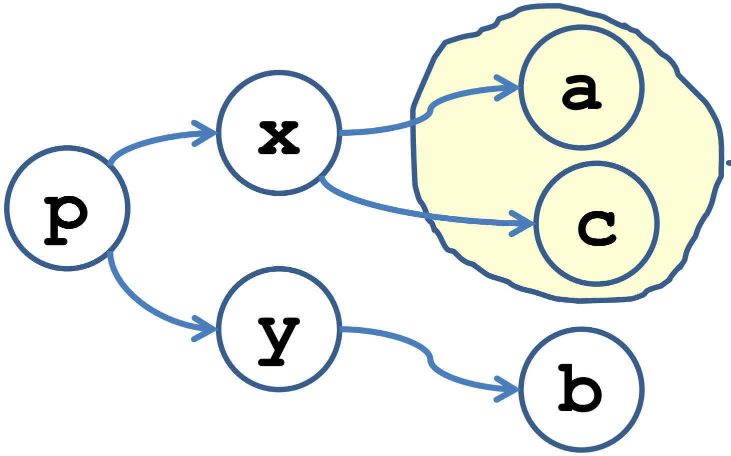
```
x = &a;
y = &b;
if (?) {
  p = &x;
} else {
  p = &y;
}

*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) ∧ a=&c}



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')
  - $g_1 \sqcup g_2$ = ?
  - $\perp$ = ?
  - $\top$ = ?
- 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
  - $[\![\text{stmt}]\!]^\#$: PTGraph → PTGraph
- $[\![\text{x := y}]\!]^\#$ (Var, E) = ?
- $[\![\text{x := null}]\!]^\#$ (Var, E) = ?
- $[\![\text{x := \&y}]\!]^\#$ (Var, E) = ?
- $[\![\text{x := *y}]\!]^\#$ (Var, E) = ?
- $[\![\text{*x := \&y}]\!]^\#$ (Var, E) = ?

Algorithm A: transformers

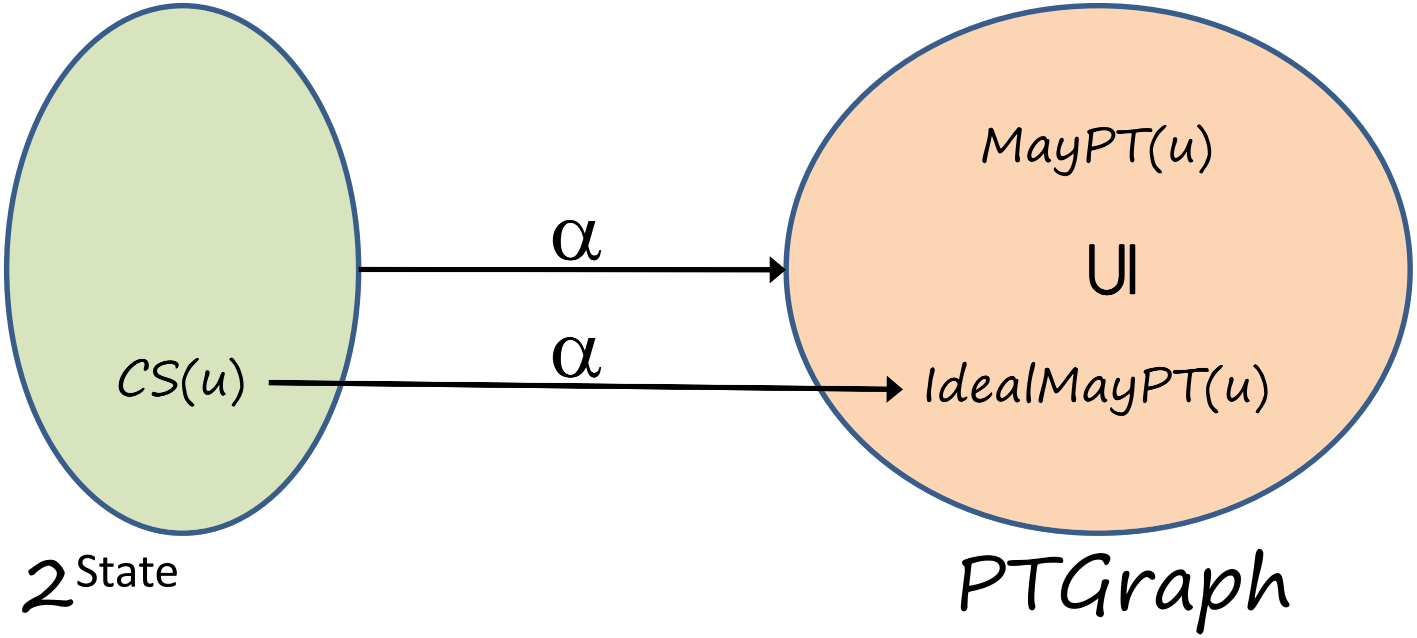
- Abstract transformers for primitive statements
  - $[\![$ stmt $]\!]^{\#}$: PTGraph → PTGraph
- $[\![$ x := y $]\!]^{\#}$ (Var, E) = (Var, E[succ(x)=succ(y)]
- $[\![$ x := null $]\!]^{\#}$ (Var, E) = (Var, E[succ(x)={null}]
- $[\![$ x := &y $]\!]^{\#}$ (Var, E) = (Var, E[succ(x)={y}]
- $[\![$ x :=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



$\alpha(Y) = \{ (p,x) \mid \text{exists } s \text{ in } Y.\ s(p) = x \}$

$\mathit{IdealMayPT}\ (u) = \alpha\ (\ \mathit{CS}(u)\ )$

Concrete transformers

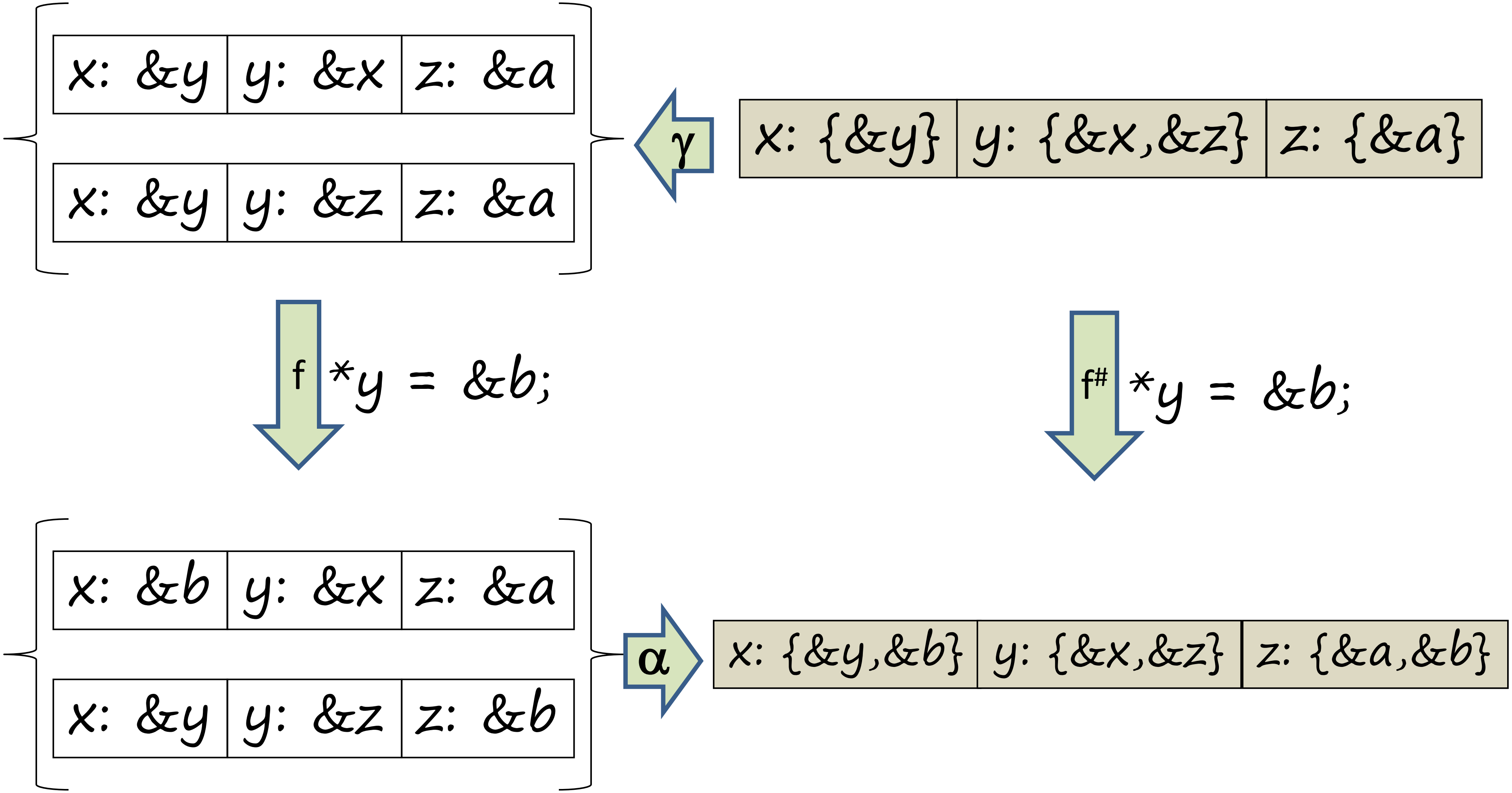
- CS[stmt] : State → State
- $[\![\, x = y \,]\!]\, s = s[x \mapsto s(y)]$
- $[\![\, x = {*}y \,]\!]\, s = s[x \mapsto s(s(y))]$
- $[\![\, {*}x = y \,]\!]\, s = s[s(x) \mapsto s(y)]$
- $[\![\, x = \text{null} \,]\!]\, s = s[x \mapsto \text{null}]$
- $[\![\, x = \&y \,]\!]\, s = s[x \mapsto y]$

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

Abstract transformers

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

Algorithm A: transformers
Weak/Strong Update



x: &y	y: &x	z: &a
x: &y	y: &z	z: &a

γ

x: {&y}	y: {&x,&z}	z: {&a}

f *y = &b;

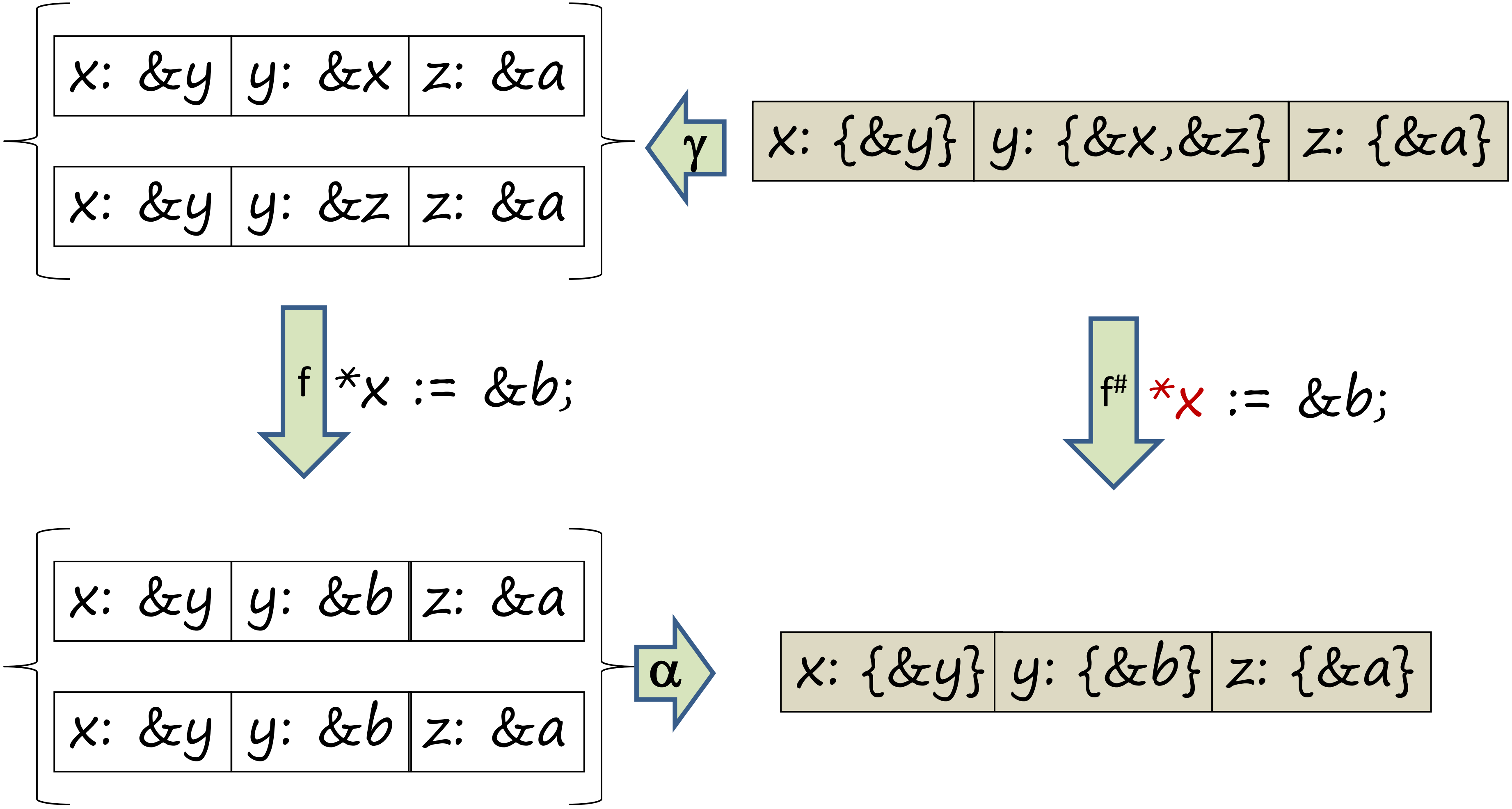
$f^{\#}$ *y = &b;

x: &b	y: &x	z: &a
x: &y	y: &z	z: &b

α

x: {&y,&b}	y: {&x,&z}	z: {&a,&b}

Algorithm A: transformers Weak/Strong Update



x: &y	y: &x	z: &a
x: &y	y: &z	z: &a

γ

x: {&y}	y: {&x,&z}	z: {&a}

f *x := &b;

f# *x := &b;

x: &y	y: &b	z: &a
x: &y	y: &b	z: &a

α

x: {&y}	y: {&b}	z: {&a}

Abstract transformers

- $[\![\, {*x} := \&y \,]\!]^{\#}$ (Var, E) =

  if succ(x) = {z} **then** (Var, E[succ(z)={y}]

  else succ(x)=$\{z_1,\ldots,z_k\}$ where k>1

  (Var, E[succ(z_1)=succ(z_1)$\cup${y}]

  …

  [succ(z_k)=succ(z_k)$\cup${y}]

Some dimensions of pointer analysis

- Intra-procedural / inter-procedural
- Flow-sensitive / flow-insensitive
- Context-sensitive / context-insensitive
- Definiteness
  - May vs. Must
- Heap modeling
  - Field-sensitive / field-insensitive
- Representation (e.g., Points-to graph)

Andersen’s Analysis

- A flow-insensitive analysis
  - Computes a single points-to solution valid at all program points
  - Ignores control-flow – treats program as a set of statements
  - Equivalent to merging all vertices into one (and applying *Algorithm A*)
  - Equivalent to adding an edge between every pair of vertices (and applying *Algorithm A*)

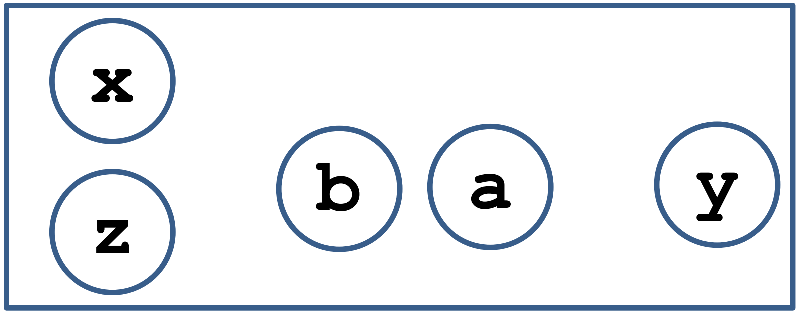
  - A (conservative) solution $R: \mathrm{Vars} \rightarrow 2^{\mathrm{Vars}'}$ such that

    $R \supseteq \mathrm{IdealMayPT}(u)$ for every vertex u

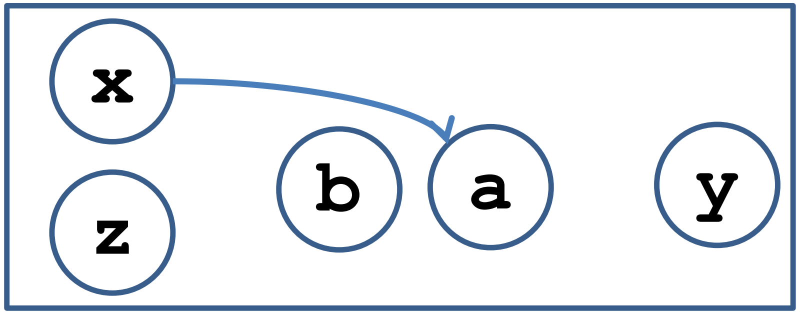
Flow-sensitive analysis

```
L1: x = &a;
L2: y = x;
L3: x = &b;
L4: z = x;
L5:
```

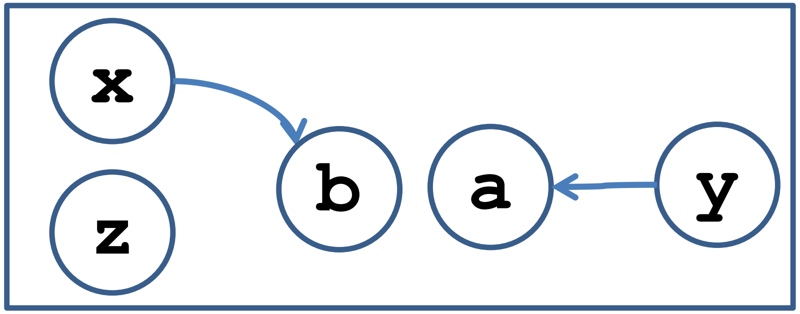
L1



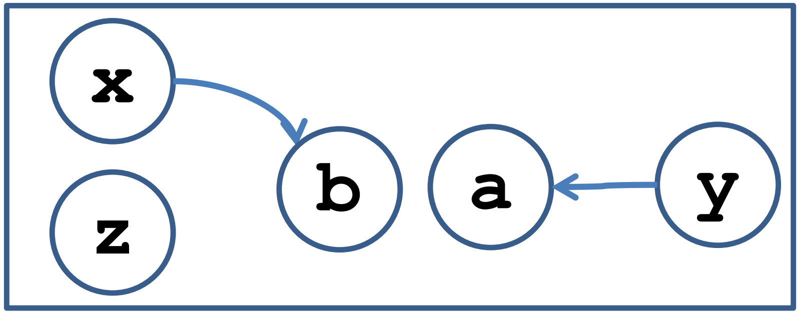
L2



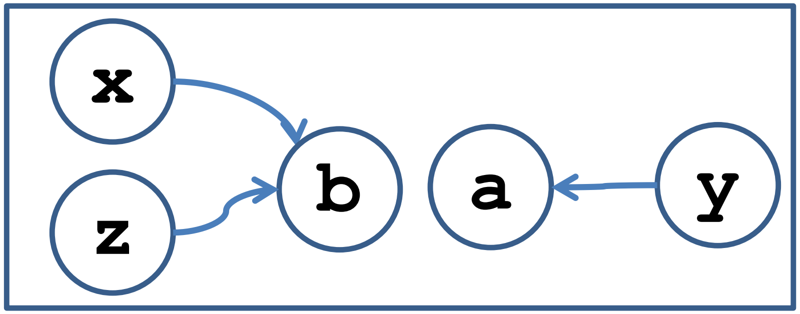
L3



L4

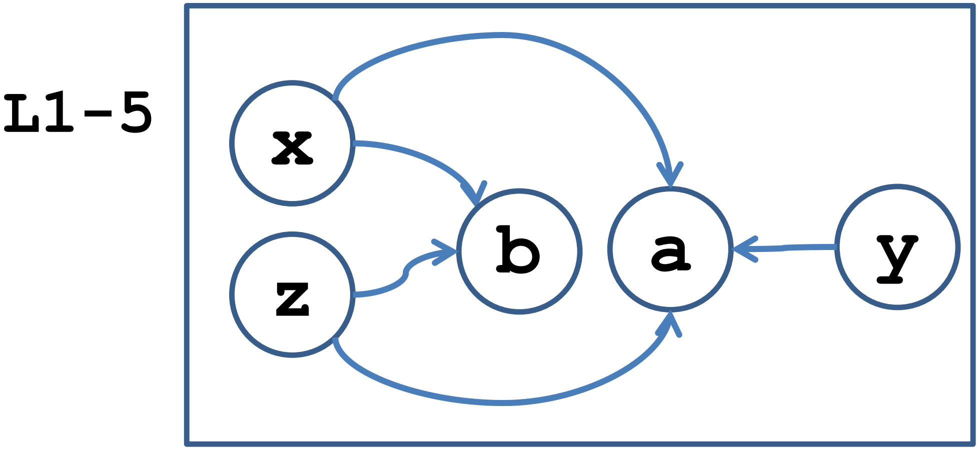


L5



Flow-insensitive analysis

