Analysis Sensitivity

Kristopher Micinski
CIS 700 —Program Analysis: Foundations and Applications
Fall ’19, Syracuse University

We want to understand the ways in which our analyses make approximations

All analyses (for Turing-complete languages) must approximate the program in some way

Have to **at least** approximate the stack!

```
(let* ([y (lambda (x) (x x)])
  (y y))
```

Definition: the **polyvariance** of an analysis is the degree to which the analysis separates reasoning about fragments of the program to gain precision

0CFA is **monovariant**, because each lambda considered only **once** (no matter how many environments)

```
(let* ([id (lambda (x) x)]
       [f  (lambda (y) …)]
       [g  (lambda (z) …)]
       [h  (id f)])
  (id g))
```

Definition: the **polyvariance** of an analysis is the degree to which the analysis separates reasoning about fragments of the program to gain precision

0CFA is **monovariant**, because each lambda considered only **once** (no matter how many environments)

```
(let* ([id (lambda (x) x)]
       [f  (lambda (y) …)]
       [g  (lambda (z) …)]
       [h  (id f)])
  (id g))
```

In 0CFA, two lambdas reach x

Thus, two lambdas reach all **callsites** of id

```
(let* ([id (lambda (x) x)]
       [f  (lambda (y) …)]
       [g  (lambda (z) …)]
       [h  (id f)])
   (id g))
```

{(lambda (y) …),
 (lambda (z) …)}

{(lambda (y) …),
 (lambda (z) …)}

This is not what we want..

Clearly, `(id g)` should just be g, not also f

```
(let* ([id (lambda (x) x)]
       [f  (lambda (y) …)]
       [g  (lambda (z) …)]
       [h  (id f)])
  (id g))
```

Ideally…

`{(lambda (z) …)}`

Unfortunately, 0CFA conflates them!

Last week, we largely focused on the lambda calculus

In that setting, the analysis computes **flow sets**
(sets of lambdas that flow to a program point)

```
(let ([f (lambda (x) x)]
      [x (f (lambda (y) y))]
      [y (f (lambda (z) z))]
  y)
```

If we use the AAM approach, x will be represented by an address **x**, in the store

Meaning that the **first** call we see to **f** will bind x to (lambda (y) …) in the store

```
(let ([f (lambda (x) x)]
      [x (f (lambda (y) y))]
      [y (f (lambda (z) z))]
  y)
```

```
Store = {…, x |-> {(lambda (y) y)}}
```

If we use the AAM approach, x will be represented by an address **x**, in the store

But when we see the **second** call to f, we will add **another** lambda at x’s address in the store

```
(let ([f (lambda (x) x)]
      [x (f (lambda (y) y))]
      [y (f (lambda (z) z))]
  y)
Store = {…, x |-> {(lambda (y) y),
                   (lambda (z) z)}}
```

When the analysis handles a **call**, we allocate space in the store, then update as appropriate

$$\sigma' = \sigma \sqcup \{\alpha \mapsto v\}$$

“Take whatever is at address α and replace it with that joined with v”

When the analysis handles a **call**, we allocate space in the store, then update as appropriate

$$\sigma' = \sigma \sqcup \{\alpha \mapsto v\}$$

In 0CFA for LC, values are just flow sets.

Meaning this will always be set addition of a lambda

Set of syntactic lambdas is finite, thus so is its powerset!

What about when we add prims…?

For 0CFA, this x only ever considered once

```
(let ([id (lambda (x) x)]
      [x (id 2)]
      [y (id 3)])
  x)
```

(I.e., all possible values of x are conflated)

We have a choice: which abstraction for ints?

(I.e., first time we see `(id 2)`, what do we make x?)