```
L1: x = &a;
L2: y = x;
L3: x = &b;
L4: z = x;
L5:
```



Andersen’s analysis

- Strong updates?

- Initial state?

Why flow-insensitive analysis?

- Reduced space requirements
  - A single points-to solution
- Reduced time complexity
  - No copying
    - Individual updates more efficient
  - No need for joins
  - Number of iterations?
  - A cubic-time algorithm
- Scales to millions of lines of code
  - Most popular points-to analysis
- Conventionally used as an upper bound for precision for pointer analysis

Andersen's analysis as set constraints

- $[\![\text{x := y}]\!]^\#$ $PT[x] \supseteq PT[y]$
- $[\![\text{x := null}]\!]^\#$ $PT[x] \supseteq \{null\}$
- $[\![\text{x := \&y}]\!]^\#$ $PT[x] \supseteq \{y\}$
- $[\![\text{x := *y}]\!]^\#$ $PT[x] \supseteq PT[z]$ for all $z \in PT[y]$
- $[\![\text{*x := \&y}]\!]^\#$ $PT[z] \supseteq PT[y]$ for all $z \in PT[x]$

Cycle elimination

- Andersen-style pointer analysis is $O(n^3)$ for number of nodes in graph
  - Improve scalability by reducing n
- Important optimization
  - Detect strongly-connected components in PTGraph and collapse to a single node
    - Why? In the final result all nodes in SCC have same PT
  - How to detect cycles efficiently?
    - Some statically, some on-the-fly

Steensgaard’s Analysis

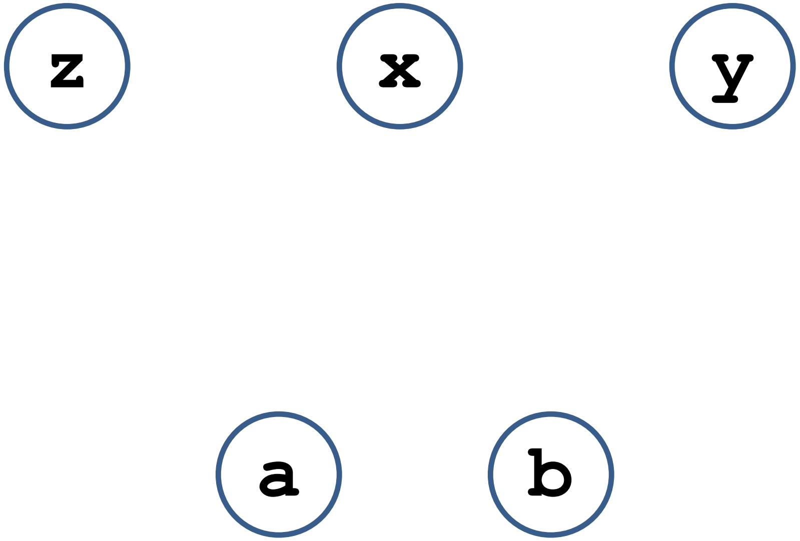
- Unification-based analysis
- Inspired by type inference
  - An assignment lhs := rhs is interpreted as a constraint that lhs and rhs have the same type
  - The type of a pointer variable is the set of variables it can point-to
- “Assignment-direction-insensitive”
  - Treats lhs := rhs as if it were both lhs := rhs
    and rhs := lhs

Steensgaard’s Analysis

- An almost-linear time algorithm
  - Uses union-find data structure
  - Single-pass algorithm; no iteration required
- Sets a lower bound in terms of performance

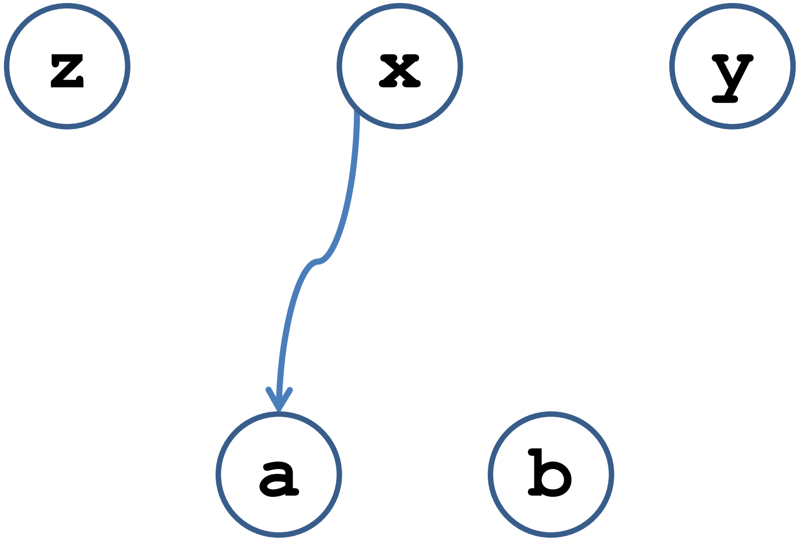
Steensgaard’s analysis initialization

```
L1: x = &a;
L2: y = x;
L3: x = &b;
L4: z = x;
L5:
```



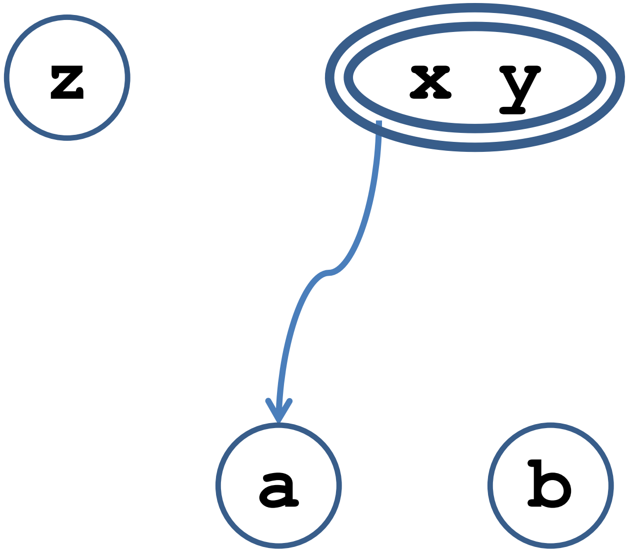
Steensgaard's analysis **`x=&a`**

```
L1: x = &a;
L2: y = x;
L3: x = &b;
L4: z = x;
L5:
```



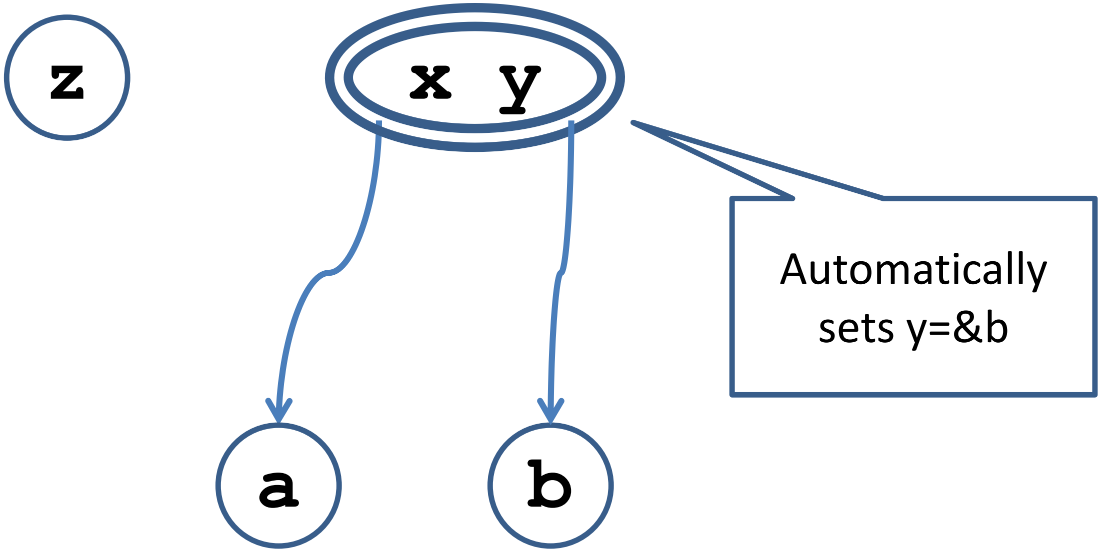
Steensgaard's analysis **`y=x`**

```
L1: x = &a;
L2: y = x;
L3: x = &b;
L4: z = x;
L5:
```



Steensgaard's analysis **`x=&b`**

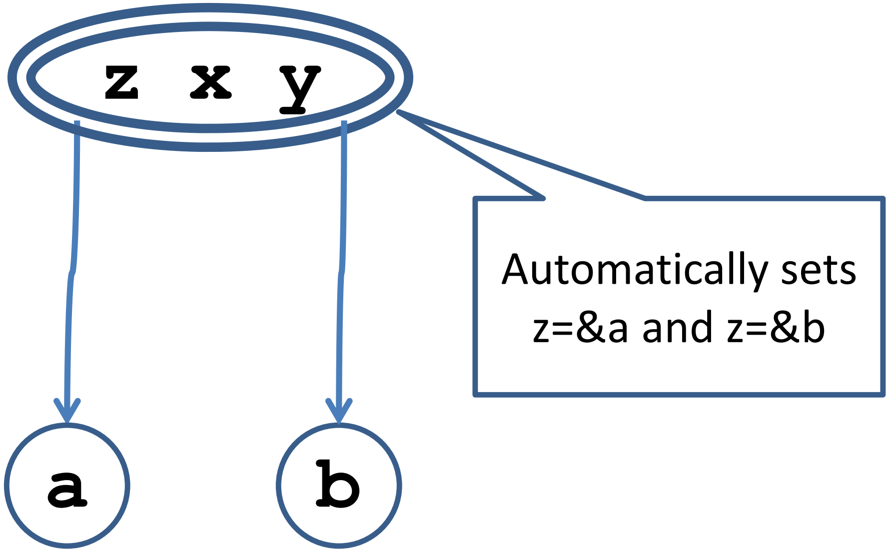
```
L1:  x = &a;
L2:  y = x;
L3:  x = &b;
L4:  z = x;
L5:
```



Automatically sets y=&b

Steensgaard’s analysis **`z=x`**

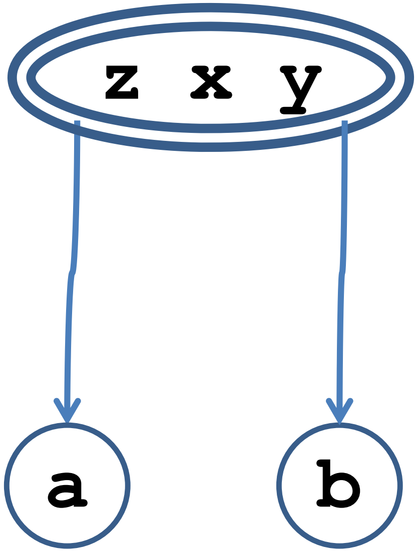
```
L1: x = &a;
L2: y = x;
L3: x = &b;
L4: z = x;
L5:
```



Automatically sets z=&a and z=&b

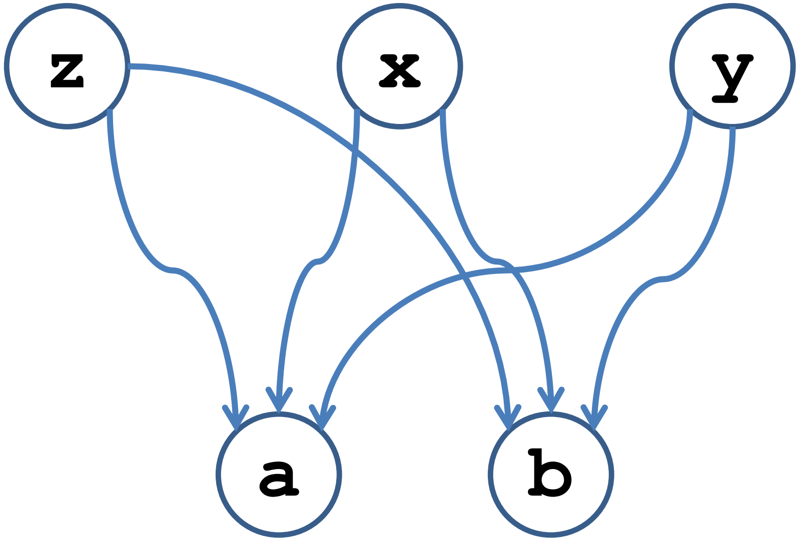
Steensgaard's analysis final result

```
L1: x = &a;
L2: y = x;
L3: x = &b;
L4: z = x;
L5:
```



Andersen’s analysis final result

```
L1: x = &a;
L2: y = x;
L3: x = &b;
L4: z = x;
L5:
```



Another example

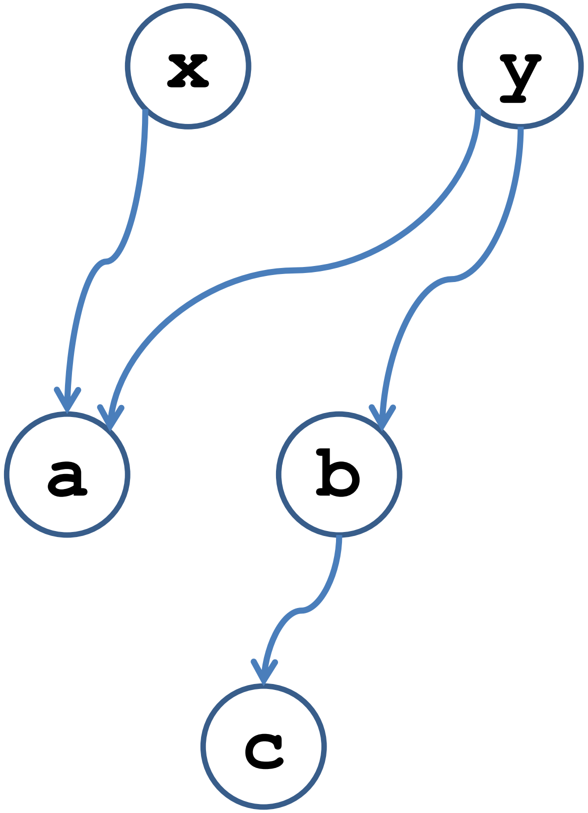
```
L1: x = &a;
L2: y = x;
L3: y = &b;
L4: b = &c;
L5:
```

Andersen’s analysis result = ?

```
L1: x = &a;
L2: y = x;
L3: y = &b;
L4: b = &c;
L5:
```

Another example

```
L1: x = &a;
L2: y = x;
L3: y = &b;
L4: b = &c;
L5:
```

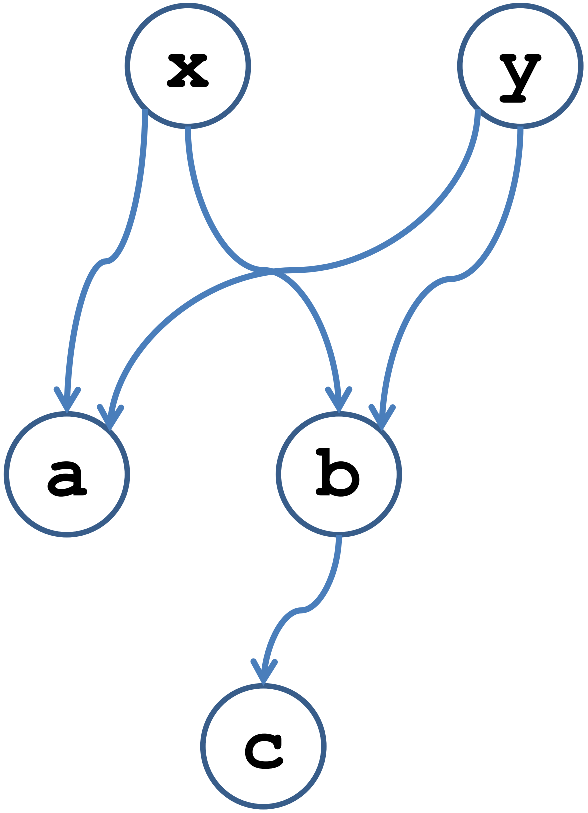


Steensgaard’s analysis result = ?

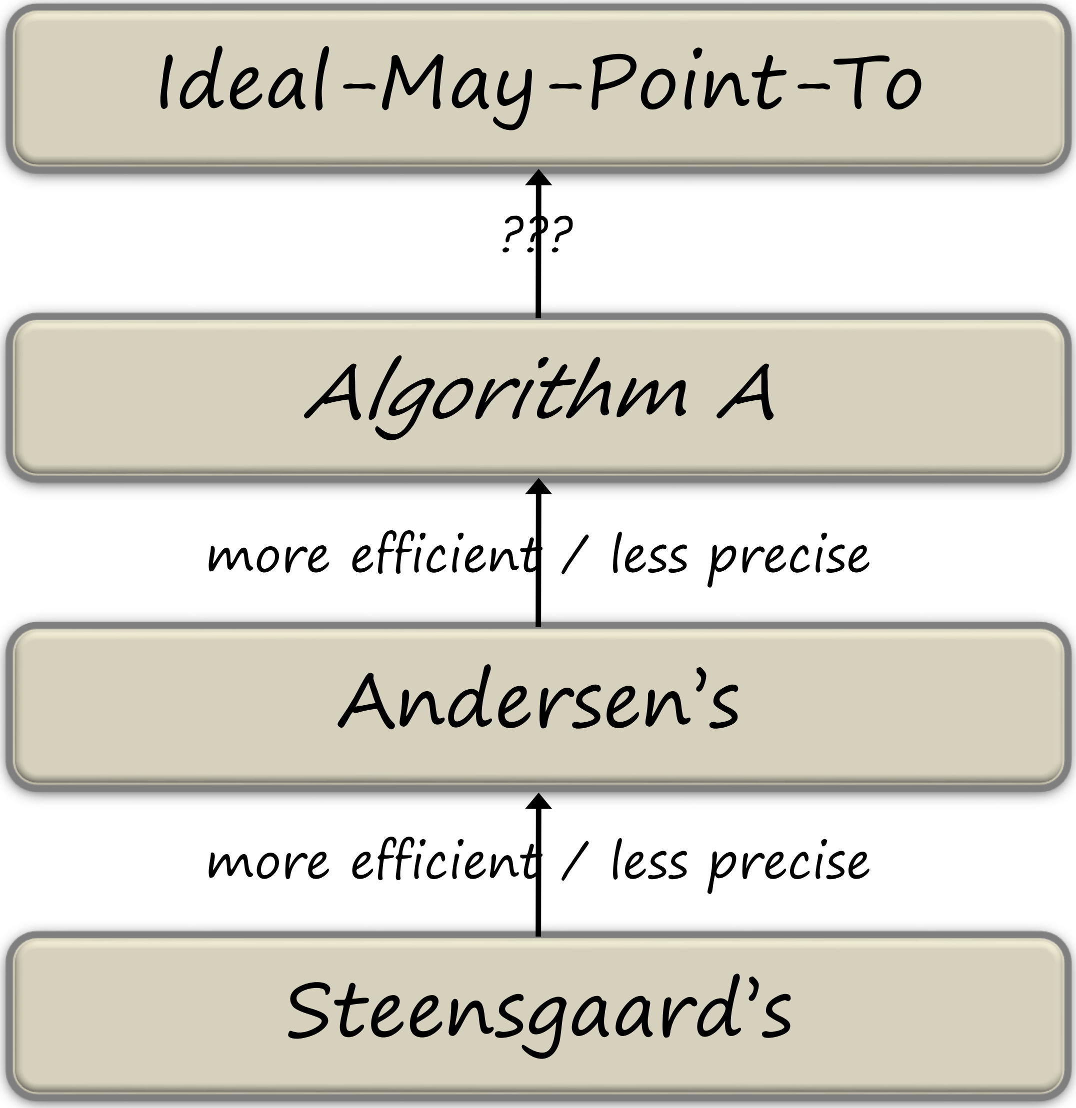
```
L1: x = &a;
L2: y = x;
L3: y = &b;
L4: b = &c;
L5:
```

Steensgaard’s analysis result =

```
L1: x = &a;
L2: y = x;
L3: y = &b;
L4: b = &c;
L5:
```



May-points-to analyses



Ideal-May-Point-To

???

Algorithm A

more efficient / less precise

Andersen's

more efficient / less precise

Steensgaard's

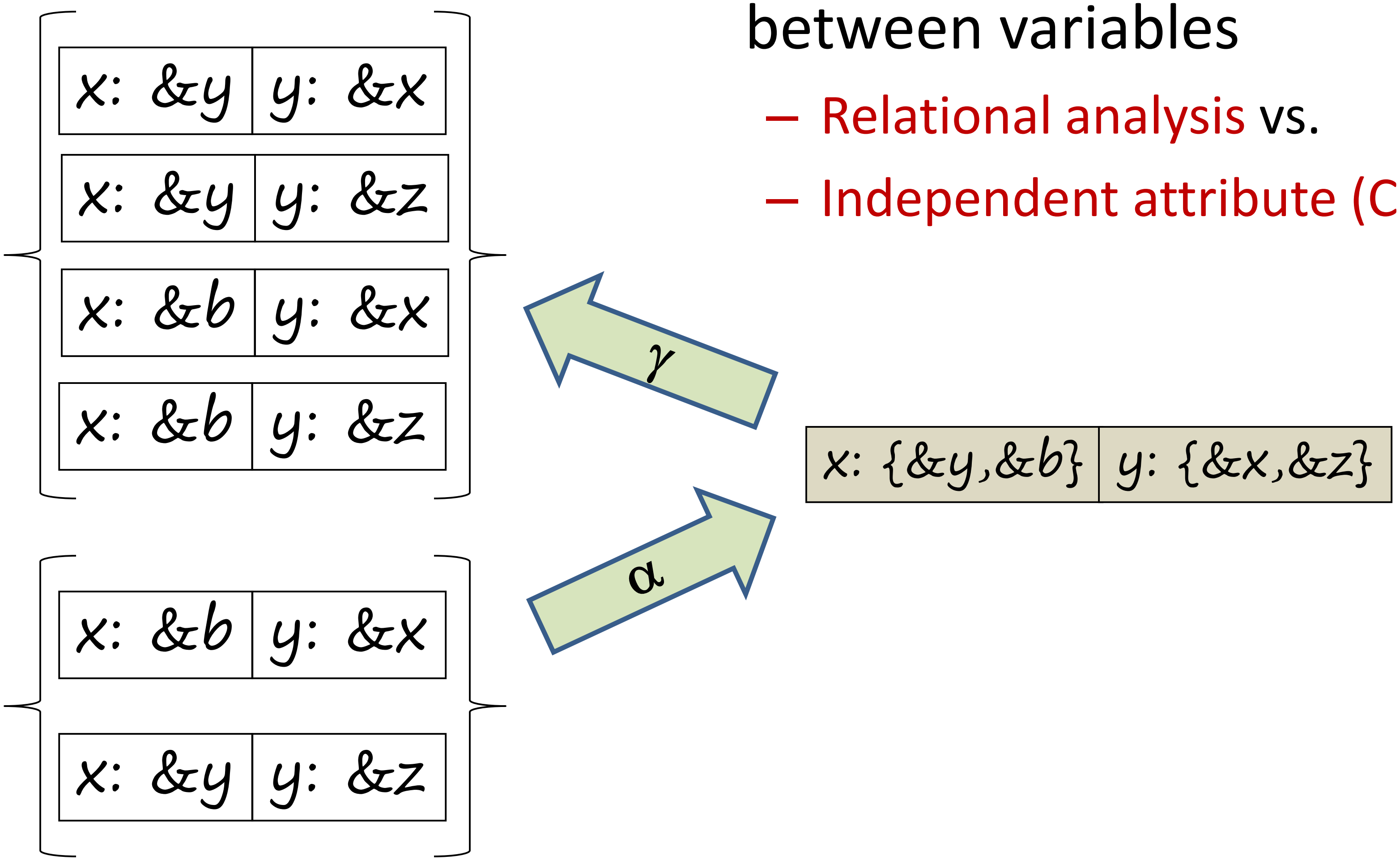
Ideal points-to analysis

- A sequence of states $s_1 s_2 \ldots s_n$ is said to be an execution (of the program) iff
  - s_1 is the Initial-State
  - $s_i \rightarrow s_{i+1}$ for 1 <= I < n
- A state s is said to be a reachable state iff there exists some execution $s_1 s_2 \ldots s_n$ is such that $s_n = s$.
- CS(u) = { s | (u,s) is reachable }
- IdealMayPT (u) = { (p,x) | ∃ s ∈ CS(u). s(p) = x }
- IdealMustPT (u) = { (p,x) | ∀ s ∈ CS(u). s(p) = x }

Does *Algorithm A* compute the most precise solution?

Ideal vs. *Algorithm A*

- Abstracts away correlations between variables
  - Relational analysis vs.
  - Independent attribute (Cartesian)



Does *Algorithm A* compute the most precise solution?

Is the precise solution computable?

- Claim: The set CS(u) of reachable concrete states (for our language) is computable

- Note: This is true for any collecting semantics with a finite state space

Computing CS(u)

Precise points-to analysis: decidability

- Corollary: Precise may-point-to analysis is computable.
- Corollary: Precise (demand) may-alias analysis is computable.
  - Given ptr-exp1, ptr-exp2, and a program point u, identify if there exists some reachable state at u where ptr-exp1 and ptr-exp2 are aliases.
- Ditto for must-point-to and must-alias
- … for our restricted language!

Precise Points-To Analysis: Computational Complexity

- What’s the complexity of the least-fixed point computation using the collecting semantics?

- The worst-case complexity of computing reachable states is exponential in the number of variables.
  - Can we do better?

- Theorem: Computing precise may-point-to is PSPACE-hard even if we have only two-level pointers

May-Point-To Analyses

Ideal-May-Point-To

more efficient / less precise

Algorithm A

more efficient / less precise

Andersen's

more efficient / less precise

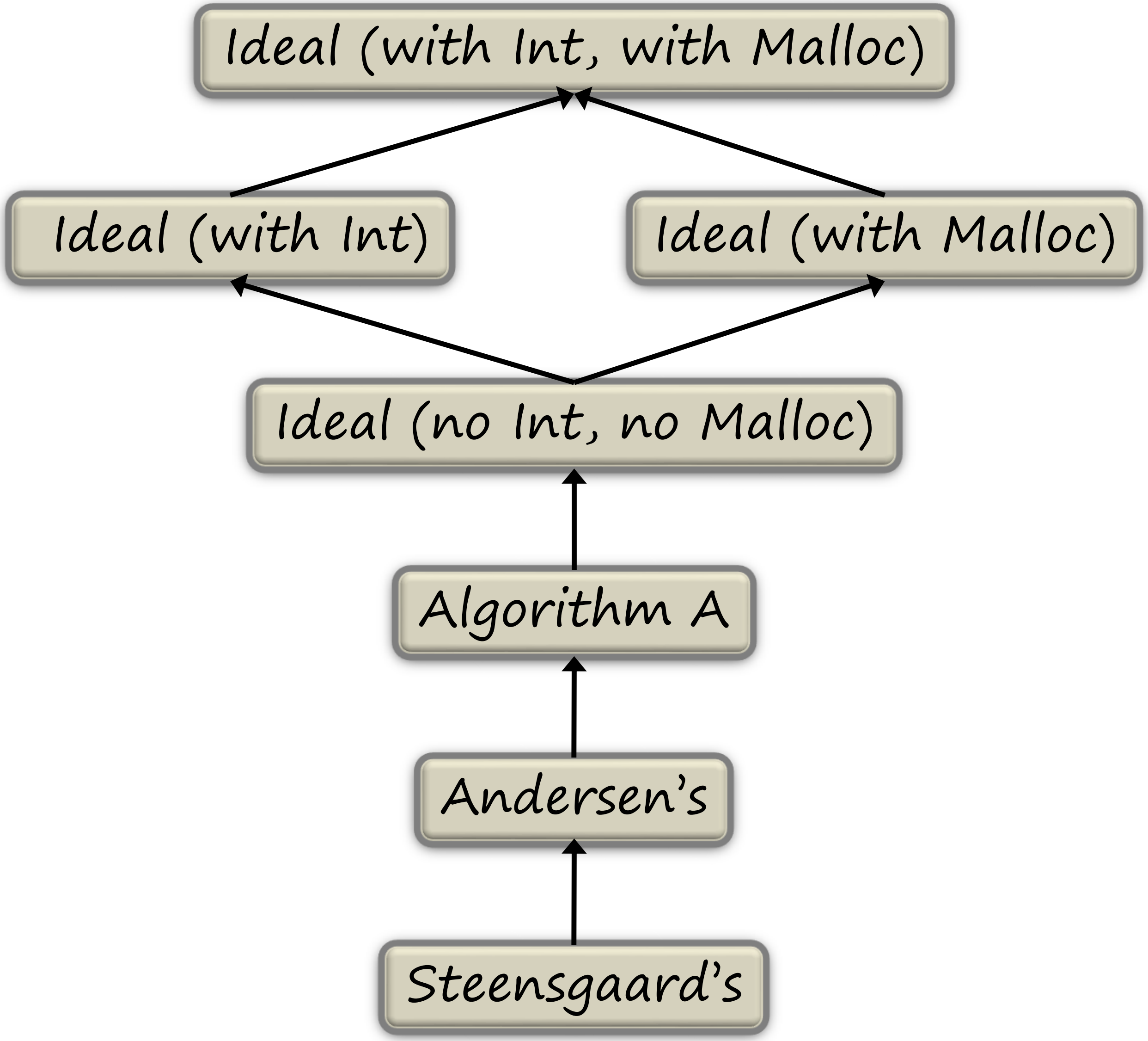
Steensgaard's

Precise points-to analysis: caveats

- Theorem: Precise may-alias analysis is undecidable in the presence of dynamic memory allocation
  - Add “x = new/malloc ()” to language
  - State-space becomes infinite

- Digression: Integer variables + conditional-branching also makes any precise analysis undecidable

High-level classification



Ideal (with Int, with Malloc)

Ideal (with Int)

Ideal (with Malloc)

Ideal (no Int, no Malloc)

Algorithm A

Andersen's

Steensgaard's

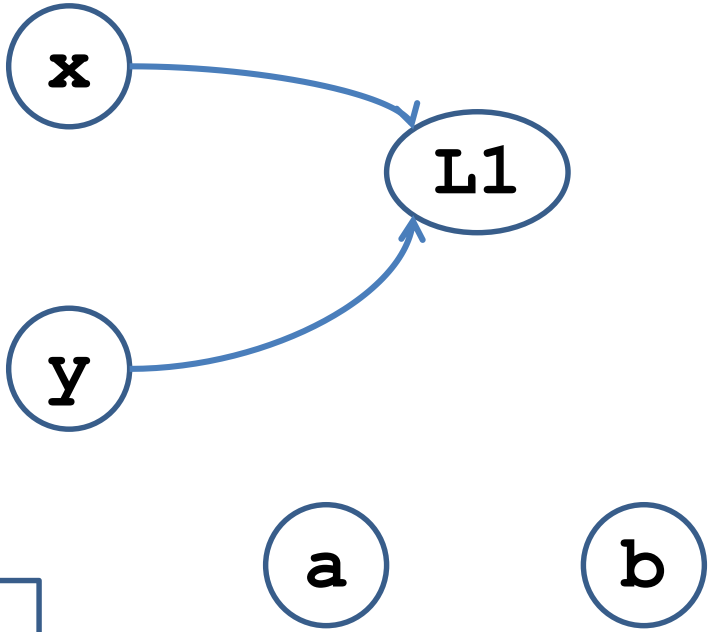
Handling memory allocation

- s: x = new () / malloc ()
- Assume, for now, that allocated object stores one pointer
  - s: x = malloc (sizeof(void*))
- Introduce a pseudo-variable V_s to represent objects allocated at statement s, and use previous algorithm
  - Treat s as if it were "x = &V_s"
  - Also track possible values of V_s
  - Allocation-site based approach
- Key aspect: V_s represents a set of objects (locations), not a single object
  - referred to as a summary object (node)

Dynamic memory allocation example

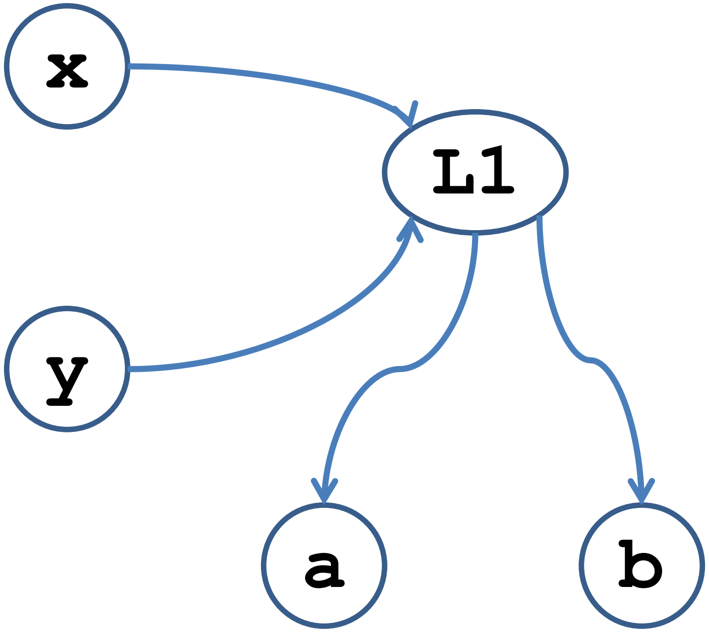
```
L1: x = new O;
L2: y = x;
L3: *y = &b;
L4: *y = &a;
```

How should we handle these statements



Summary object update

```
L1: x = new O;
L2: y = x;
L3: *y = &b;
L4: *y = &a;
```



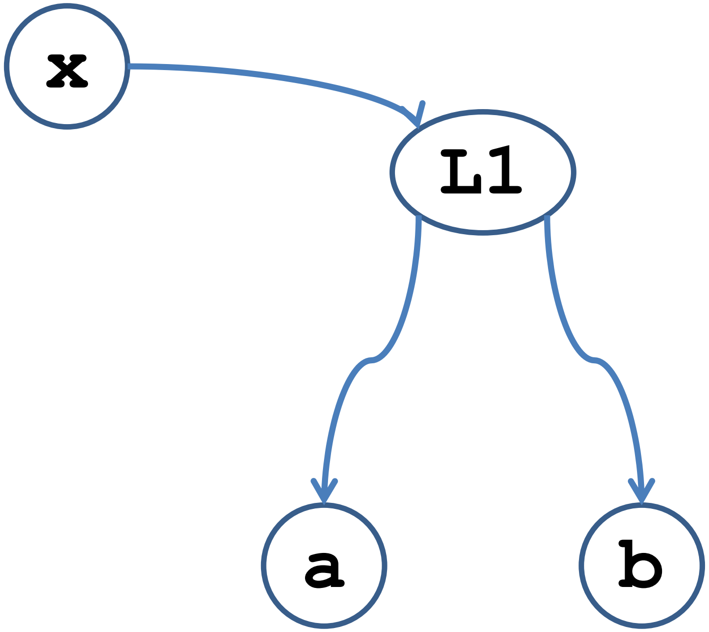
Object fields

- Field-insensitive analysis

```
class Foo {
    A* f;
    B* g;
}
L1: x = new Foo()

x->f = &b;

x->g = &a;
```



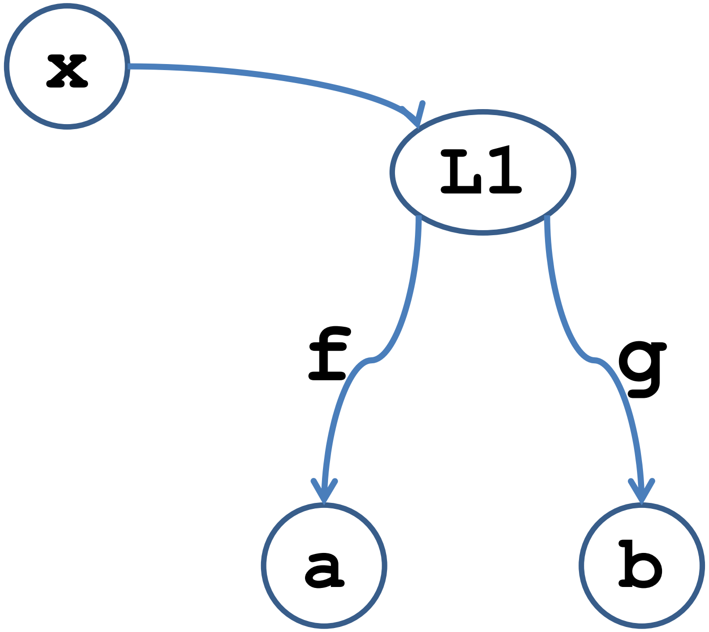
Object fields

- Field-sensitive analysis

```
class Foo {
    A* f;
    B* g;
}
L1: x = new Foo()

x->f = &b;

x->g = &a;
```



Other Aspects

- Context-sensitivity
- Indirect (virtual) function calls and call-graph construction
- Pointer arithmetic
- Object-sensitivity

Combining abstract domains

Three example analyses

- Abstract states are conjunctions of constraints
- **V**ariable **E**qualities
  - *VE-factoids* = { x=y | x, $y \in$ Var} $\cup$ false
    VE = ($2^{VE\text{-}factoids}$, $\supseteq$, $\cap$, $\cup$, *false*, $\varnothing$)
- **C**onstant **P**ropagation
  - *CP-factoids* = { x=c | $x \in$ Var, $c \in \mathbf{Z}$} $\cup$ false
    CP = ($2^{CP\text{-}factoids}$, $\supseteq$, $\cap$, $\cup$, *false*, $\varnothing$)
- **A**vailable **E**xpressions
  - *AE-factoids* = { x=y+z | $x \in$ Var, y,$z \in$ Var$\cup\mathbf{Z}$} $\cup$ false
    A = ($2^{AE\text{-}factoids}$, $\supseteq$, $\cap$, $\cup$, *false*, $\varnothing$)

Lattice combinators reminder

- Cartesian Product
  - $L_1 = (D_1, \sqsubseteq_1, \sqcup_1, \sqcap_1, \bot_1, \top_1)$
    $L_2 = (D_2, \sqsubseteq_2, \sqcup_2, \sqcap_2, \bot_2, \top_2)$
  - $\mathrm{Cart}(L_1, L_2) = (D_1 \times D_2, \sqsubseteq_{cart}, \sqcup_{cart}, \sqcap_{cart}, \bot_{cart}, \top_{cart})$
- Disjunctive completion
  - $L = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$
  - $\mathrm{Disj}(L) = (2^D, \sqsubseteq_\vee, \sqcup_\vee, \sqcap_\vee, \bot_\vee, \top_\vee)$
- Relational Product
  - $\mathrm{Rel}(L_1, L_2) = \mathrm{Disj}(\mathrm{Cart}(L_1, L_2))$

Cartesian product of complete lattices

- For two complete lattices
  $L_1 = (D_1, \sqsubseteq_1, \sqcup_1, \sqcap_1, \bot_1, \top_1)$
  $L_2 = (D_2, \sqsubseteq_2, \sqcup_2, \sqcap_2, \bot_2, \top_2)$
- Define the poset
  $L_{cart} = (D_1 \times D_2, \sqsubseteq_{cart}, \sqcup_{cart}, \sqcap_{cart}, \bot_{cart}, \top_{cart})$
  as follows:
  - $(x_1, x_2) \sqsubseteq_{cart} (y_1, y_2)$ iff
    $x_1 \sqsubseteq_1 y_1$ and
    $x_2 \sqsubseteq_2 y_2$
  - $\sqcup_{cart} = ?$ $\sqcap_{cart} = ?$ $\bot_{cart} = ?$ $\top_{cart} = ?$
- **Lemma:** L is a complete lattice
- Define the Cartesian constructor $L_{cart} = \text{Cart}(L_1, L_2)$

Cartesian product of GCs

- $GC^{C,A}=(C, \alpha^{C,A}, \gamma^{A,C}, A)$
  $GC^{C,B}=(C, \alpha^{C,B}, \gamma^{B,C}, B)$
- Cartesian Product

  $$GC^{C,A\times B} = (C, \alpha^{C,A\times B}, \gamma^{A\times B,C}, A\times B)$$

  - $\alpha^{C,A\times B}(X)=$?
  - $\gamma^{A\times B,C}(Y) =$?

Cartesian product of GCs

- $GC^{C,A}=(C, \alpha^{C,A}, \gamma^{A,C}, A)$
  $GC^{C,B}=(C, \alpha^{C,B}, \gamma^{B,C}, B)$
- Cartesian Product
  $$GC^{C,A\times B} = (C, \alpha^{C,A\times B}, \gamma^{A\times B,C}, A\times B)$$
  - $\alpha^{C,A\times B}(X) = (\alpha^{C,A}(X), \alpha^{C,B}(X))$
  - $\gamma^{A\times B,C}(Y) = \gamma^{A,C}(X) \cap \gamma^{B,C}(X)$
- What about transformers?

Cartesian product transformers

- $GC^{C,A}$=(C, $\alpha^{C,A}$, $\gamma^{A,C}$, A) F^{A}[st] : $A \rightarrow A$
  $GC^{C,B}$=(C, $\alpha^{C,B}$, $\gamma^{B,C}$, B) F^{B}[st] : $B \rightarrow B$
- Cartesian Product
  $$GC^{C,A\times B} = (C, \alpha^{C,A\times B}, \gamma^{A\times B,C}, A\times B)$$
  - $\alpha^{C,A\times B}(X) = (\alpha^{C,A}(X), \alpha^{C,B}(X))$
  - $\gamma^{A\times B,C}(Y) = \gamma^{A,C}(X) \cap \gamma^{B,C}(X)$
- How should we define $F^{A\times B}$[st] : $A\times B \rightarrow A\times B$

Cartesian product transformers

- $GC^{C,A}=(C, \alpha^{C,A}, \gamma^{A,C}, A)$ $F^{A}[st] : A \rightarrow A$
  $GC^{C,B}=(C, \alpha^{C,B}, \gamma^{B,C}, B)$ $F^{B}[st] : B \rightarrow B$
- Cartesian Product
  $$GC^{C,A\times B} = (C, \alpha^{C,A\times B}, \gamma^{A\times B,C}, A\times B)$$
  - $\alpha^{C,A\times B}(X) = (\alpha^{C,A}(X), \alpha^{C,B}(X))$
  - $\gamma^{A\times B,C}(Y) = \gamma^{A,C}(X) \cap \gamma^{B,C}(X)$
- How should we define $F^{A\times B}[st] : A\times B \rightarrow A\times B$
- Idea: $F^{A\times B}[st](a, b) = (F^{A}[st]\ a, F^{B}[st]\ b)$
- Are component-wise transformers precise?

Cartesian product analysis example

- Abstract interpreter 1: **C**onstant **P**ropagation
- Abstract interpreter 2: **V**ariable **E**qualities
- Let’s compare
  - Running them separately and combining results
  - Running the analysis with their Cartesian product

CP analysis

```
a := 9;      {a=9}
b := 9;      {a=9, b=9}
c := a;      {a=9, b=9, c=9}
```

VE analysis

```
a := 9;      {}
b := 9;      {}
c := a;      {c=a}
```

Cartesian product analysis example

- Abstract interpreter 1: **C**onstant **P**ropagation
- Abstract interpreter 2: **V**ariable **E**qualities
- Let's compare
  - Running them separately and combining results
  - Running the analysis with their Cartesian product

CP analysis + VE analysis

```
a := 9;       {a=9}
b := 9;       {a=9, b=9}
c := a;       {a=9, b=9, c=9, c=a}
```

Cartesian product analysis example

- Abstract interpreter 1: **C**onstant **P**ropagation
- Abstract interpreter 2: **V**ariable **E**qualities
- Let's compare
  - Running them separately and combining results
  - Running the analysis with their Cartesian product

CP×VE analysis