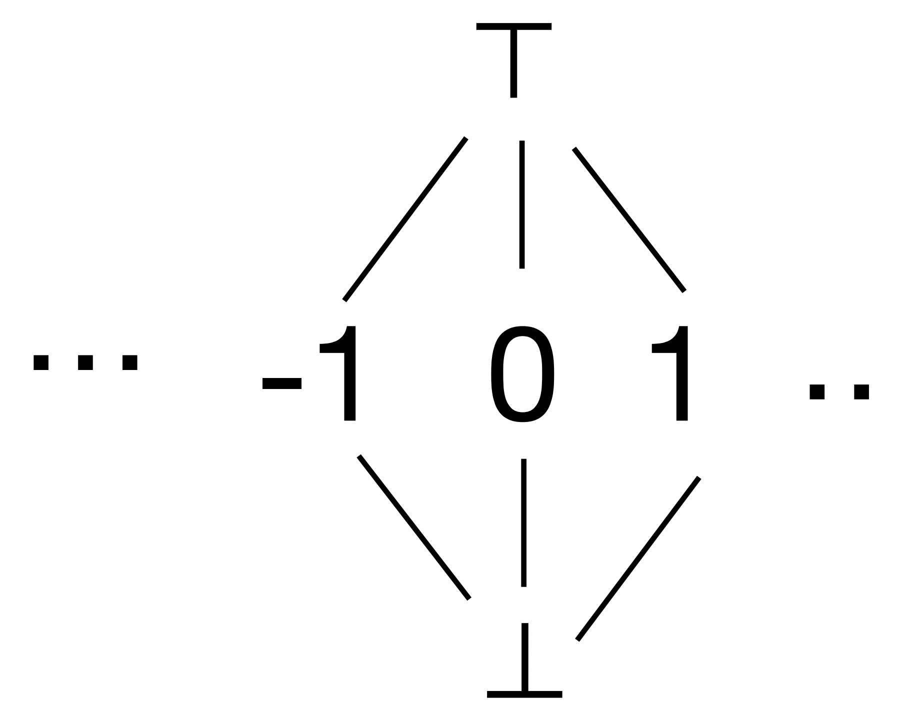
```
(let ([id (lambda (x) x)]
      [x (id 2)]
      [y (id 3)])
  x)
```

We have a choice: which abstraction for ints?

(I.e., first time we see `(id 2)`, what do we make x?)

```
(let ([id (lambda (x) x)]
      [x (id 2)]
      [y (id 3)])
  x)
```

Choice 1:
Constant propagation flat lattice



First call to id, set x to {2}

```
(let ([id (lambda (x) x)]
      [x (id 2)]
      [y (id 3)])
  x)
```

$$\sigma(x) = \{2\}$$

Second call to id, set x to $\{2\} \sqcup \{3\} = \top$

```
(let ([id (lambda (x) x)]
      [x (id 2)]
      [y (id 3)])
  x)
```

$$\sigma(x) = \top$$

Second call to id, set x to $\{2\} \sqcup \{3\} = \top$

```
(let ([id (lambda (x) x)]
      [x (id 2)]
      [y (id 3)])
  x)
```

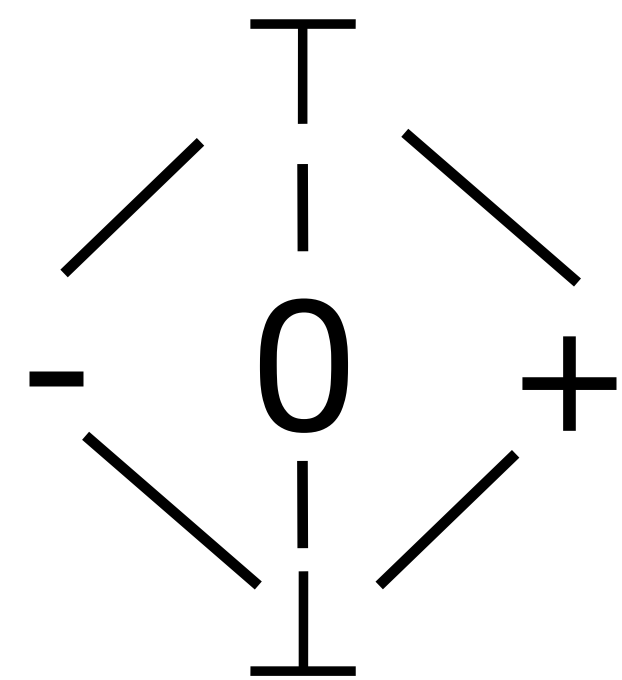
$$\sigma(x) = \top$$

In 0CFA, each time we see a call to a function, we join its arguments with the arguments we've already seen

Another example, how will loss of precision impact **if**?

```
(define (foo x)
  (if (> x 0)
      1
      0))

(foo 1)
(foo -1)
```