```
a := 9;      {a=9}
b := 9;      {a=9, b=9}
c := a;      {a=9, b=9, c=9, c=a} {a=b, b=c}
```

Missing

Transformers for Cartesian product

- Naïve (component-wise) transformers do not utilize information from both components
  - Same as running analyses separately and then combining results
- Can we treat transformers from each analysis as black box and obtain best transformer for their combination?

Can we combine transformer modularly?

- No generic method for any abstract interpretations

Reducing values for CP×VE

- X = set of CP constraints of the form x=c (e.g., **`a=9`**)
- Y = set of VE constraints of the form x=y
- Reduce$^{\mathrm{CP}\times\mathrm{VE}}$($X$, Y) = (X', Y') such that (X', Y') $\sqsubseteq$ (X', Y')
- Ideas?

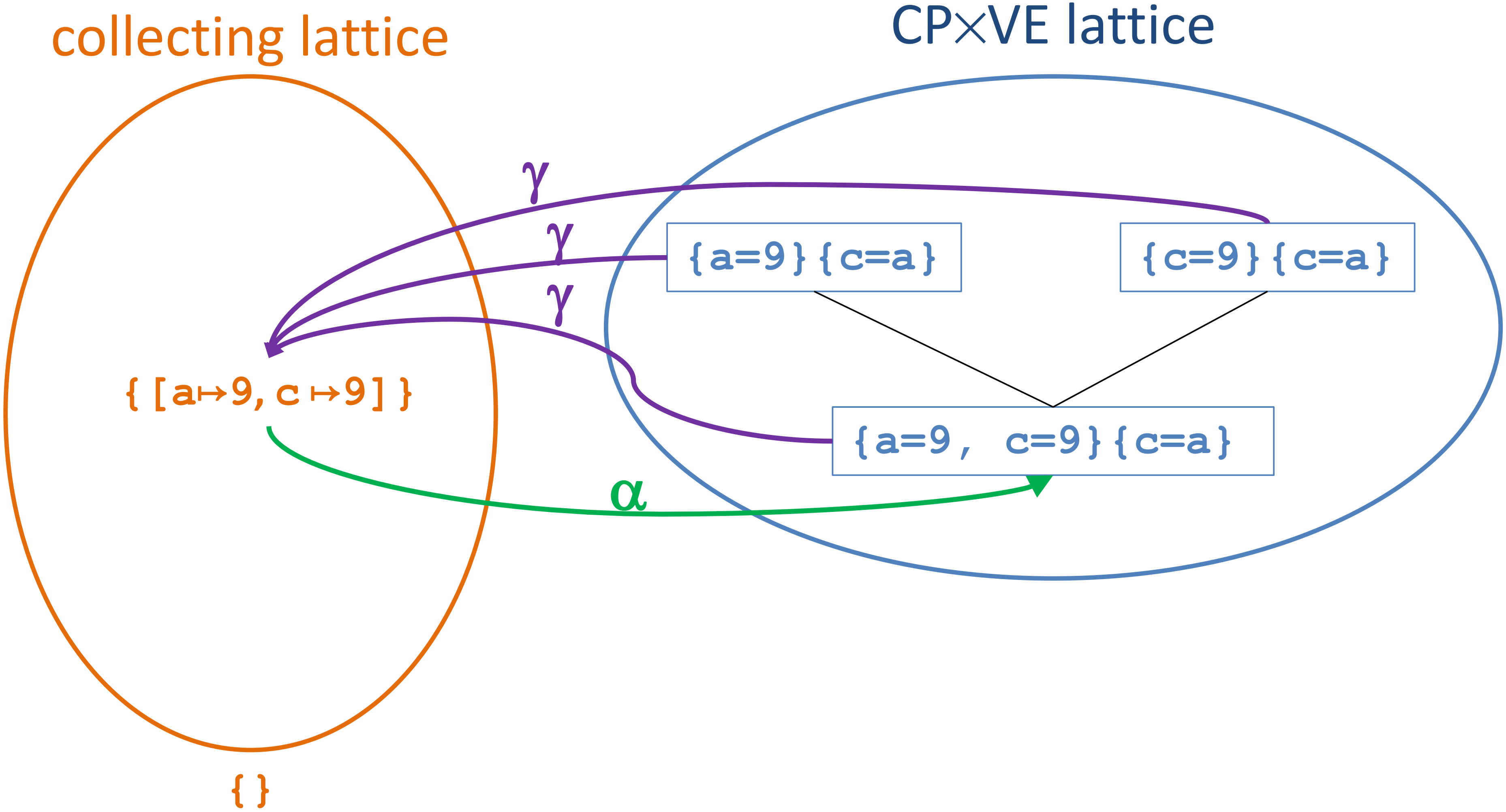
Reducing values for CP×VE

- X = set of CP constraints of the form $x=c$ (e.g., **`a=9`**)
- Y = set of VE constraints of the form x=y
- $\text{Reduce}^{CP\times VE}(X, Y) = (X', Y')$ such that $(X', Y') \sqsubseteq (X', Y')$
- ReduceRight:
  - if a=b $\in X$ and a=c $\in Y$ then add b=c to Y
- ReduceLeft:
  - If a=c and b=c $\in Y$ then add a=b to X
- Keep applying ReduceLeft and ReduceRight and reductions on each domain separately until reaching a fixed-point

Transformers for Cartesian product

- Do we get the best transformer by applying component-wise transformer followed by reduction?
  - Unfortunately, no (what’s the intuition?)
  - Can we do better?
  - Logical Product [Gulwani and Tiwari, PLDI 2006]

Product vs. reduced product



collecting lattice

CP×VE lattice

`{a=9}{c=a}`

`{c=9}{c=a}`

`{a=9, c=9}{c=a}`

`{[a↦9,c↦9]}`

`{}`

γ

γ

γ

α

Reduced product

- For two complete lattices
  $L_1 = (D_1, \sqsubseteq_1, \sqcup_1, \sqcap_1, \bot_1, \top_1)$
  $L_2 = (D_2, \sqsubseteq_2, \sqcup_2, \sqcap_2, \bot_2, \top_2)$
- Define the reduced poset
  $D_1 \sqcap D_2 = \{(d_1,d_2) \in D_1 \times D_2 \mid (d_1,d_2) = \alpha \circ \gamma\ (d_1,d_2)\ \}$
  $L_1 \sqcap L_2 = (D_1 \sqcap D_2, \sqsubseteq_{cart}, \sqcup_{cart}, \sqcap_{cart}, \bot_{cart}, \top_{cart})$

Transformers for Cartesian product

- Do we get the best transformer by applying component-wise transformer followed by reduction?
  - Unfortunately, no (what’s the intuition?)
  - Can we do better?
  - Logical Product [Gulwani and Tiwari, PLDI 2006]

Combining Abstract Interpreters

Sumit Gulwani
Microsoft Research
sumitg@microsoft.com

Ashish Tiwari
SRI International
tiwari@csl.sri.com

Abstract

We present a methodology for automatically combining abstract interpreters over given lattices to construct an abstract interpreter for the combination of those lattices. This lends modularity to the process of design and implementation of abstract interpreters.

We define the notion of logical product of lattices. This kind of combination is more precise than the reduced product combination. We give algorithms to obtain the join operator and the existential quantification operator for the combined lattice from the corresponding operators of the individual lattices. We also give a bound on the number of steps required to reach a fixed point across loops during analysis over the combined lattice in terms of the corresponding bounds for the individual lattices. We prove that our combination methodology yields the most precise abstract interpretation operators over the logical product of lattices when the individual lattices are over theories that are convex, stably infinite, and disjoint.

We also present an interesting application of logical product wherein some lattices can be reduced to combination of other (unrelated) lattices with known abstract interpreters.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Algorithms, Theory, Verification

Keywords Abstract Interpreter, Logical Product, Reduced Product, Nelson-Oppen Combination

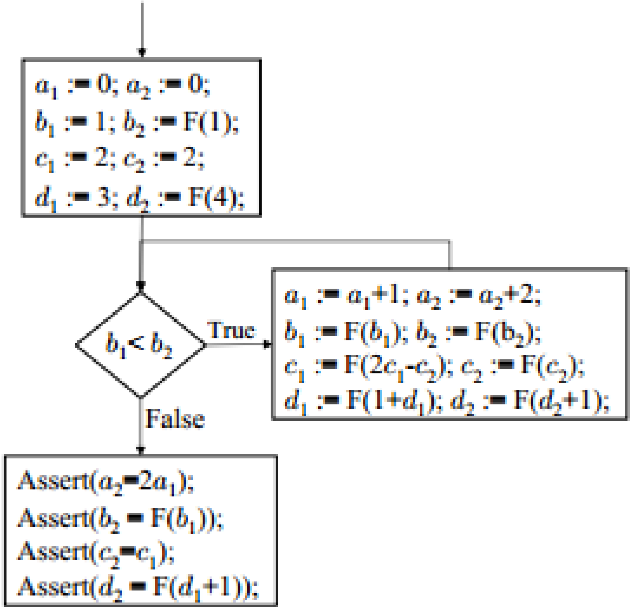


Figure 1. This program illustrates the difference between precision of performing analysis over *direct product*, *reduced product*, and *logical product* of the linear arithmetic lattice and uninterpreted functions lattice. Analysis over direct product can verify the first two assertions, while analysis over reduced product can verify the first three assertions. The analysis over logical product can verify all assertions. *F* denotes some function without any side-effects and can be modeled as an uninterpreted function for purpose of proving the assertions.

Logical product--

- Assume $A=(D,\ldots)$ is an abstract domain that supports two operations: for $x \in D$
  - inferEqualities(x) = $\{ a=b \mid \gamma(x) \vDash a=b \}$
    returns a set of equalities between variables that are satisfied in all states given by x
  - refineFromEqualities(x, $\{a=b\}$) = y
    such that
    - $\gamma(x)=\gamma(y)$
    - $y \sqsubseteq x$

Developing a transformer for *EQ* - 1

- Input has the form $X = \bigwedge\{a=b\}$
- $\mathrm{sp}(x{:=}expr,\ \varphi) = \exists v.\ x=\mathrm{expr}[v/\mathrm{x}] \wedge \varphi[v/\mathrm{x}]$
- $\mathrm{sp}(x{:=}y,\ X) = \exists v.\ x=y[v/\mathrm{x}] \wedge \bigwedge\{a=b\}[v/\mathrm{x}] = \ldots$
- Let's define helper notations:
  - $\mathrm{EQ}(X, y) = \{y=a, b=y \in X\}$
    - Subset of equalities containing *y*
  - $\mathrm{EQc}(X, y) = X \setminus \mathrm{EQ}(X, y)$
    - Subset of equalities **not** containing *y*

Developing a transformer for *EQ* - 2

- sp(x:=y, X) = $\exists v.\ x=y[v/\mathrm{x}] \wedge \bigwedge\{a=b\}[v/\mathrm{x}]$ = …
- Two cases
  - x is y: sp(x:=y, X) = X
  - x is different from y:
    sp(x:=y, X) = $\exists v.\ x=y \wedge \mathrm{EQ}(\mathrm{X}, \mathrm{x})[v/\mathrm{x}] \wedge \mathrm{EQc}(X, x)[v/\mathrm{x}]$
    $= x=y \wedge\ \mathrm{EQc}(X, x)\ \wedge \exists v.\ \mathrm{EQ}(\mathrm{X}, \mathrm{x})[v/\mathrm{x}]$
    $\Rightarrow x=y \wedge \mathrm{EQc}(X, x)$
- Vanilla transformer: $[\![x{:=}y]\!]^{\#1}\ X = x=y \wedge\ \mathrm{EQc}(X, x)$
- Example: $[\![x{:=}y]\!]^{\#1} \bigwedge\{\mathrm{x=p,\ q=x,\ m=n}\} = \bigwedge\{\mathrm{x=y,\ m=n}\}$
  Is this the most precise result?

Developing a transformer for *EQ* - 3

- $[\![x{:=}y]\!]^{\#1} \wedge\{x=p, x=q, m=n\} = \bigwedge\{x=y, m=n\} \sqsupseteq \bigwedge\{x=y, m=n, p=q\}$
  - Where does the information p=q come from?
- $\mathrm{sp}(x{:=}y,\ X) =$

  $$x=y \wedge \mathrm{EQc}(X, x) \wedge \exists v.\ \mathrm{EQ}(X, x)[v/x]$$
- $\exists v.\ \mathrm{EQ}(X, x)[v/x]$ holds possible equalities between different *a*'s and *b*'s – how can we account for that?

Developing a transformer for *EQ* - 4

- Define a reduction operator:

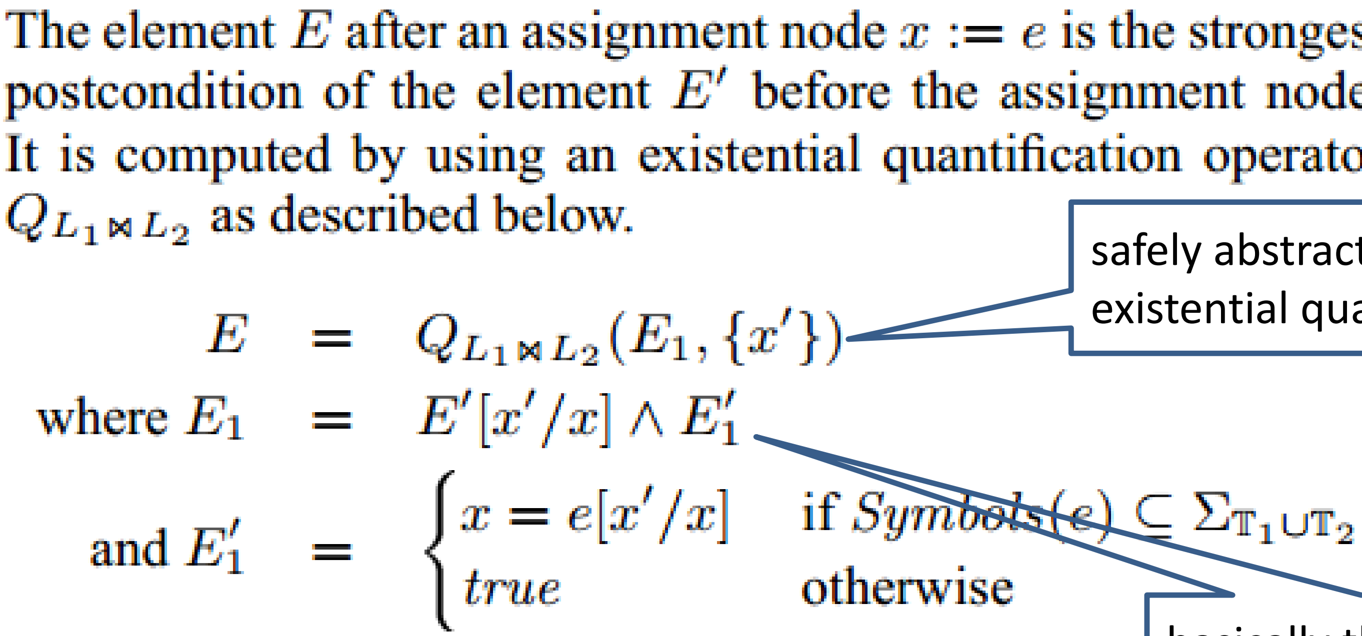
```
Explicate(X) = if exist $\{a=b, b=c\} \subseteq X$
        but not $\{a=c\} \subseteq X$ then
    Explicate($X \cup \{a=c\}$)
        else
            X
```

- Define $[\![x{:=}y]\!]^{\#2} = [\![x{:=}y]\!]^{\#1} \circ \text{Explicate}$
- $[\![x{:=}y]\!]^{\#2} \wedge(\{x=p, x=q, m=n\}) = \bigwedge\{x=y, m=n, p=q\}$ is this the best transformer?

Developing a transformer for *EQ* - 5

- $[\![x{:=}y]\!]^{\#2}\ \bigwedge(\{y=z\}) = \{x=y, y=z\} \sqsupseteq \{x=y, y=z, x=z\}$
- Idea: apply reduction operator again after the vanilla transformer
- $[\![x{:=}y]\!]^{\#3} = \text{Explicate} \circ [\![x{:=}y]\!]^{\#1} \circ \text{Explicate}$

Logical Product-



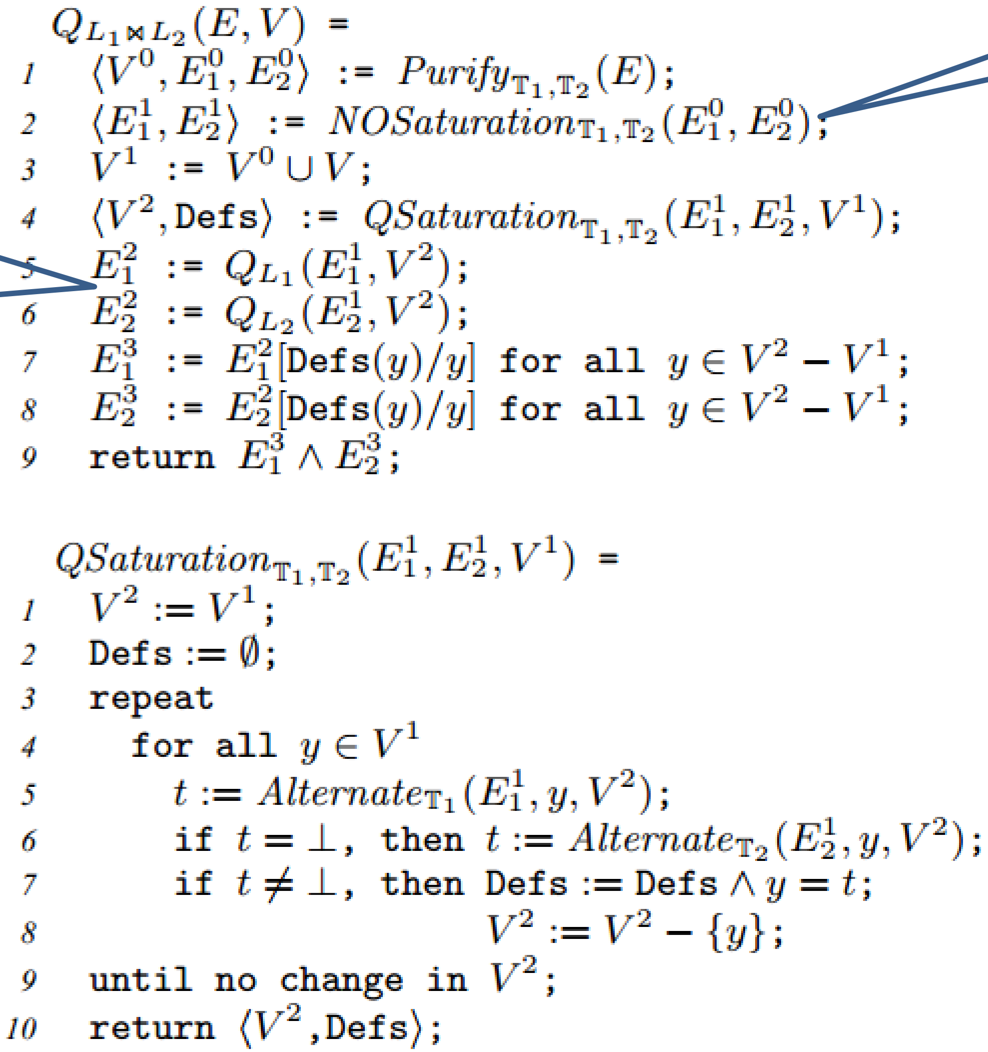
The element E after an assignment node $x := e$ is the strongest postcondition of the element E' before the assignment node. It is computed by using an existential quantification operator $Q_{L_1 \bowtie L_2}$ as described below.

$$\begin{array}{rcl} E & = & Q_{L_1 \bowtie L_2}(E_1, \{x'\}) \\ \text{where } E_1 & = & E'[x'/x] \wedge E_1' \\ \text{and } E_1' & = & \begin{cases} x = e[x'/x] & \text{if } \mathit{Symbols}(e) \subseteq \Sigma_{\mathrm{T}_1 \cup \mathrm{T}_2} \\ \mathit{true} & \text{otherwise} \end{cases} \end{array}$$

safely abstracting the existential quantifier

basically the strongest postcondition

Abstracting the existential



```
$Q_{L_1 \bowtie L_2}(E, V)$ =
  $\langle V^0, E_1^0, E_2^0 \rangle$ := $Purify_{T_1,T_2}(E)$;
  $\langle E_1^1, E_2^1 \rangle$ := $NOSaturation_{T_1,T_2}(E_1^0, E_2^0)$;
  $V^1$ := $V^0 \cup V$;
  $\langle V^2$,Defs$\rangle$ := $QSaturation_{T_1,T_2}(E_1^1, E_2^1, V^1)$;
  $E_1^2$ := $Q_{L_1}(E_1^1, V^2)$;
  $E_2^2$ := $Q_{L_2}(E_2^1, V^2)$;
  $E_1^3$ := $E_1^2[\mathtt{Defs}(y)/y]$ for all $y \in V^2 - V^1$;
  $E_2^3$ := $E_2^2[\mathtt{Defs}(y)/y]$ for all $y \in V^2 - V^1$;
  return $E_1^3 \wedge E_2^3$;
```

Reduce the pair

Abstract away existential quantifier for each domain

```
$QSaturation_{T_1,T_2}(E_1^1, E_2^1, V^1)$ =
  $V^2 := V^1$;
  Defs $:= \emptyset$;
  repeat
    for all $y \in V^1$
      $t := Alternate_{T_1}(E_1^1, y, V^2)$;
      if $t = \bot$, then $t := Alternate_{T_2}(E_2^1, y, V^2)$;
      if $t \neq \bot$, then Defs := Defs $\wedge\ y = t$;
                             $V^2 := V^2 - \{y\}$;
  until no change in $V^2$;
  return $\langle V^2$,Defs$\rangle$;
```

Example

Information loss example