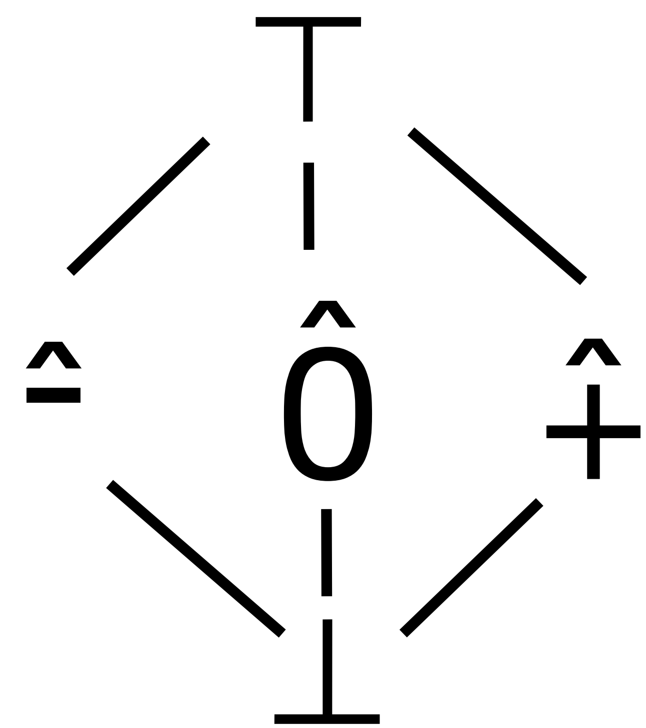


This time, assume we use sign lattice…

Another example, how will loss of precision impact **if**?

```
(define (foo x)
  (if (> x 0)
      1
      0))

(foo 1)
(foo -1)
```



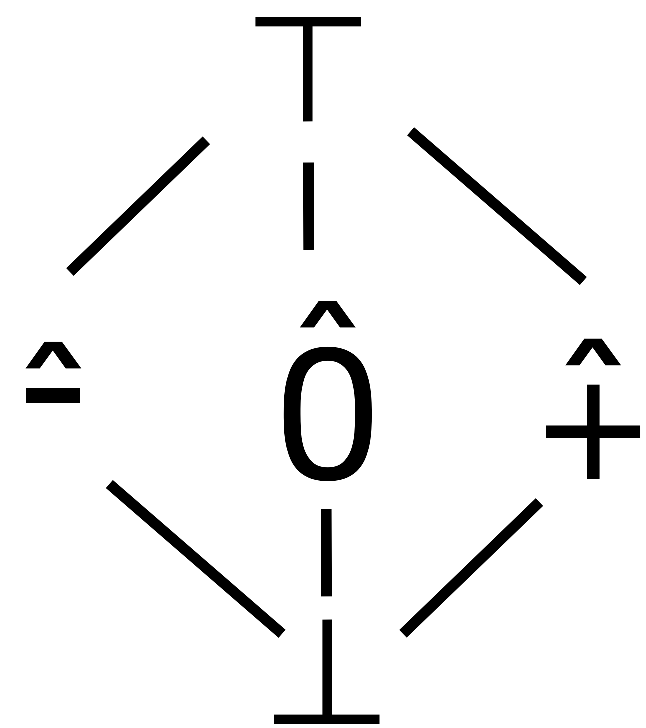
First time around... $\sigma(\mathbf{x}) = \hat{+}$

Another example, how will loss of precision impact **if**?

Consequently, analysis
says branch only goes to 1

```
(define (foo x)
  (if (> x 0)
      1
      0))

(foo 1)
(foo -1)
```

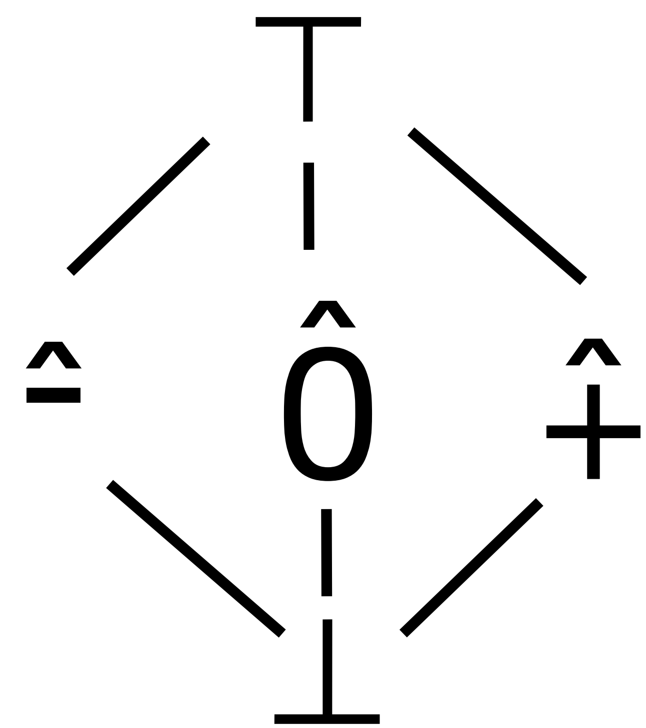


First time around... $\sigma(\mathbf{x}) = \hat{+}$

Another example, how will loss of precision impact **if**?

```
(define (foo x)
  (if (> x 0)
      1
      0))

(foo 1)
(foo -1)
```



Second time around... $\sigma(\mathbf{x}) = \hat{+} \sqcup \hat{-} = \top$

Another example, how will loss of precision impact **if**?

```
(define (foo x)
  (if (> x 0)
      1    ⊤
      0))

(foo 1)
(foo -1)
```

Now analysis goes to **both** branches

And all callsites return $\top$

When the analysis handles a **call**, we allocate space in the store, then update as appropriate

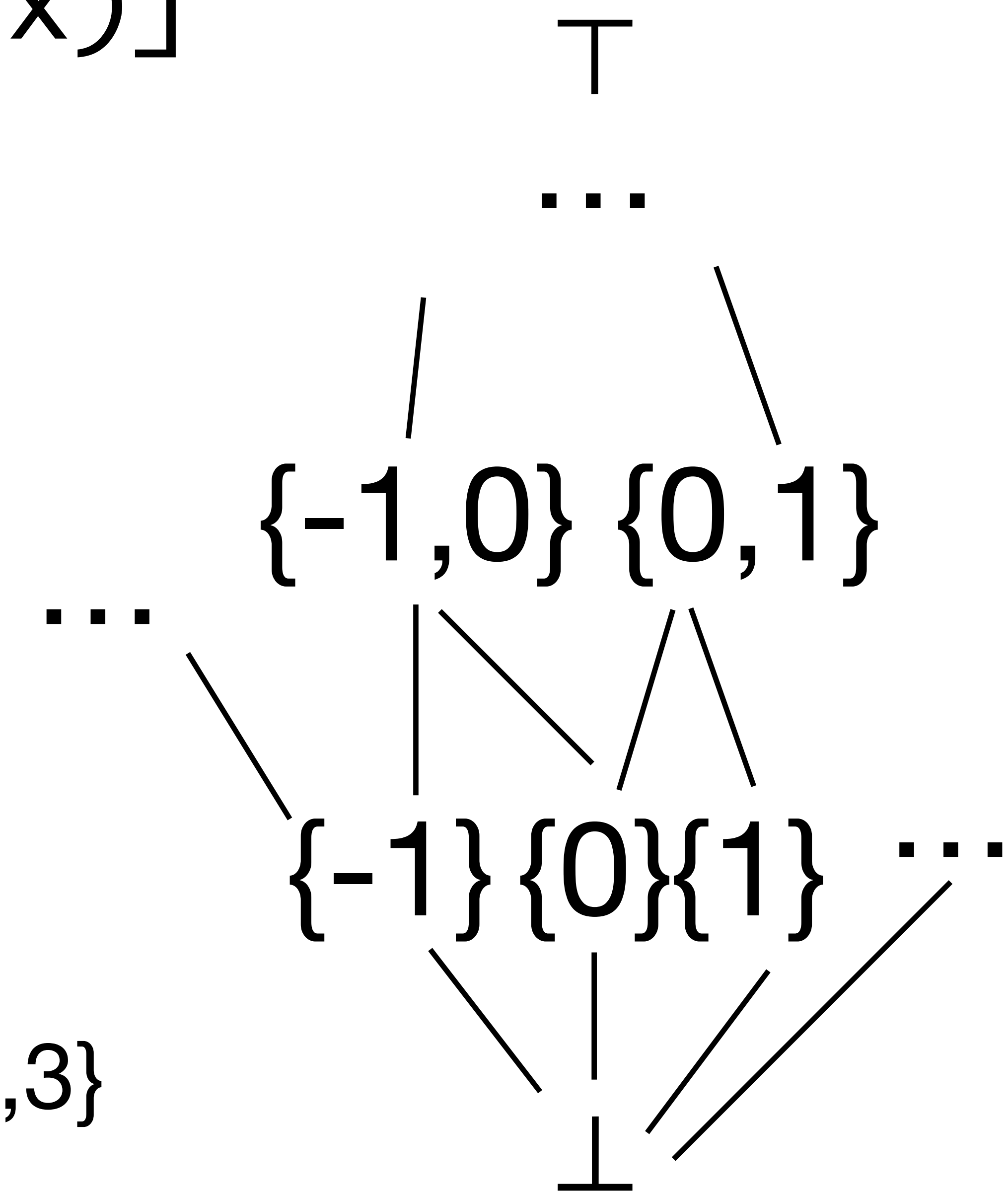
$$\sigma' = \sigma \sqcup \{\alpha \mapsto v\}$$

This is where possible conflation / approximation happens!!!

```
(let ([id (lambda (x) x)]
      [x (f (lambda (y) y))]
      [y (f (lambda (z) z))]
  y)
```

Other possible choices for the lattice…

```
(let ([id (lambda (x) x)]
      [x (id 2)]
      [y (id 3)])
  x)
```