```
if (…)          {}
  b := 5        {b=5}
else
  b := -5       {b=-5}
                {b=⊤}
if (b>0)
  b := b-5      {b=⊤}
else
  b := b+5      {b=⊤}
assert b==0     can’t prove
```

Disjunctive completion of a lattice

- For a complete lattice
  $L = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$
- Define the powerset lattice
  $L_\vee = (2^D, \sqsubseteq_\vee, \sqcup_\vee, \sqcap_\vee, \bot_\vee, \top_\vee)$
  $\sqsubseteq_\vee = ?$ $\sqcup_\vee = ?$ $\sqcap_\vee = ?$ $\bot_\vee = ?$ $\top_\vee = ?$
- **Lemma:** $L_\vee$ is a complete lattice
- $L_\vee$ contains all subsets of *D*, which can be thought of as disjunctions of the corresponding predicates
- Define the disjunctive completion constructor
  $L_\vee = \text{Disj}(L)$

Disjunctive completion for GCs

- $GC^{C,A}=(C, \alpha^{C,A}, \gamma^{A,C}, A)$
  $GC^{C,B}=(C, \alpha^{C,B}, \gamma^{B,C}, B)$
- Disjunctive completion

  $$GC^{C,P(A)} = (C, \alpha^{P(A)}, \gamma^{P(A)}, P(A))$$

  - $\alpha^{C,P(A)}(X) = ?$
  - $\gamma^{P(A),C}(Y) = ?$

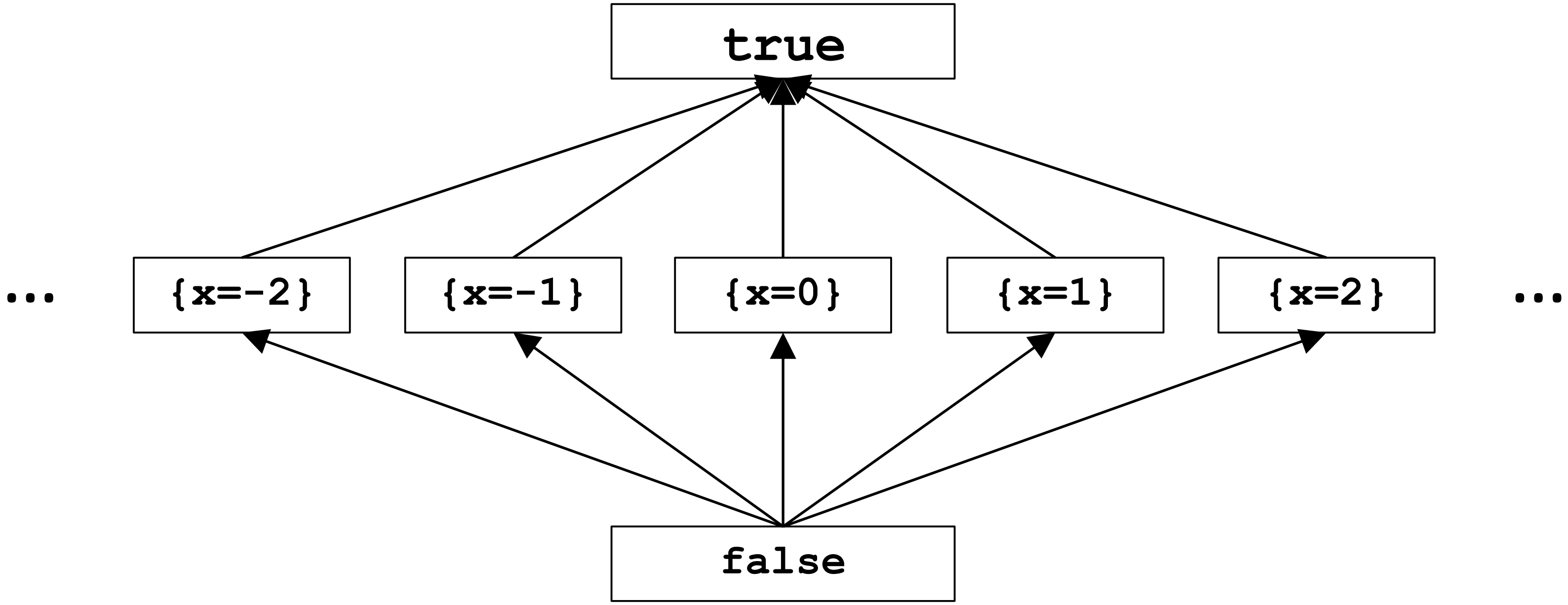
Disjunctive completion for GCs

- $GC^{C,A}=(C, \alpha^{C,A}, \gamma^{A,C}, A)$
  $GC^{C,B}=(C, \alpha^{C,B}, \gamma^{B,C}, B)$
- Disjunctive completion
  $$GC^{C,P(A)} = (C, \alpha^{P(A)}, \gamma^{P(A)}, P(A))$$
  - $\alpha^{C,P(A)}(X) = \{\alpha^{C,A}(\{x\}) \mid x\in X\}$
  - $\gamma^{P(A),C}(Y) = \cup\{\gamma^{P(A)}(y) \mid y\in Y\}$
- What about transformers?

Information loss example

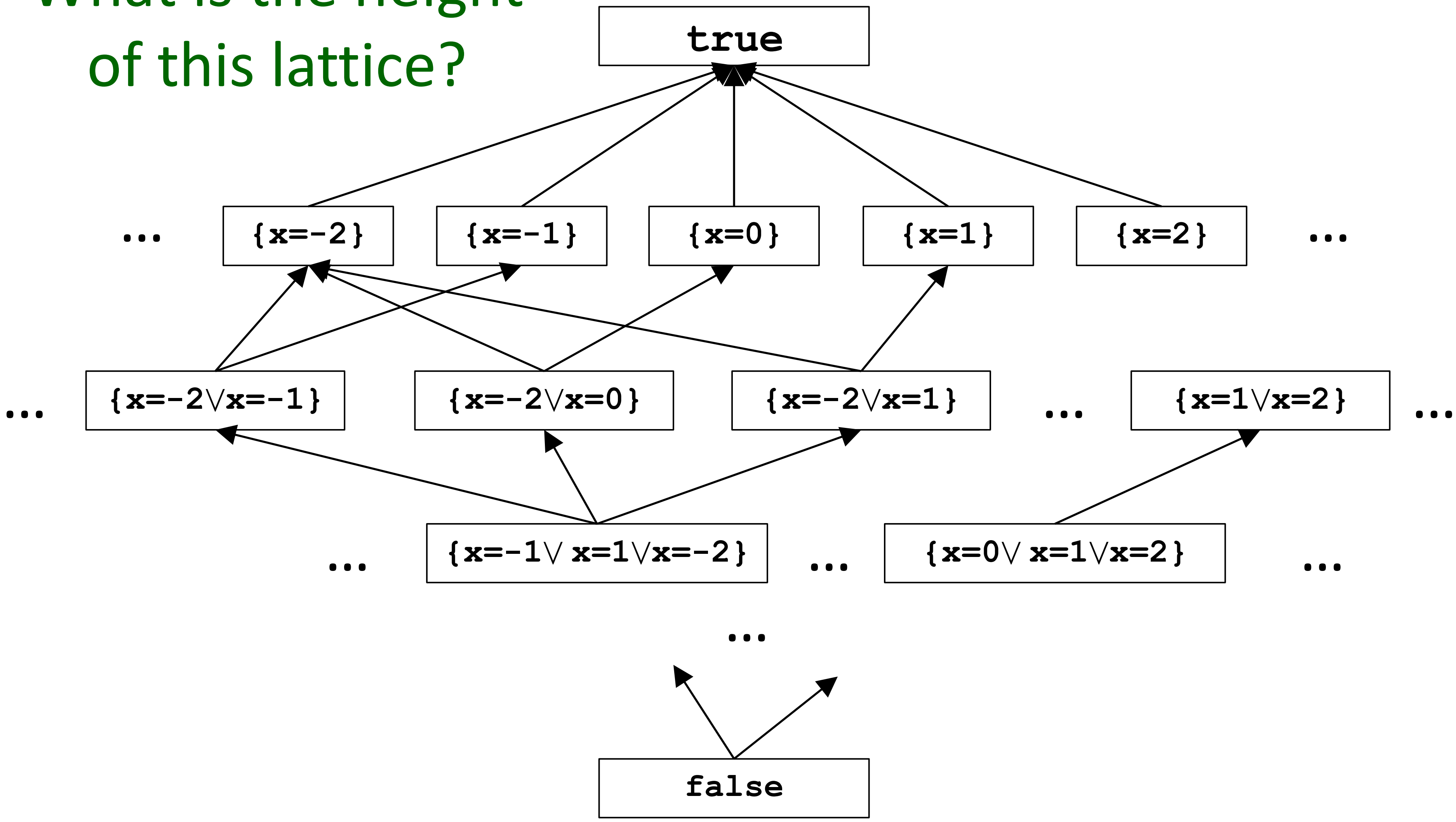
```
if (…)          {}
   b := 5       {b=5}
else
   b := -5      {b=-5}
                {b=5 ∨ b=-5}
if (b>0)
   b := b-5     {b=0}
else
   b := b+5     {b=0}
assert b==0    proved
```

The base lattice CP



The disjunctive completion of CP

What is the height of this lattice?



Taming disjunctive completion

- Disjunctive completion is very precise
  - Maintains correlations between states of different analyses
  - Helps handle conditions precisely
  - But very expensive – number of abstract states grows exponentially
  - May lead to non-termination
- Base analysis (usually product) is less precise
  - Analysis terminates if the analyses of each component terminates
- How can we combine them to get more precision yet ensure termination and state explosion?

Taming disjunctive completion

- Use different abstractions for different program locations
  - At loop heads use coarse abstraction (base)
  - At other points use disjunctive completion
- Termination is guaranteed (by base domain)
- Precision increased **inside** loop body

With Disj(CP)

```
while (…) {
   if (…)
      b := 5
   else
      b := -5

   if (b>0)
      b := b-5
   else
      b := b+5
   assert b==0
}
```

Doesn’t terminate

With tamed Disj(CP)

CP

Disj(CP)

```
while (…) {
   if (…)
     b := 5
   else
     b := -5

   if (b>0)
     b := b-5
   else
     b := b+5
   assert b==0
 }
```

terminates

What `MultiCartDomain` implements

Reducing disjunctive elements

- A disjunctive set *X* may contain within it an ascending chain $Y = a \sqsubseteq b \sqsubseteq c\ldots$
- We only need max(*Y*) – remove all elements below

Relational product of lattices

- $L_1 = (D_1, \sqsubseteq_1, \sqcup_1, \sqcap_1, \bot_1, \top_1)$
  $L_2 = (D_2, \sqsubseteq_2, \sqcup_2, \sqcap_2, \bot_2, \top_2)$
- $L_{rel} = (2^{D_1 \times D_2}, \sqsubseteq_{rel}, \sqcup_{rel}, \sqcap_{rel}, \bot_{rel}, \top_{rel})$
  as follows:
  - L_{rel} = ?

Relational product of lattices

- $L_1 = (D_1, \sqsubseteq_1, \sqcup_1, \sqcap_1, \bot_1, \top_1)$
  $L_2 = (D_2, \sqsubseteq_2, \sqcup_2, \sqcap_2, \bot_2, \top_2)$
- $L_{rel} = (2^{D_1 \times D_2}, \sqsubseteq_{rel}, \sqcup_{rel}, \sqcap_{rel}, \bot_{rel}, \top_{rel})$ as follows:
  - L_{rel} = Disj(Cart(L_1, L_2))
- **Lemma:** *L* is a complete lattice
- What does it buy us?
  - How is it relative to Cart(Disj(L_1), Disj(L_2))?
- What about transformers?

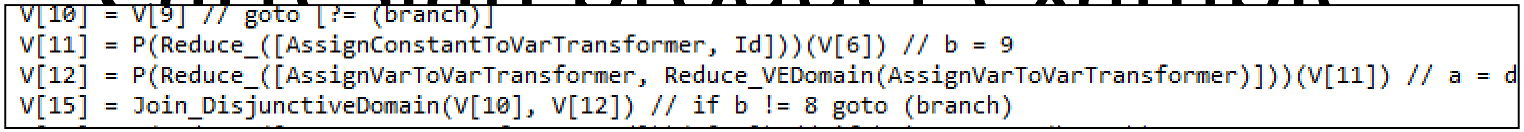
Relational product of GCs

- $GC^{C,A}=(C, \alpha^{C,A}, \gamma^{A,C}, A)$
  $GC^{C,B}=(C, \alpha^{C,B}, \gamma^{B,C}, B)$
- Relational Product
  $GC^{C,P(A\times B)} = (C, \alpha^{C,P(A\times B)}, \gamma^{P(A\times B),C}, P(A\times B))$
  - $\alpha^{C,P(A\times B)}(X) = ?$
  - $\gamma^{P(A\times B),C}(Y) = ?$

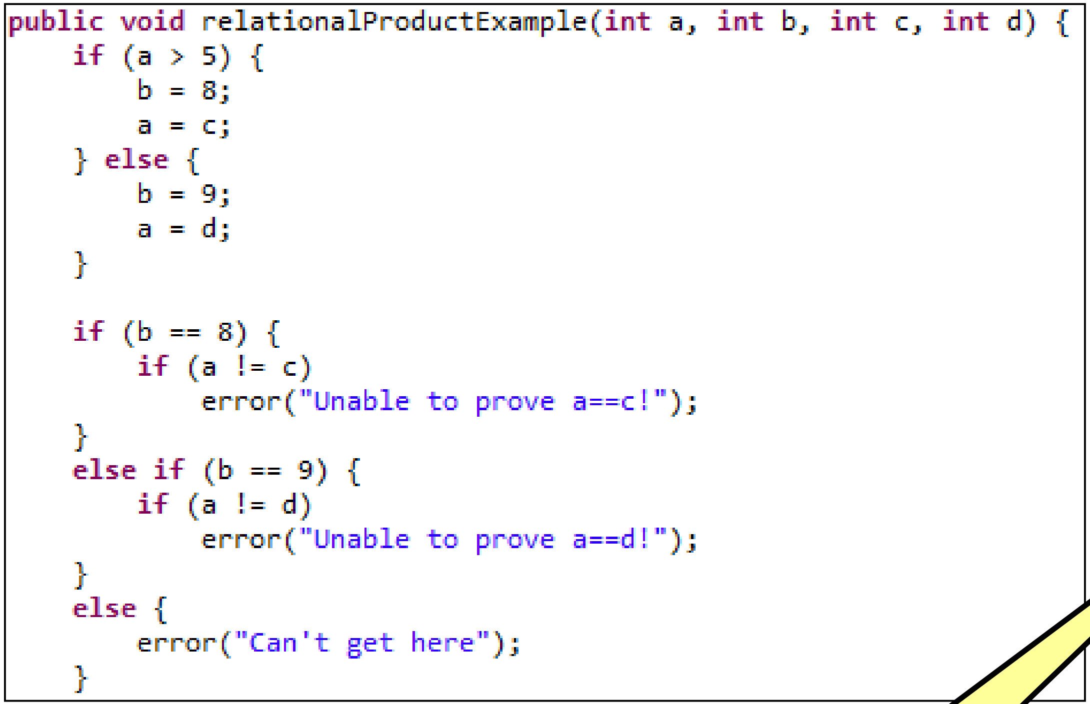
Relational product of GCs

- $GC^{C,A}=(C, \alpha^{C,A}, \gamma^{A,C}, A)$
  $GC^{C,B}=(C, \alpha^{C,B}, \gamma^{B,C}, B)$
- Relational Product
  $GC^{C,P(A\times B)} = (C, \alpha^{C,P(A\times B)}, \gamma^{P(A\times B),C}, P(A\times B))$
  - $\alpha^{C,P(A\times B)}(X) = \{(\alpha^{C,A}(\{x\}), \alpha^{C,B}(\{x\})) \mid x\in X\}$
  - $\gamma^{P(A\times B),C}(Y) = \cup\{\gamma^{A,C}(y_A) \cap \gamma^{B,C}(y_B) \mid (y_A,y_B)\in Y\}$

Cartesian product example

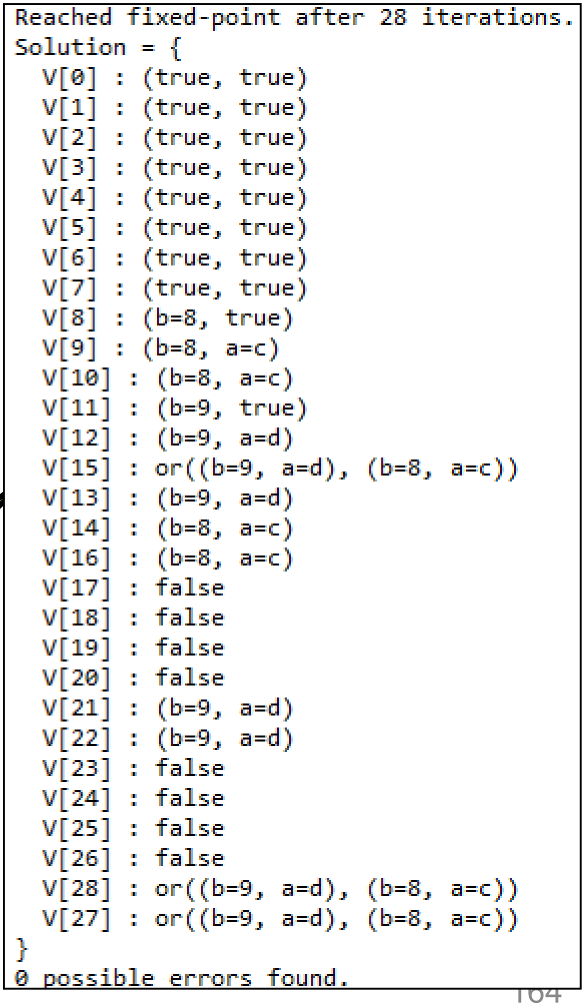


```
V[10] = V[9] // goto [?= (branch)]
V[11] = P(Reduce_([AssignConstantToVarTransformer, Id]))(V[6]) // b = 9
V[12] = P(Reduce_([AssignVarToVarTransformer, Reduce_VEDomain(AssignVarToVarTransformer)]))(V[11]) // a = d
V[15] = Join_DisjunctiveDomain(V[10], V[12]) // if b != 8 goto (branch)
```



```
public void relationalProductExample(int a, int b, int c, int d) {
    if (a > 5) {
        b = 8;
        a = c;
    } else {
        b = 9;
        a = d;
    }

    if (b == 8) {
        if (a != c)
            error("Unable to prove a==c!");
    }
    else if (b == 9) {
        if (a != d)
            error("Unable to prove a==d!");
    }
    else {
        error("Can't get here");
    }
```



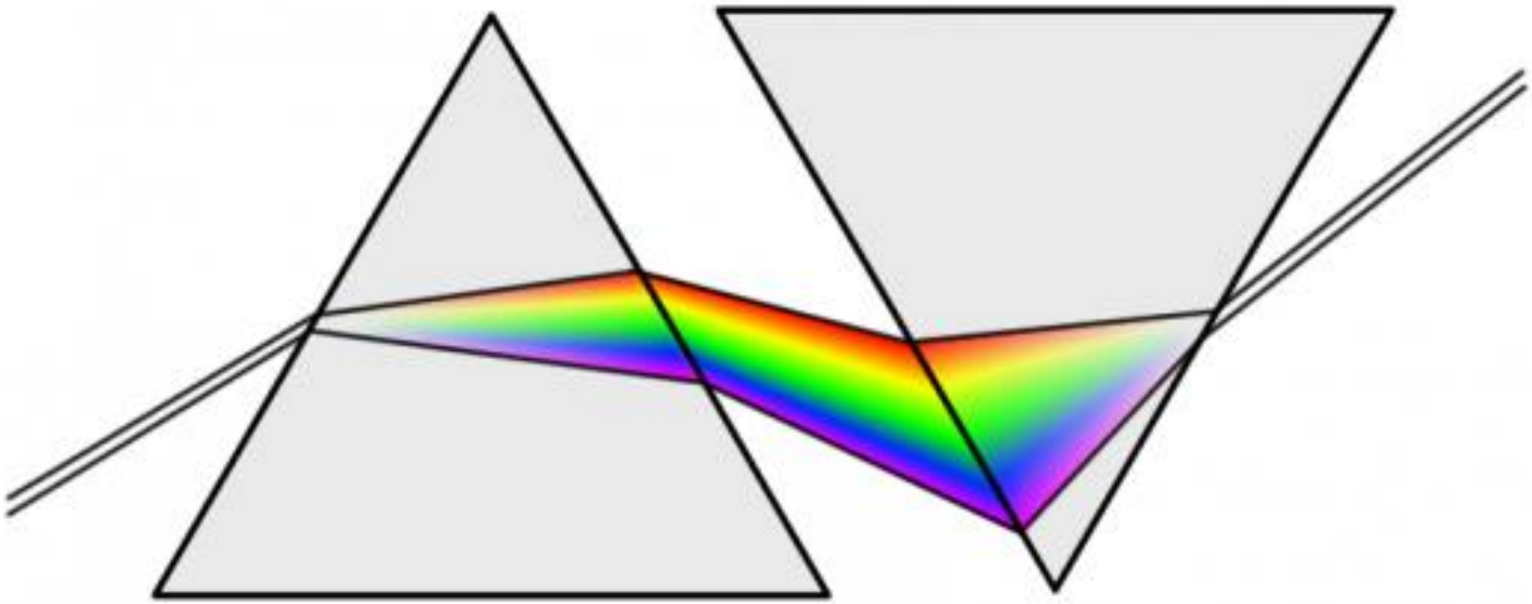
```
Reached fixed-point after 28 iterations.
Solution = {
  V[0] : (true, true)
  V[1] : (true, true)
  V[2] : (true, true)
  V[3] : (true, true)
  V[4] : (true, true)
  V[5] : (true, true)
  V[6] : (true, true)
  V[7] : (true, true)
  V[8] : (b=8, true)
  V[9] : (b=8, a=c)
  V[10] : (b=8, a=c)
  V[11] : (b=9, true)
  V[12] : (b=9, a=d)
  V[15] : or((b=9, a=d), (b=8, a=c))
  V[13] : (b=9, a=d)
  V[14] : (b=8, a=c)
  V[16] : (b=8, a=c)
  V[17] : false
  V[18] : false
  V[19] : false
  V[20] : false
  V[21] : (b=9, a=d)
  V[22] : (b=9, a=d)
  V[23] : false
  V[24] : false
  V[25] : false
  V[26] : false
  V[28] : or((b=9, a=d), (b=8, a=c))
  V[27] : or((b=9, a=d), (b=8, a=c))
}
0 possible errors found.
```

Correlations preserved

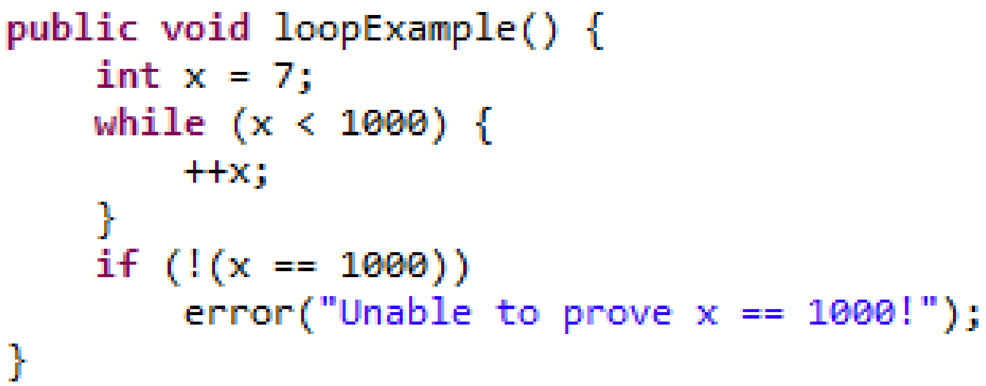
Function space

- $GC^{C,A}=(C, \alpha^{C,A}, \gamma^{A,C}, A)$
  $GC^{C,B}=(C, \alpha^{C,B}, \gamma^{B,C}, B)$
- Denote the set of monotone functions from A to B by $A \longrightarrow B$
- Define $\sqcup$ for elements of $A \longrightarrow B$ as follows
  $(a_1, b_1) \sqcup (a_2, b_2)$ = if $a_1=a_2$ then $\{(a_1, b_1 \sqcup_B b_1)\}$
  else $\{(a_1, b_1), (a_2, b_2)\}$
- Reduced cardinal power
  $GC^{C,A \longrightarrow B} = (C, \alpha^{C,A \longrightarrow B}, \gamma^{A \longrightarrow B,C}, A \longrightarrow B)$
  - $\alpha^{C,A \longrightarrow B}(X) = \sqcup\{(\alpha^{C,A}(\{x\}), \alpha^{C,B}(\{x\})) \mid x \in X\}$
  - $\gamma^{A \longrightarrow B,C}(Y) = \cup\{\gamma^{A,C}(y_A) \cap \gamma^{B,C}(y_B) \mid (y_A,y_B) \in Y\}$
- Useful when A is small and B is much larger
  - E.g., typestate verification

Widening/Narrowing

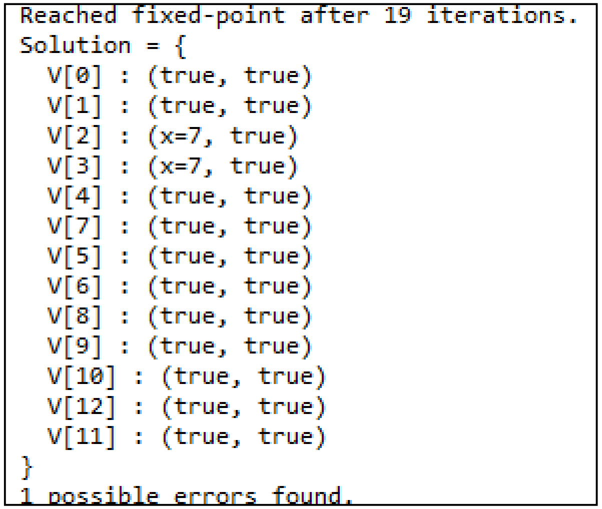


How can we prove this automatically?



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

RelProd(CP, VE)

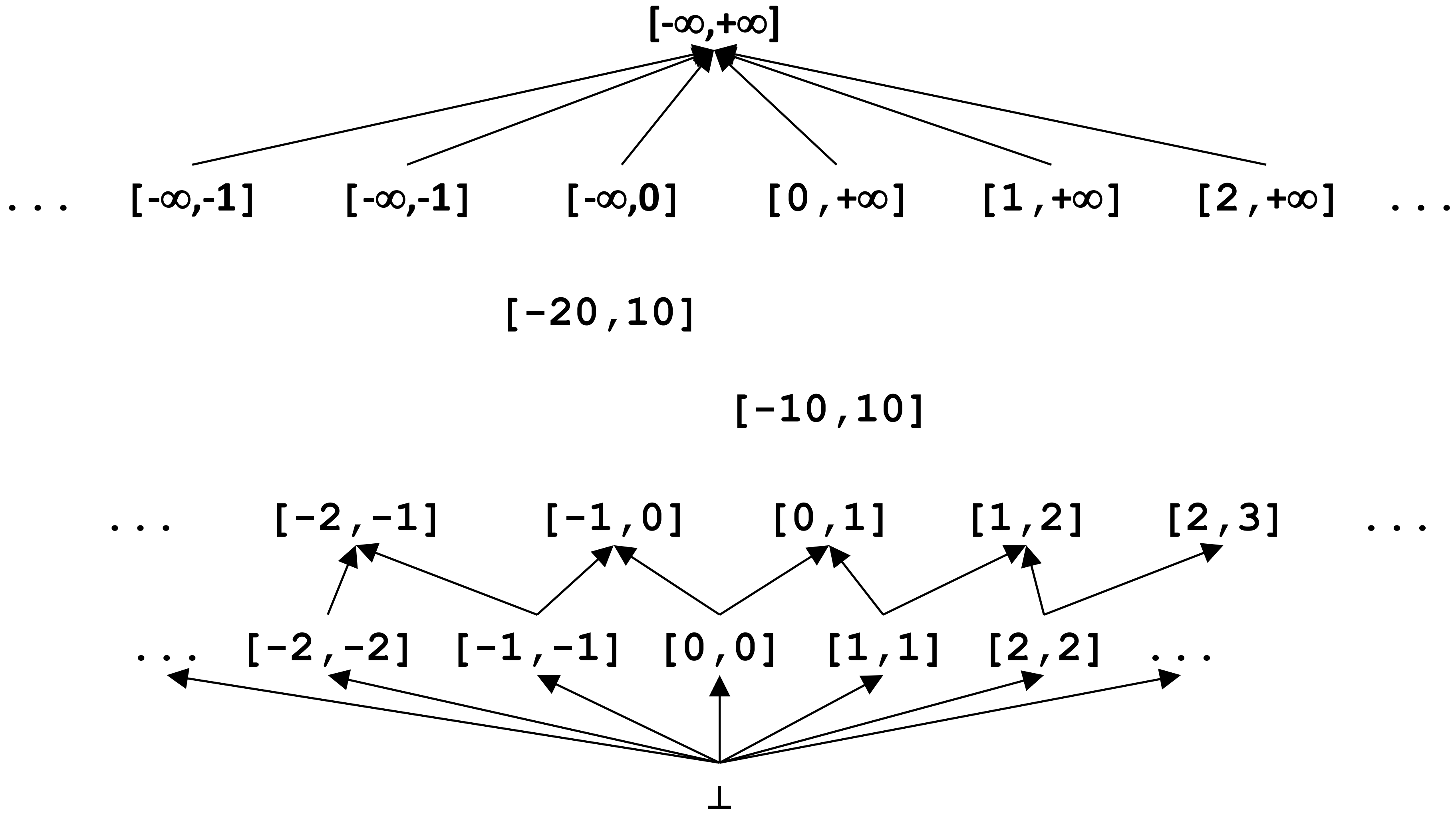