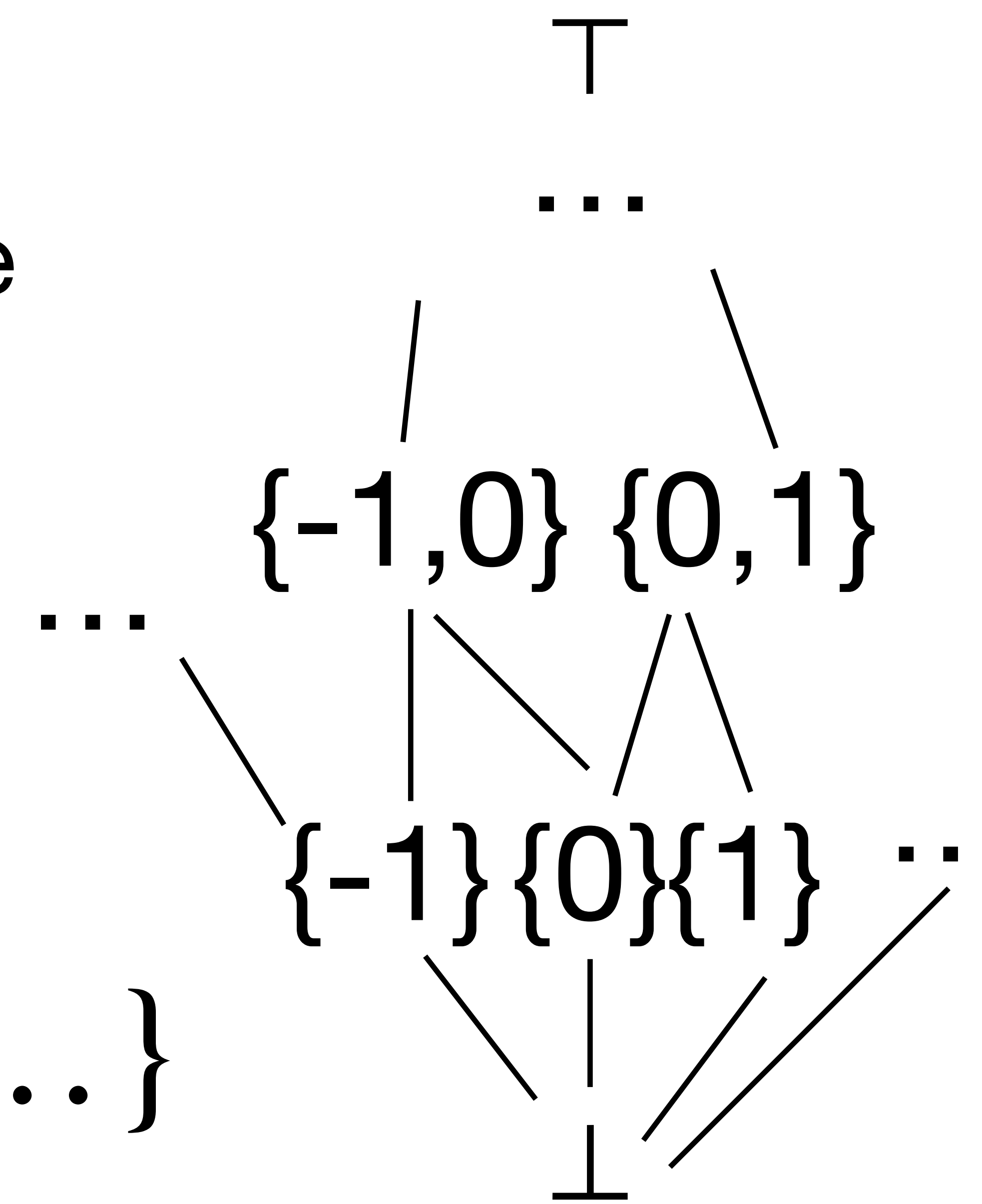


Choice 2:
“Arbitrary sets of constants”

Allows us to approximate x as {2,3}

(A little better: we know it **can’t** be 0,1, …)

```
(letrec
    ([f (lambda (x) (if (= x 0)
                        x
                        (f (+ x 1))))])
  (f 2))
```



Unfortunately, using this lattice doesn't ensure termination

$$x \mapsto \{2\}$$
$$x \mapsto \{2, 3\}$$
$$x \mapsto \{2, 3, 4\}$$
$$x \mapsto \{2, 3, 4, \ldots\}$$

```
(letrec
    ([f (lambda (x) (if (= x 0)
                        x
                        (f (+ x 1))))])
  (f 2))
```

We can use a **widening** operator

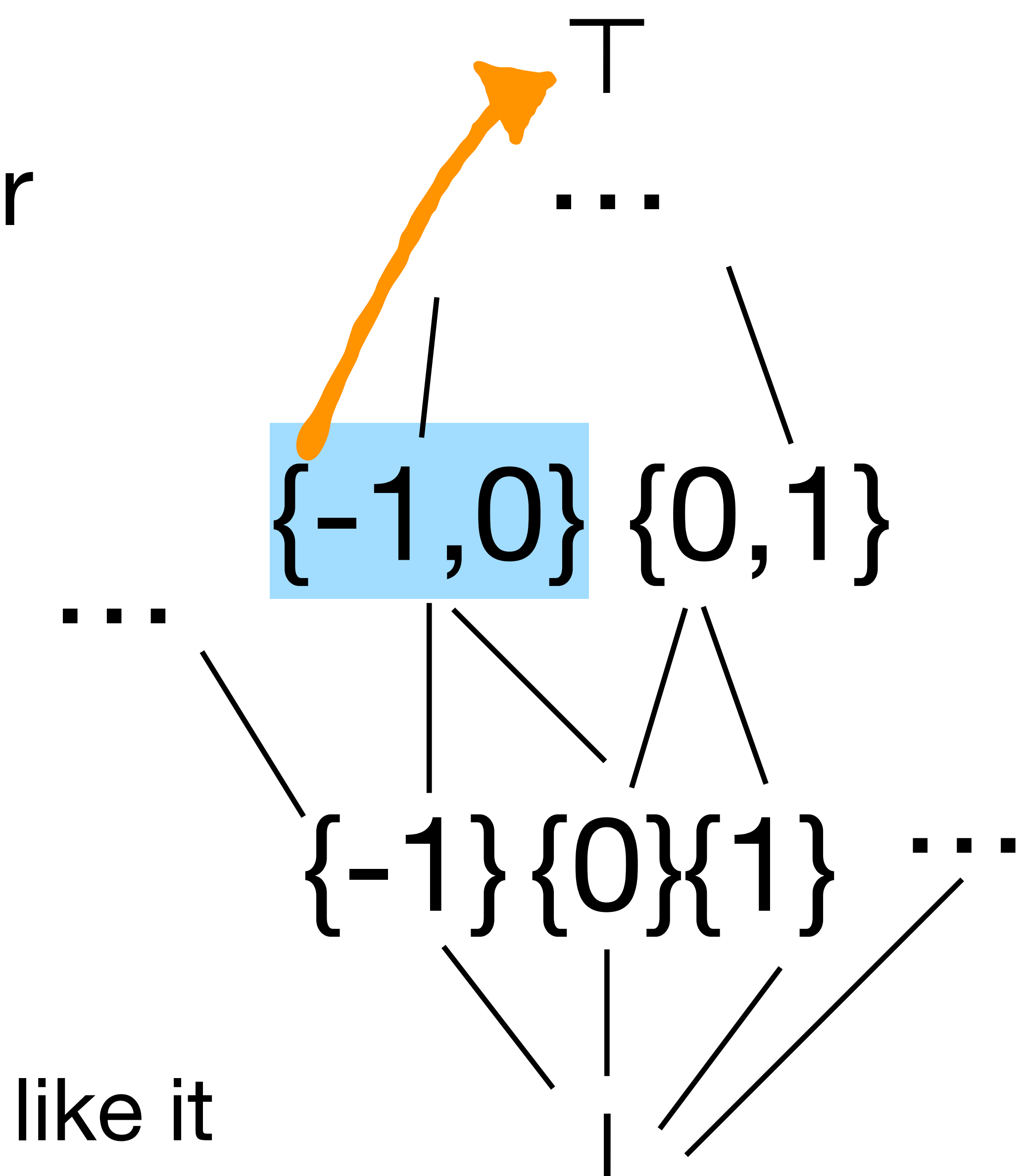
$$x \mapsto \{2\}$$

$$x \mapsto \{2,3\}$$

$$x \mapsto \{2,3,4\}$$

Hypothetical

$$x \nabla \{5\} = \top$$



Intuitively, widening says "this looks like it won't terminate, bump it up the lattice"

We won’t talk much about widening (at least for now)

High level thing to know about it:
Allows you to use AI with an infinite domain by allowing you to “bump up to top” when you realize you might be about to diverge

In practice, I do not know much about designing widening / narrowing operators. I usually choose a better finite domain instead

However, finite domains give strictly **worse precision** than infinite domains + widening/narrowing

But does this loss of precision matter in practice for analyses we care about? I don't know (I am not sure if anyone does know).

Comparing the Galois Connection and Widening/ Narrowing Approaches to Abstract Interpretation, Patrick and Rhadia Cousot, LNCS '92

When the analysis handles a **call**, we allocate space in the store, then update as appropriate

$$\sigma' = \sigma \sqcup \{\alpha \mapsto v\}$$

This is where possible conflation / approximation happens!!!

Thus, if we want to **improve** precision, we need to **avoid** this conflation from happening!