```
Reached fixed-point after 19 iterations.
Solution = {
  V[0] : (true, true)
  V[1] : (true, true)
  V[2] : (x=7, true)
  V[3] : (x=7, true)
  V[4] : (true, true)
  V[7] : (true, true)
  V[5] : (true, true)
  V[6] : (true, true)
  V[8] : (true, true)
  V[9] : (true, true)
  V[10] : (true, true)
  V[12] : (true, true)
  V[11] : (true, true)
}
1 possible errors found.
```

Intervals domain

- One of the simplest numerical domains
- Maintain for each variable *x* an interval [*L*,*H*]
  - *L* is either an integer of -∞
  - *H* is either an integer of +∞
- A (non-relational) numeric domain

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*

- $D^{int}[x] = \{ (L,H) \mid L \in -\infty,\mathbf{Z}$ and $H \in \mathbf{Z},+\infty$ and $L \leq H\}$
- $\bot$
- $\top = [-\infty,+\infty]$
- $\sqsubseteq$ = ?
  - $[1,2] \sqsubseteq [3,4]$?
  - $[1,4] \sqsubseteq [1,3]$?
  - $[1,3] \sqsubseteq [1,4]$?
  - $[1,3] \sqsubseteq [-\infty,+\infty]$?
- What is the lattice height?

Intervals lattice for variable *x*

- $D^{int}[x] = \{\, (L,H) \mid L \in -\infty,\mathbf{Z} \text{ and } H \in \mathbf{Z},+\infty \text{ and } L \leq H \}$
- $\bot$
- $\top = [-\infty,+\infty]$
- $\sqsubseteq$ = ?
  - $[1,2] \sqsubseteq [3,4]$ no
  - $[1,4] \sqsubseteq [1,3]$ no
  - $[1,3] \sqsubseteq [1,4]$ yes
  - $[1,3] \sqsubseteq [-\infty,+\infty]$ yes
- What is the lattice height? Infinite

Joining/meeting intervals

- $[a,b] \sqcup [c,d] = ?$
  - $[1,1] \sqcup [2,2] = ?$
  - $[1,1] \sqcup [2, +\infty] = ?$
- $[a,b] \sqcap [c,d] = ?$
  - $[1,2] \sqcap [3,4] = ?$
  - $[1,4] \sqcap [3,4] = ?$
  - $[1,1] \sqcap [1,+\infty] = ?$
- Check that indeed $x \sqsubseteq y$ if and only if $x \sqcup y = y$

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

- $D^{int}[x] = \{ (L,H) \mid L \in -\infty,\mathbf{Z}$ and $H \in \mathbf{Z},+\infty$ and $L \leq H \}$
- For a program with variables $Var=\{x_1,\ldots,x_k\}$
- $D^{int}[Var]$ = ?

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?

Interval domain for programs

- $D^{int}[x] = \{ (L,H) \mid L \in -\infty,\mathbf{Z}$ 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: $S = \wedge\{x \geq 9, y \geq 5, y \leq 10\}$
  - Helper operations
    - $c + +\infty = +\infty$
    - remove(S, x) = S without any x-constraints
    - lb(S, x) =

Assignment transformers

- $[\![x := c]\!]^\# \, S = ?$
- $[\![x := y]\!]^\# \, S = ?$
- $[\![x := y+c]\!]^\# \, S = ?$
- $[\![x := y+z]\!]^\# \, S = ?$
- $[\![x := y*c]\!]^\# \, S = ?$
- $[\![x := y*z]\!]^\# \, S = ?$

Assignment transformers

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

`assume` transformers

- $[\![\texttt{assume}\ x=c]\!]^\# S$ = ?
- $[\![\texttt{assume}\ x<c]\!]^\# S$ = ?
- $[\![\texttt{assume}\ x=y]\!]^\# S$ = ?
- $[\![\texttt{assume}\ x\neq c]\!]^\# S$ = ?

`assume` transformers

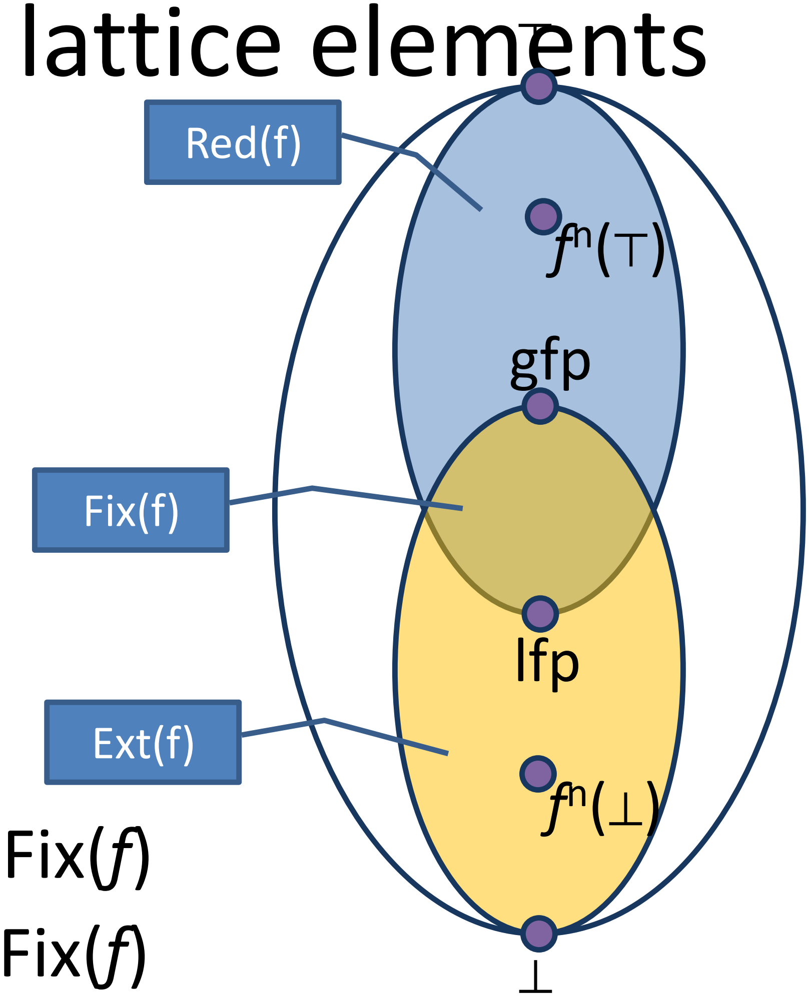
- $[\![\texttt{assume}\ x=c]\!]^\#\ S = S \sqcap \{x \geq c, x \leq c\}$
- $[\![\texttt{assume}\ x<c]\!]^\#\ S = S \sqcap \{x \leq c\text{-}1\}$
- $[\![\texttt{assume}\ x=y]\!]^\#\ S = S \sqcap \{x \geq \mathrm{lb}(S,y), x \leq \mathrm{ub}(S,y)\}$
- $[\![\texttt{assume}\ x \neq c]\!]^\#\ S = ?$

`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`** $x\neq c]\!]^\# S = (S \sqcap \{x\leq c\text{-}1\}) \sqcup (S \sqcap \{x\geq c+1\})$

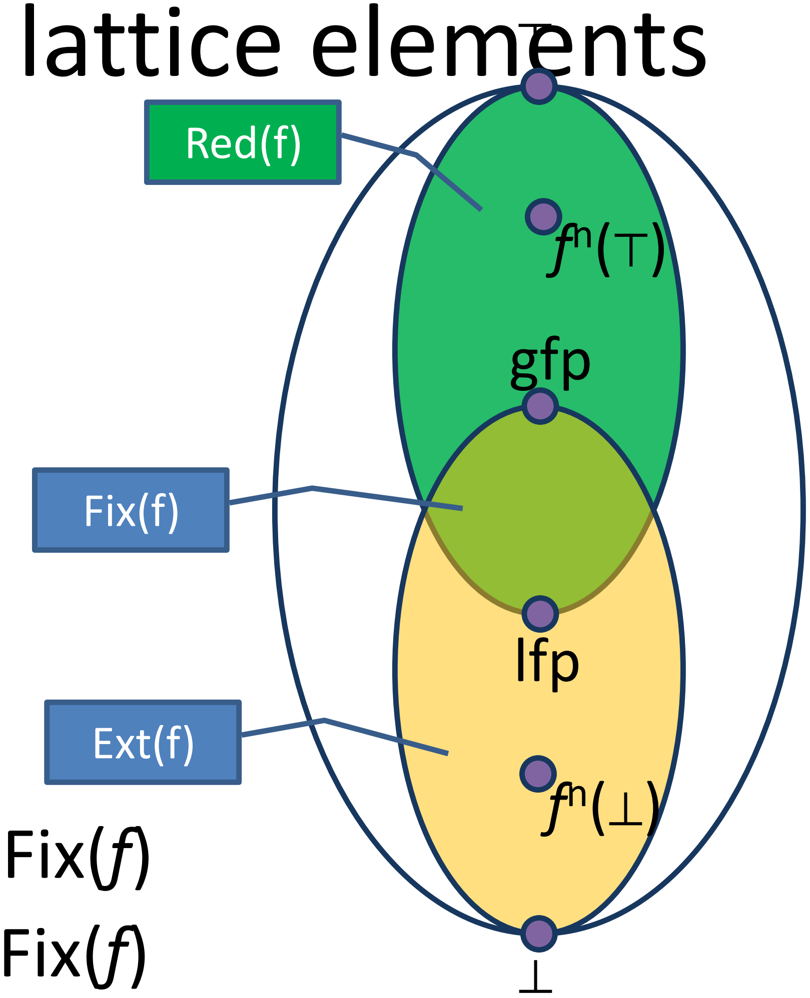
Effect of function *f* on lattice elements

- $L = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$
- $f : D \to D$ **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]
  - $\mathrm{lfp}(f) = \sqcap \mathrm{Fix}(f) = \sqcap \mathrm{Red}(f) \in \mathrm{Fix}(f)$
  - $\mathrm{gfp}(f) = \sqcup \mathrm{Fix}(f) = \sqcup \mathrm{Ext}(f) \in \mathrm{Fix}(f)$



Effect of function *f* on lattice elements

- $L = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$
- $f : D \to D$ **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]
  - $\mathrm{lfp}(f) = \sqcap\mathrm{Fix}(f) = \sqcap\mathrm{Red}(f) \in \mathrm{Fix}(f)$
  - $\mathrm{gfp}(f) = \sqcup\mathrm{Fix}(f) = \sqcup\mathrm{Ext}(f) \in \mathrm{Fix}(f)$



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:

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

Fixed-point theorem [Kleene]

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

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

Resulting algorithm

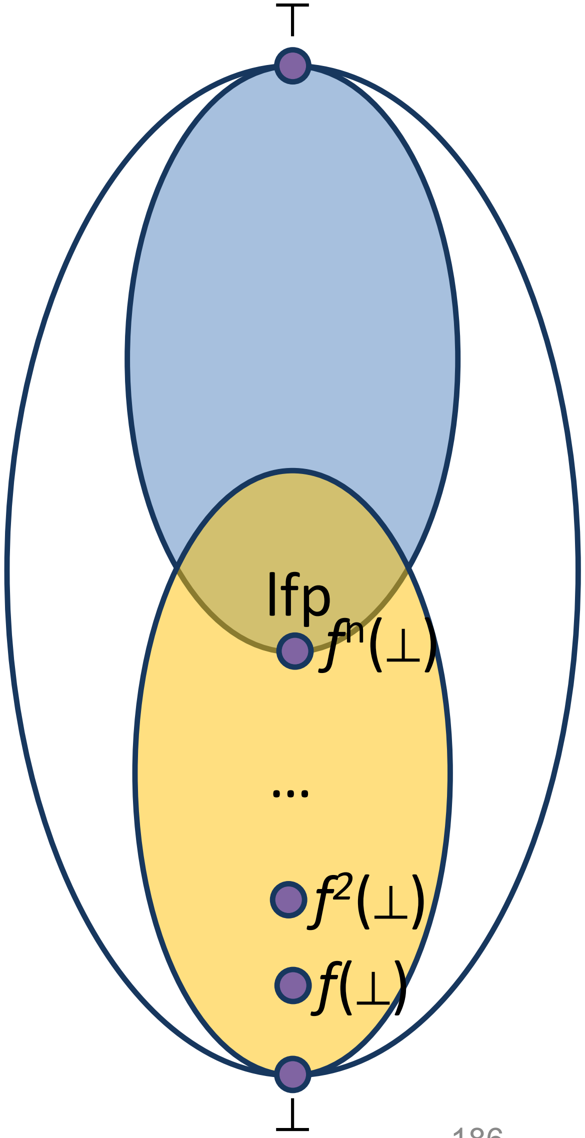
- Kleene's fixed point theorem gives a constructive method for computing the lfp

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** $d$
```

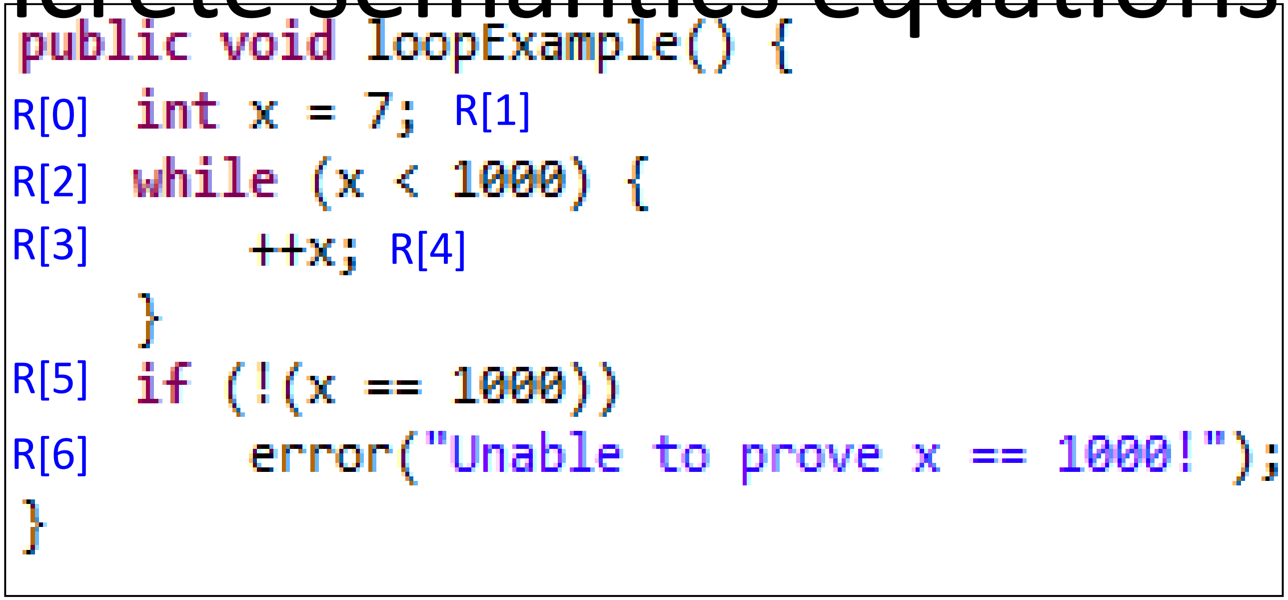


Chaotic iteration

- Input:
  - A cpo $L = (D, \sqsubseteq, \sqcup, \perp)$ satisfying ACC
  - $L^n = L \times L \times \ldots \times L$
  - A monotone function $f : D^n \rightarrow D^n$
  - A system of equations $\{ X[\mathrm{i}] \mid f(X) \mid 1 \leq i \leq n \}$
- Output: lfp(f)
- A worklist-based algorithm

```
for i:=1 to n do
  X[i] := $\perp$
WL = {1,…,n}
while WL $\neq \varnothing$ do
  j := pop WL // choose index non-deterministically
  N := F[i](X)
  if N $\neq$ X[i] then
    X[i] := N
    add all the indexes that directly depend on i to WL
    (X[j] depends on X[i] if F[j] contains X[i])
return X
```

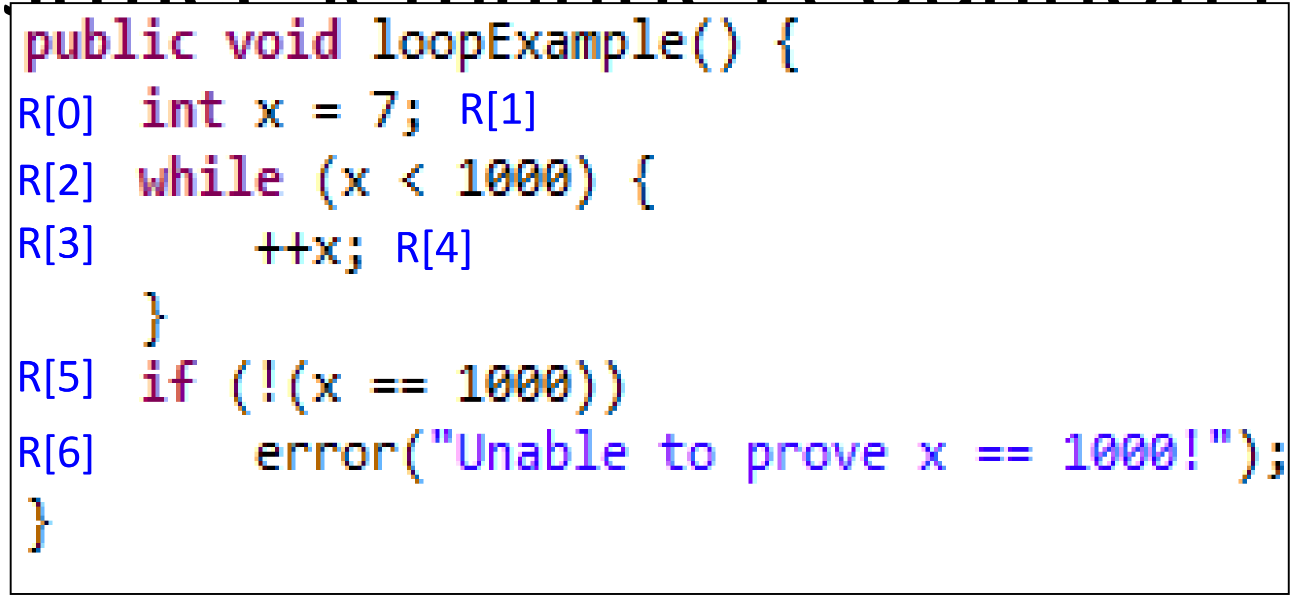
Concrete semantics equations



```
public void loopExample() {
R[0]  int x = 7;  R[1]
R[2]  while (x < 1000) {
R[3]      ++x;  R[4]
      }
R[5]  if (!(x == 1000))
R[6]      error("Unable to prove x == 1000!");
}
```