Let's look back at this example again…

```
(let* ([f (lambda (x) x)]
       [a (f (lambda (y) y))^l1]
       [b (f (lambda (z) z))^l2]
  b)
```

```
(let* ([f (lambda (x) x)]
       [a (f (lambda (y) y))^l1]
       [b (f (lambda (z) z))^l2]
  b)
```

If we were to run this program **concretely** using a CESK*-style semantics

x would be bound to α_1, and

$\sigma(\alpha_1)$ = { (lambda (y) y), [] }

y would be bound to α_2, and

$\sigma(\alpha_2)$ = { (lambda (z) z), [] }

```
(let* ([f (lambda (x) x)]
       [a (f (lambda (y) y))^l1]
       [b (f (lambda (z) z))^l2]
  b)
```

In the AAM style, x is bound to x (an **address**)

And…

```
σ(x) = { (lambda (y) y),
         (lambda (x) x) }
```

```
(let* ([f (lambda (x) x)]
       [a (f (lambda (y) y))^l1]
       [b (f (lambda (z) z))^l2]
  b)
```

What we **really** want to say is that f has two **different** behaviors:

- f called via the callsite at l1
- f called via the callsite at l2

Considering these cases **separately** leads to better precision!

```
(let* ([f (lambda (x) x)]
       [a (f (lambda (y) y))^l1]
       [b (f (lambda (z) z))^l2]
  b)
```

How can we pick α_1 and αs such that….

x would be bound to α_1, and
$\sigma(\alpha_1)$ = { (lambda (y) y) }

y would be bound to α_2, and
$\sigma(\alpha_2)$ = { (lambda (z) z) }

Idea: allocate variables based on the function that **called** them

```
(let ([id (lambda (x) x)]
      [x (f (lambda (y) y))^l1]
      [y (f (lambda (z) z))^l2]
  y)
```

Thus, we consider two possible “runs" of `id`

`id` called via `l1`

`id` called via `l2`

`(lambda (x) x)`

$$\rho = [x \mapsto \langle x, l1 \rangle]$$

$$\sigma(\langle x, l1 \rangle) = \{(\lambda(y)\ y)\}$$

`(lambda (x) x)`

$$\rho = [x \mapsto \langle x, l2 \rangle]$$

$$\sigma(\langle x, l2 \rangle) = \{(\lambda(z)\ z)\}$$

Idea: allocate variables based on the function that **called** them

```
(let ([id (lambda (x) x)]
      [x (f (lambda (y) y))^l1]
      [y (f (lambda (z) z))^l2]
  y)
```

```
(lambda (x) x)
```

$$\rho = [x \mapsto \langle x, l1 \rangle]$$

$$\sigma(\langle x, l1 \rangle) = (\lambda(y)\ y)$$

```
(lambda (x) x)
```

$$\rho = [x \mapsto \langle x, l2 \rangle]$$

$$\sigma(\langle x, l2 \rangle) = (\lambda(z)\ z)$$

We are adding **call** sensitivity for the value at the var x

Recall, using constant propagation lattice, 0CFA calculates x = ⊤

Let's say we use the trick in the previous slide…

Now we consider two different cases for x within foo:

- `foo` called via `l1`
- `foo` called via `l2`

```
(define (foo x)
  (if (< x 0)
    0
    1))
```

```
(let*
  ([x (foo 5)l1]
   [y (foo -5)l2])
  x)
```

$\rho = [x \mapsto \langle x, l1 \rangle]$ $\rho = [x \mapsto \langle x, l2 \rangle]$

$\sigma(\langle x, l1 \rangle) = \{5\}$ $\sigma(\langle x, l2 \rangle) = \{-5\}$

To apply this trick in general…
When allocating variables, allocate not based on **name**, but **name+label of most recent callsite**

We call this 1CFA, since it’s a control-flow analysis keeping 1 calling context

We *strategically* lose precision for bindings more than 1 callsite away!

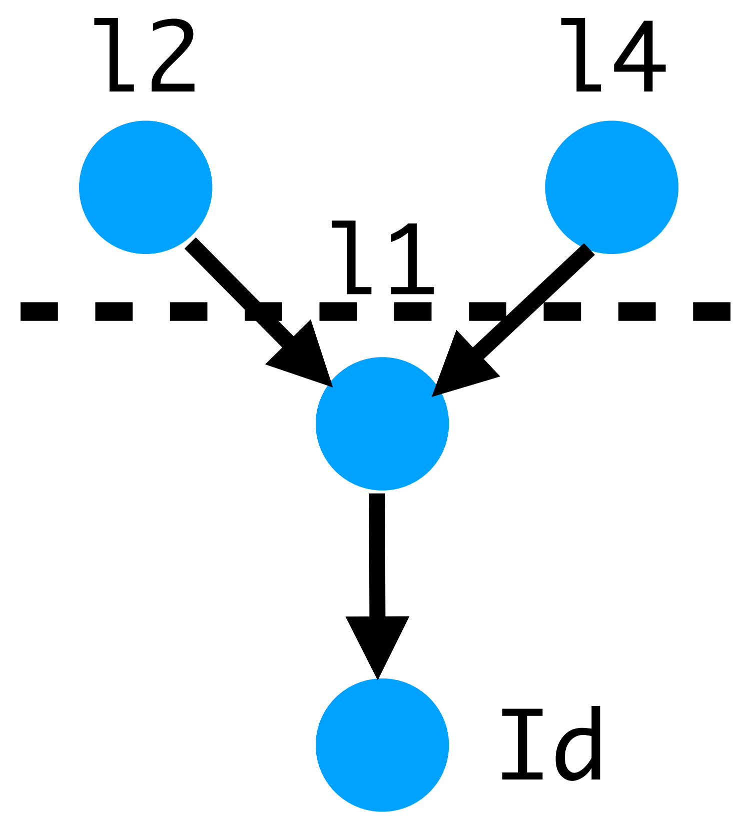
```
(let*
  ([id (lambda (y) y)]
   [id1 (lambda (x) (id x)^l1)]
   [x (id1 5)^l2]
   [y (id1 -5)^l4])
  x)
```

Conceptually, 1-CFA is “forgetting” callsites past l1

$$x \mapsto \langle x, l2 \rangle$$

$$x \mapsto \langle x, l4 \rangle$$

$$y \mapsto \langle y, l1 \rangle$$



$$\hat{\sigma}(\langle x, l2 \rangle) = \{5\}$$

$$\hat{\sigma}(\langle x, l2 \rangle) = \{-5\}$$

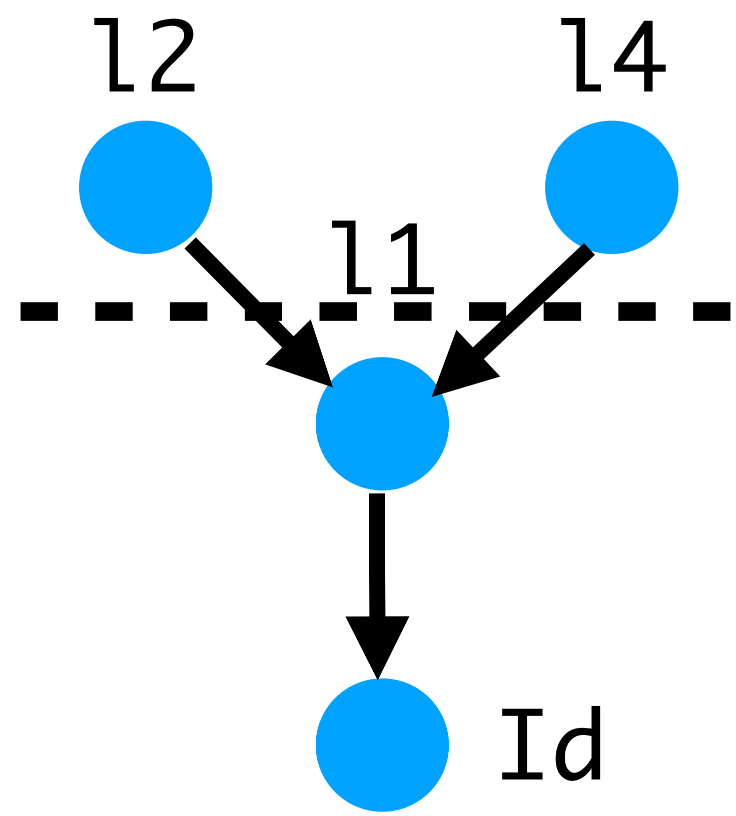
$$\hat{\sigma}(\langle y, l1 \rangle) = \{5\} \sqcup \{-5\} = \top$$

Solution: 2-CFA?

Now variables are precise up to most recent **two** callsites

Add more polyvariance $\langle y, l1, l2 \rangle$

$\langle y, l1, l4 \rangle$




```
(let*
  ([id (lambda (y) y)]
   [id1 (lambda (x) (id x)^l1)]
   [x (id1 5)^l2]
   [y (id1 -5)^l4])
  x)
```

$$\big(\lambda(x)\ e\big), c \Rightarrow \big(\lambda(x)\ e\big), c$$

$$\big(e_0\ e_1\big)^l \quad \frac{\big(\lambda(x)\ e'\big), c \Rightarrow e_0 \qquad v, c \Rightarrow e_1}{v, c \Rightarrow x, \lfloor l :: c \rfloor_k}$$

$$\frac{\big(\lambda(x)\ e\big) \Rightarrow e_0 \qquad v \Rightarrow e}{v \Rightarrow (e_0\ e_1)}$$