- $R[0] = \{\mathbf{x} \in \mathbf{Z}\}$
  $R[1] = [\![\mathbf{x:=7}]\!]$
  $R[2] = R[1] \cup R[4]$
  $R[3] = R[2] \cap \{s \mid s(x) < 1000\}$
  $R[4] = [\![\mathbf{x:=x+1}]\!]\ R[3]$
  $R[5] = R[2] \cap \{s \mid s(x) \geq 1000\}$
  $R[6] = R[5] \cap \{s \mid s(x) \neq 1001\}$

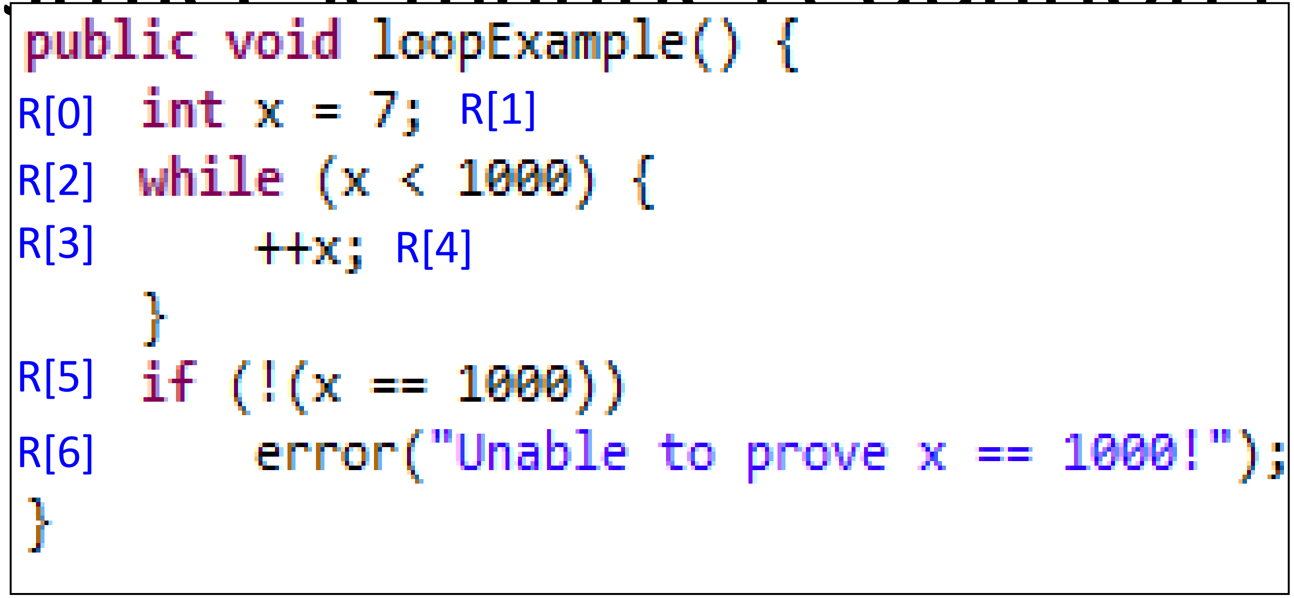
Abstract semantics equations



```
public void loopExample() {
R[0]  int x = 7; R[1]
R[2]  while (x < 1000) {
R[3]      ++x; R[4]
      }
R[5]  if (!(x == 1000))
R[6]      error("Unable to prove x == 1000!");
}
```

- $R[0] = \alpha(\{\mathbf{x} \in \mathbf{Z}\})$
  $R[1] = \llbracket \mathbf{x := 7} \rrbracket^{\#}$
  $R[2] = R[1] \sqcup R[4]$
  $R[3] = R[2] \sqcap \alpha(\{s \mid s(x) < 1000\})$
  $R[4] = \llbracket \mathbf{x := x+1} \rrbracket^{\#}\ R[3]$
  $R[5] = R[2] \sqcap \alpha(\{s \mid s(x) \geq 1000\})$
  $R[6] = R[5] \sqcap \alpha(\{s \mid s(x) \geq 1001\}) \sqcup R[5] \sqcap \alpha(\{s \mid s(x) \leq 999\})$

Abstract semantics equations



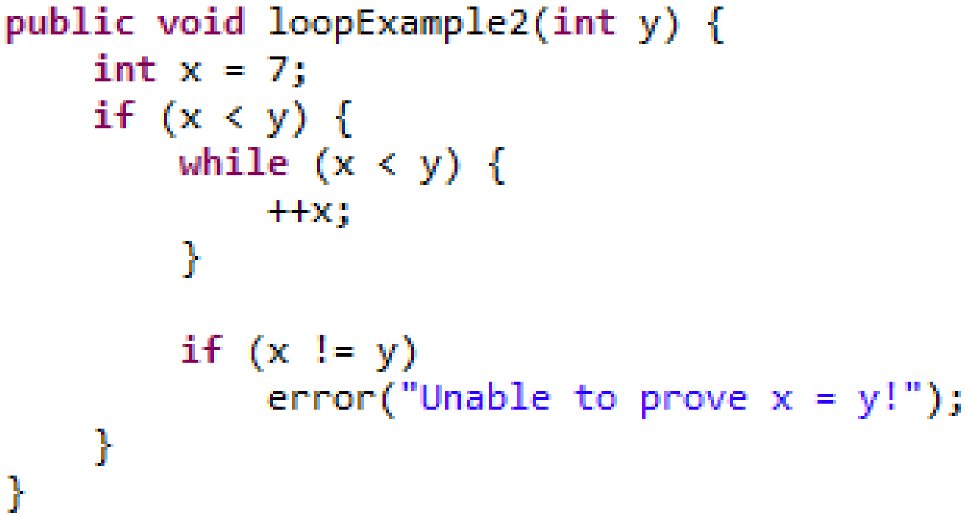
```
public void loopExample() {
R[0]  int x = 7; R[1]
R[2]  while (x < 1000) {
R[3]        ++x; R[4]
      }
R[5]  if (!(x == 1000))
R[6]        error("Unable to prove x == 1000!");
}
```

- $R[0] = \top$
  $R[1] = [7,7]$
  $R[2] = R[1] \sqcup R[4]$
  $R[3] = R[2] \sqcap [-\infty,999]$
  $R[4] = R[3] + [1,1]$
  $R[5] = R[2] \sqcap [1000,+\infty]$
  $R[6] = R[5] \sqcap [999,+\infty] \sqcup R[5] \sqcap [1001,+\infty]$

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.
```

How many iterations for this one?



```
public void loopExample2(int y) {
    int x = 7;
    if (x < y) {
        while (x < y) {
            ++x;
        }

        if (x != y)
            error("Unable to prove x = y!");
    }
}
```

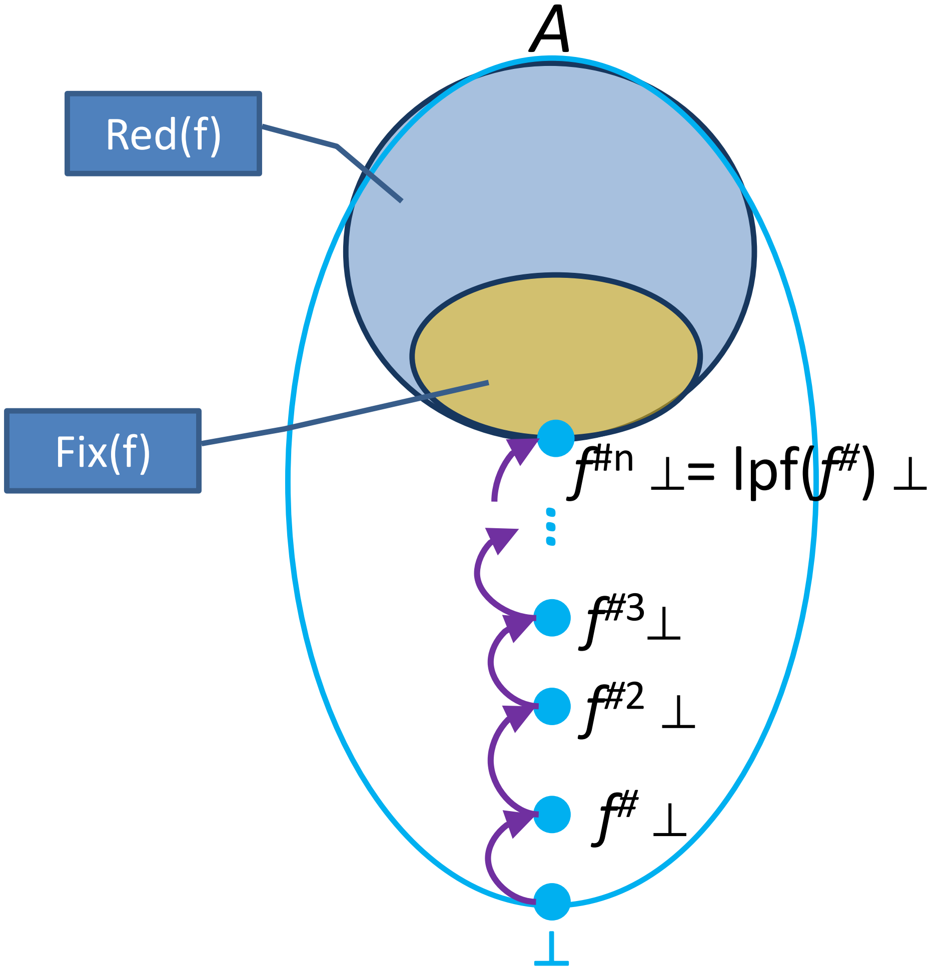
Widening

- Introduce a new binary operator to ensure termination
  - A kind of extrapolation
- Enables static analysis to use infinite height lattices
  - Dynamically adapts to given program
- Tricky to design
- Precision less predictable then with finite-height domains (widening non-monotone)

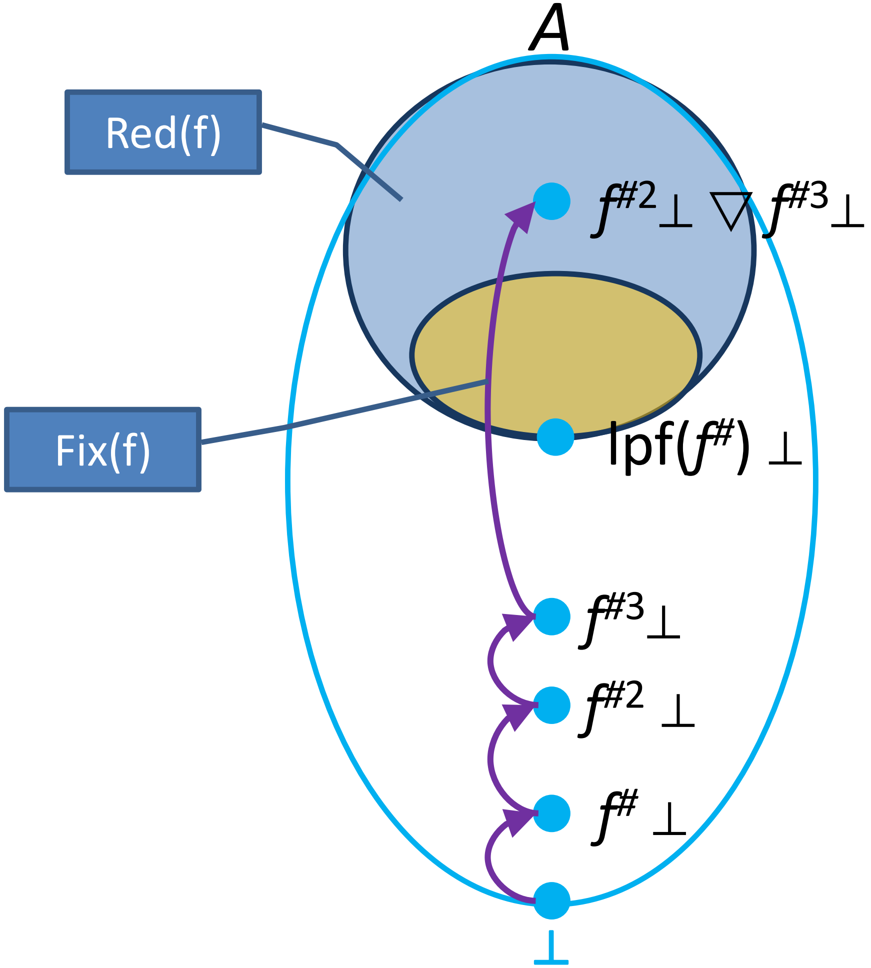
Formal definition

- For all elements $d_1 \sqcup d_2 \sqsubseteq d_1 \triangledown d_2$
- For all ascending chains $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots$ the following sequence is finite
  - $y_0 = d_0$
  - $y_{i+1} = y_i \triangledown d_{i+1}$
- For a monotone function $f : D \rightarrow D$ define
  - $x_0 = \perp$
  - $x_{i+1} = x_i \triangledown f(x_i)$
- Theorem:
  - There exits k such that $x_{k+1} = x_k$
  - $x_k \in \text{Red}(f) = \{ d \mid d \in D \text{ and } f(d) \sqsubseteq d \}$

Analysis with finite-height lattice



Analysis with widening

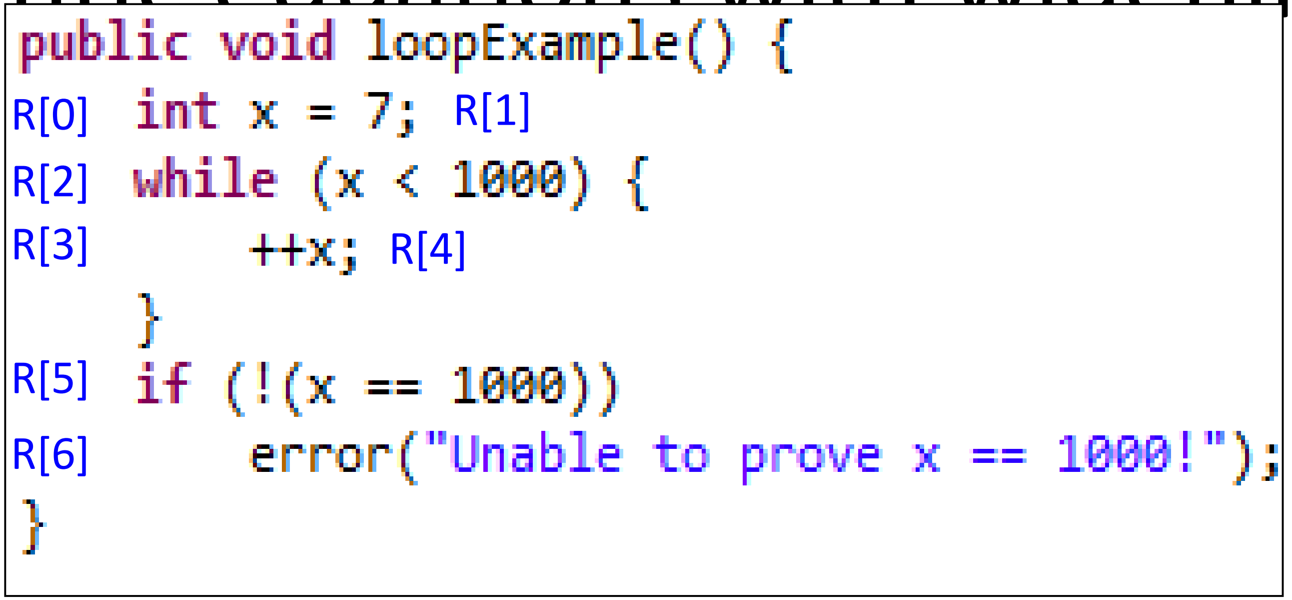


Widening for Intervals Analysis

- $\bot \nabla [c, d] = [c, d]$
- $[a, b] \ \nabla [c, d] =$ [

```
        if $a \leq c$
        then a
        else $-\infty$,
  if $b \geq d$
        then b
        else $\infty$
```

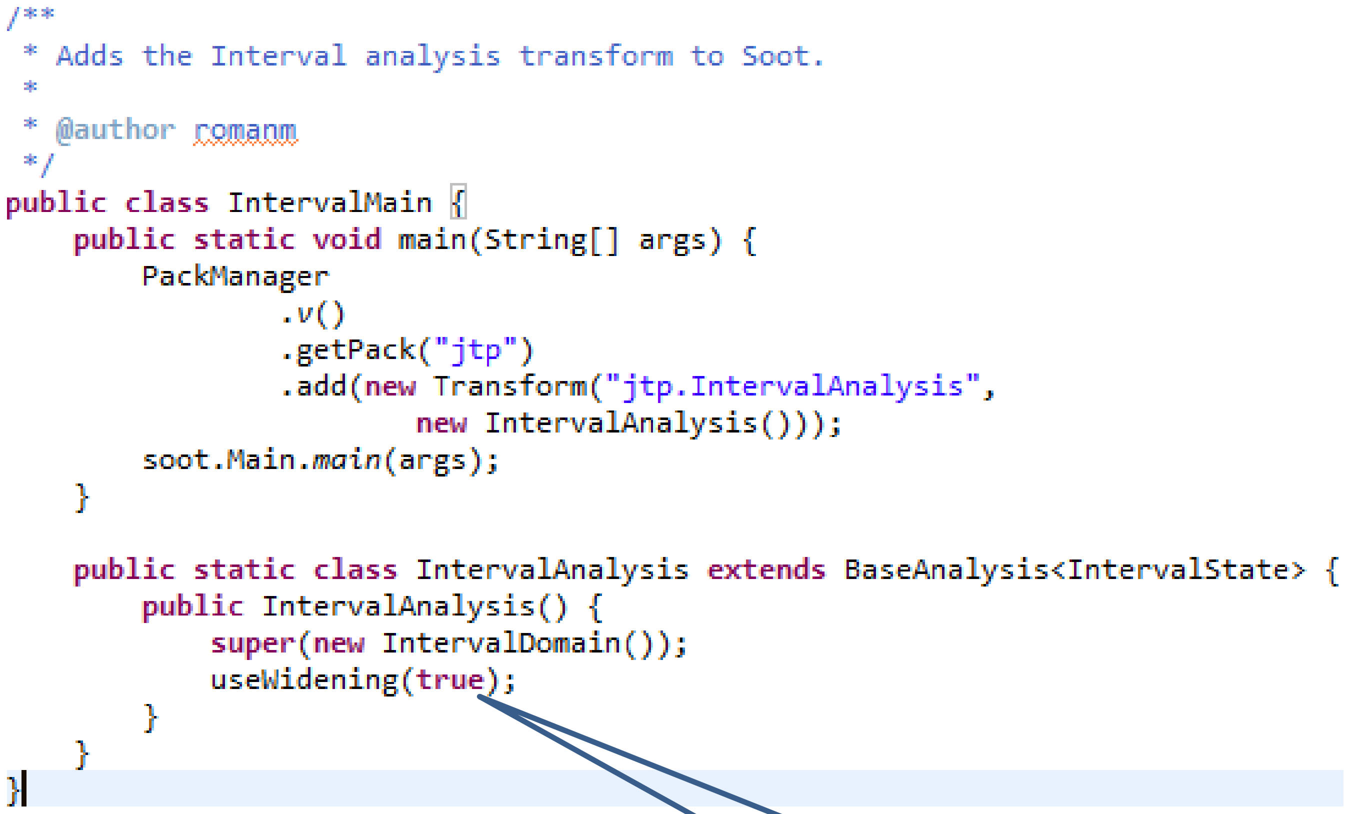
Semantic equations with widening



```
public void loopExample() {
R[0]  int x = 7;  R[1]
R[2]  while (x < 1000) {
R[3]        ++x;  R[4]
      }
R[5]  if (!(x == 1000))
R[6]        error("Unable to prove x == 1000!");
}
```

- $R[0] = \top$
  $R[1] = [7,7]$
  $R[2] = R[1] \sqcup R[4]$
  $R[2.1] = R[2.1] \triangledown R[2]$
  $R[3] = R[2.1] \sqcap [-\infty,999]$
  $R[4] = R[3] + [1,1]$
  $R[5] = R[2] \sqcap [1001,+\infty]$
  $R[6] = R[5] \sqcap [999,+\infty] \sqcup R[5] \sqcap [1001,+\infty]$

Choosing analysis with widening



```
/**
 * Adds the Interval analysis transform to Soot.
 *
 * @author romanm
 */
public class IntervalMain {
    public static void main(String[] args) {
        PackManager
                .v()
                .getPack("jtp")
                .add(new Transform("jtp.IntervalAnalysis",
                        new IntervalAnalysis()));
        soot.Main.main(args);
    }

    public static class IntervalAnalysis extends BaseAnalysis<IntervalState> {
        public IntervalAnalysis() {
            super(new IntervalDomain());
            useWidening(true);
        }
    }
}
```

Enable widening

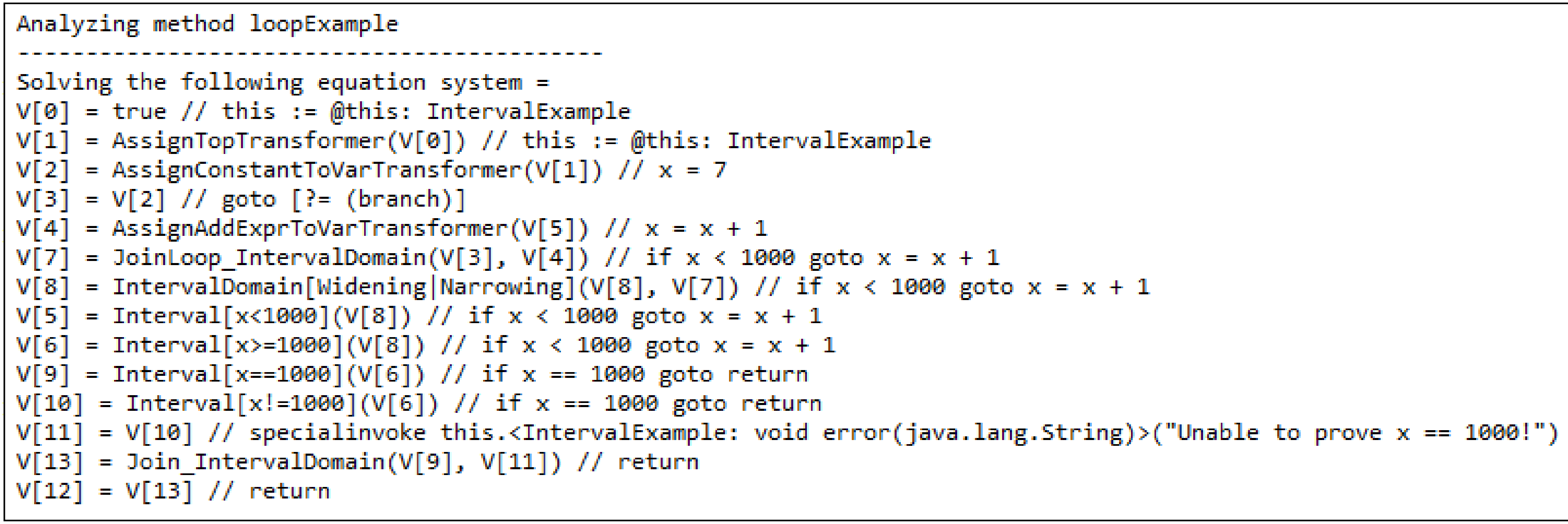
Non monotonicity of widening

- $[0,1] \triangledown [0,2]$ = ?
- $[0,2] \triangledown [0,2]$ = ?

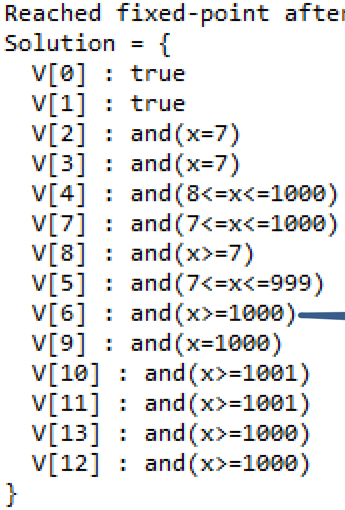
Non monotonicity of widening

- $[0,1] \triangledown [0,2] = [0, \infty]$
- $[0,2] \triangledown [0,2] = [0,2]$

Analysis results with widening



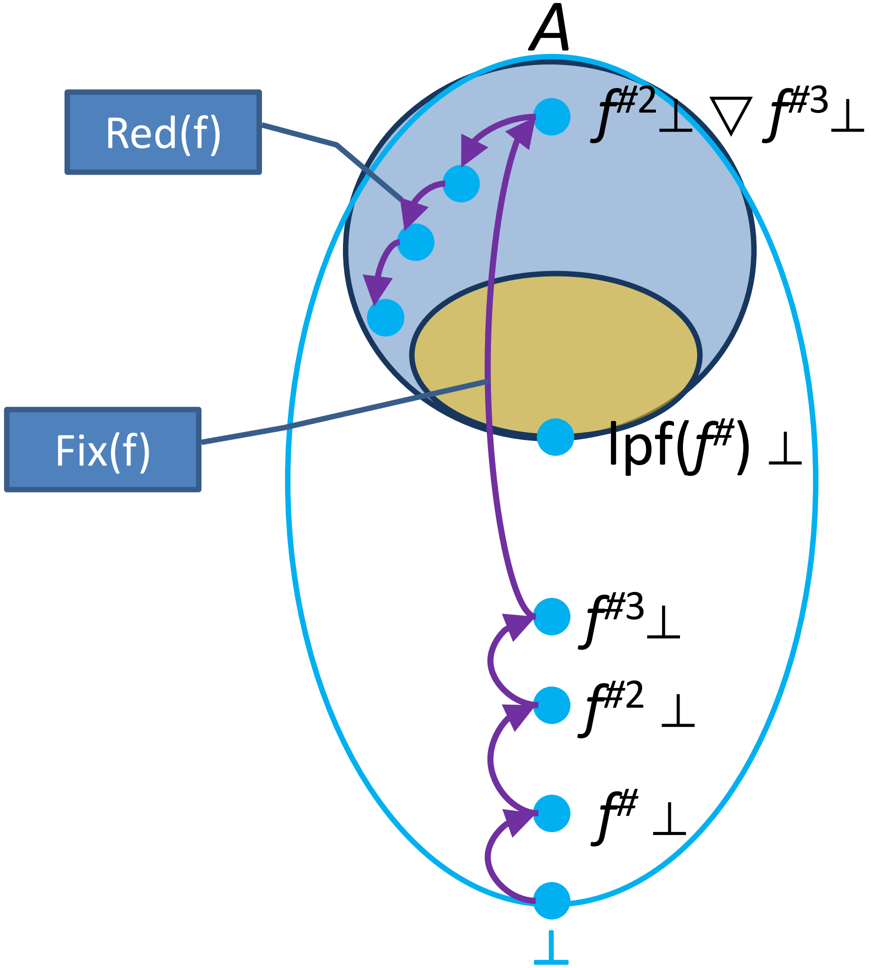
```
Analyzing method loopExample
-------------------------------------------
Solving the following equation system =
V[0] = true // this := @this: IntervalExample
V[1] = AssignTopTransformer(V[0]) // this := @this: IntervalExample
V[2] = AssignConstantToVarTransformer(V[1]) // x = 7
V[3] = V[2] // goto [?= (branch)]
V[4] = AssignAddExprToVarTransformer(V[5]) // x = x + 1
V[7] = JoinLoop_IntervalDomain(V[3], V[4]) // if x < 1000 goto x = x + 1
V[8] = IntervalDomain[Widening|Narrowing](V[8], V[7]) // if x < 1000 goto x = x + 1
V[5] = Interval[x<1000](V[8]) // if x < 1000 goto x = x + 1
V[6] = Interval[x>=1000](V[8]) // if x < 1000 goto x = x + 1
V[9] = Interval[x==1000](V[6]) // if x == 1000 goto return
V[10] = Interval[x!=1000](V[6]) // if x == 1000 goto return
V[11] = V[10] // specialinvoke this.<IntervalExample: void error(java.lang.String)>("Unable to prove x == 1000!")
V[13] = Join_IntervalDomain(V[9], V[11]) // return
V[12] = V[13] // return
```



```
Reached fixed-point after 23 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[8] : and(x>=7)
  V[5] : and(7<=x<=999)
  V[6] : and(x>=1000)
  V[9] : and(x=1000)
  V[10] : and(x>=1001)
  V[11] : and(x>=1001)
  V[13] : and(x>=1000)
  V[12] : and(x>=1000)
}
```

Did we prove it?

Analysis with narrowing



Formal definition of narrowing

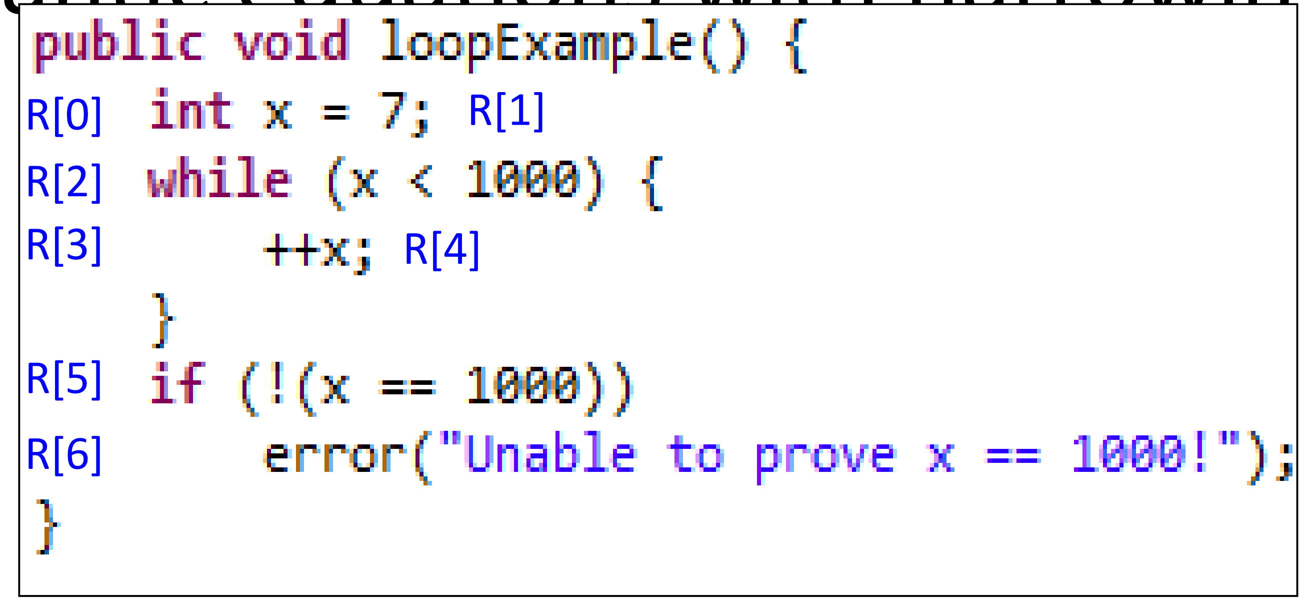
- Improves the result of widening
- $y \sqsubseteq x \Rightarrow y \sqsubseteq (x \triangle y) \sqsubseteq x$
- For all decreasing chains $x_0 \sqsupseteq x_1 \sqsupseteq \dots$ the following sequence is finite
  - $y_0 = x_0$
  - $y_{i+1} = y_i \triangle x_{i+1}$
- For a monotone function f: $D \rightarrow D$ and $x_k \in \mathrm{Red}(f) = \{ d \mid d \in D \text{ and } f(d) \sqsubseteq d \}$ define
  - $y_0 = x$
  - $y_{i+1} = y_i \triangle f(y_i)$
- Theorem:
  - There exits k such that $y_{k+1} = y_k$
  - $y_k \in \mathrm{Red}(f) = \{ d \mid d \in D \text{ and } f(d) \sqsubseteq d \}$

Narrowing for Interval Analysis

- $[a, b] \triangle \perp = [a, b]$
- $[a, b] \triangle [c, d] = [$

```
        if $a = -\infty$
        then c
        else a,
    if $b = \infty$
        then d
        else b
            ]
```

Semantic equations with narrowing

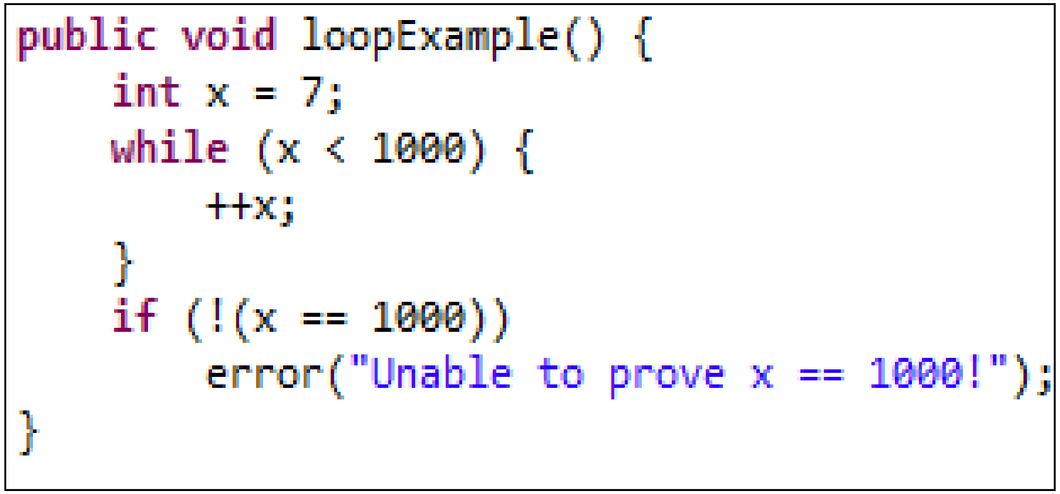


```
     public void loopExample() {
R[0]     int x = 7;  R[1]
R[2]     while (x < 1000) {
R[3]           ++x;  R[4]
         }
R[5]     if (!(x == 1000))
R[6]           error("Unable to prove x == 1000!");
     }
```

- $R[0] = \top$
  $R[1] = [7,7]$
  $R[2] = R[1] \sqcup R[4]$
  $R[2.1] = R[2.1] \vartriangle R[2]$
  $R[3] = R[2.1] \sqcap [-\infty,999]$
  $R[4] = R[3]+[1,1]$
  $R[5] = R[2]^{\#} \sqcap [1000,+\infty]$
  $R[6] = R[5] \sqcap [999,+\infty] \sqcup R[5] \sqcap [1001,+\infty]$

Analysis with widening/narrowing

- Two phases
  - Phase 1: analyze with widening until converging
  - Phase 2: use values to analyze with narrowing



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

Phase 1:

$R[0] = \top$

$R[1] = [7,7]$

$R[2] = R[1] \sqcup R[4]$

$R[2.1] = R[2.1] \triangledown R[2]$

$R[3] = R[2.1] \sqcap [-\infty,999]$

$R[4] = R[3] + [1,1]$

$R[5] = R[2] \sqcap [1001,+\infty]$

$R[6] = R[5] \sqcap [999,+\infty] \sqcup R[5] \sqcap [1001,+\infty]$

Phase 2:

$R[0] = \top$

$R[1] = [7,7]$

$R[2] = R[1] \sqcup R[4]$

$R[2.1] = R[2.1] \triangle R[2]$

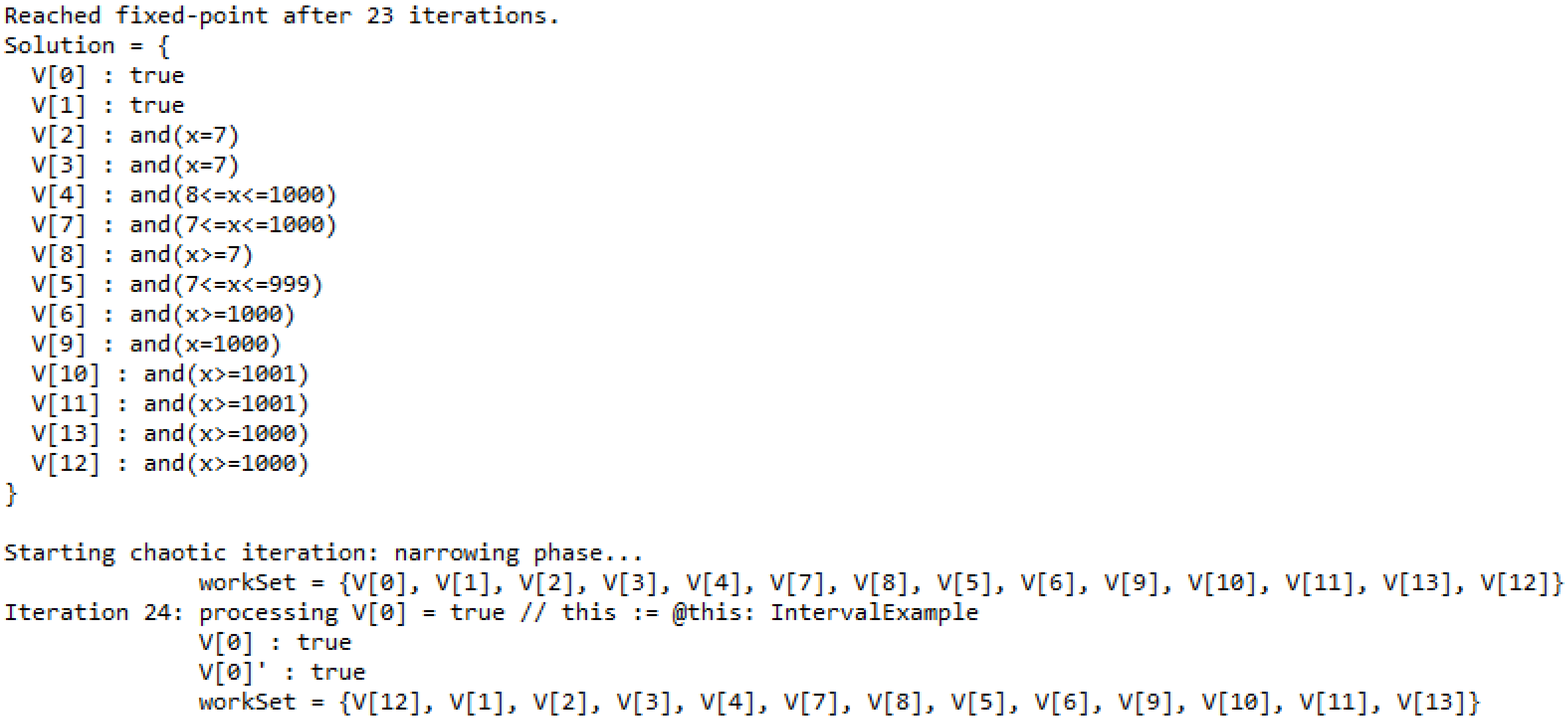
$R[3] = R[2.1] \sqcap [-\infty,999]$

$R[4] = R[3]+[1,1]$

$R[5] = R[2]^{\#} \sqcap [1000,+\infty]$

$R[6] = R[5] \sqcap [999,+\infty] \sqcup R[5] \sqcap [1001,+\infty]$

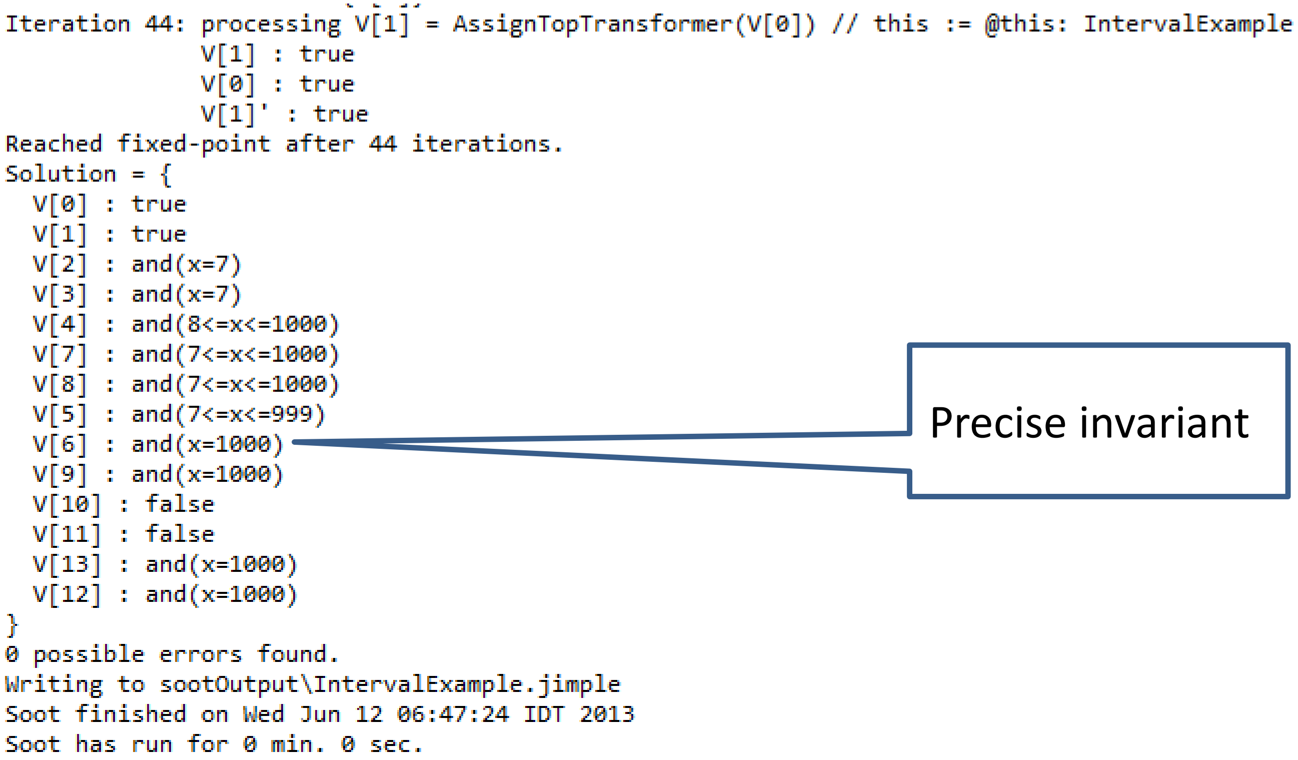
Analysis with widening/narrowing



```
Reached fixed-point after 23 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[8] : and(x>=7)
  V[5] : and(7<=x<=999)
  V[6] : and(x>=1000)
  V[9] : and(x=1000)
  V[10] : and(x>=1001)
  V[11] : and(x>=1001)
  V[13] : and(x>=1000)
  V[12] : and(x>=1000)
}

Starting chaotic iteration: narrowing phase...
              workSet = {V[0], V[1], V[2], V[3], V[4], V[7], V[8], V[5], V[6], V[9], V[10], V[11], V[13], V[12]}
Iteration 24: processing V[0] = true // this := @this: IntervalExample
              V[0] : true
              V[0]' : true
              workSet = {V[12], V[1], V[2], V[3], V[4], V[7], V[8], V[5], V[6], V[9], V[10], V[11], V[13]}
```

Analysis results widening/narrowing



```
Iteration 44: processing V[1] = AssignTopTransformer(V[0]) // this := @this: IntervalExample
              V[1] : true
              V[0] : true
              V[1]' : true
Reached fixed-point after 44 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[8] : and(7<=x<=1000)
  V[5] : and(7<=x<=999)
  V[6] : and(x=1000)
  V[9] : and(x=1000)
  V[10] : false
  V[11] : false
  V[13] : and(x=1000)
  V[12] : and(x=1000)
}
0 possible errors found.
Writing to sootOutput\IntervalExample.jimple
Soot finished on Wed Jun 12 06:47:24 IDT 2013
Soot has run for 0 min. 0 sec.
```

Precise invariant