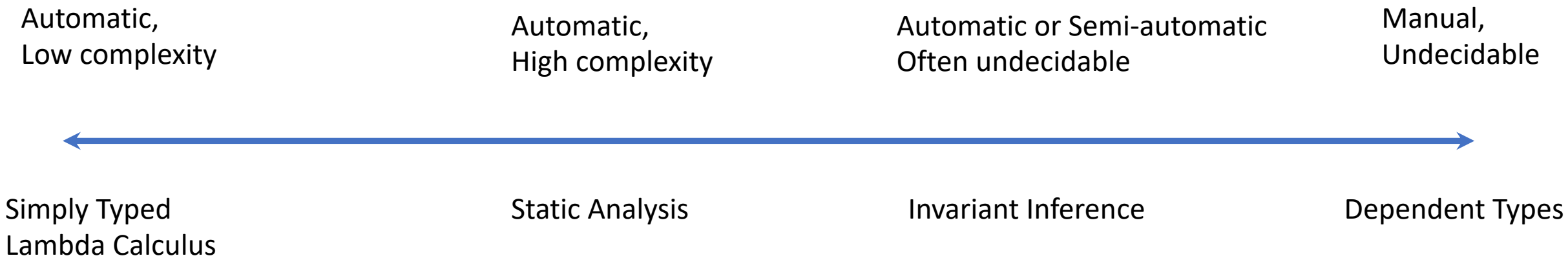
Set Constraints

CS242

Lecture 14

Approaches to Proving Properties of Programs



Closure Analysis: The Problem

- A *call graph* is a graph where
  - The nodes are function (method) names
  - There is a directed edge (f,g) if f may call g

- Call graphs can be overestimates
  - If f may call g at run time, there must be an edge (f,g) in the call graph
  - If f cannot call g at run time, there is no requirement on the graph

- Call graphs are used heavily in implementations of programming languages

Recall: Untyped Lambda Calculus

$$e \rightarrow x \mid \lambda x.e \mid e\ e$$

A Definition

- Assume all bound variables are unique
  - So a bound variable uniquely identifies a function
  - Can be done by renaming variables

- For each application $e_1\ e_2$, what is the set of lambda terms $L(e_1)$ to which e_1 may evaluate?
  - $L(\ldots)$ is a set of static, or syntactic, lambdas
  - $L(\ldots)$ defines a call graph
    - the set of functions that may be called by an application

A More General Definition

- To compute L(…) for applications, we must compute it for every expression.

- Define:

L(e) is the set of syntactic lambda abstractions to which e may evaluate

- The problem is to compute L(e) for every expression e

Defining L(…)

$\lambda x.e$

$\lambda x.e \subseteq L(\lambda x.e)$

$e_1\ e_2$

for each $\lambda x.e \subseteq L(e_1)$

$L(e_2) \subseteq L(x)$

$L(e) \subseteq L(e_1\ e_2)$

The actual argument of the call flows to the formal argument

The value of the application includes the value of the function body

Rephrasing the Constraints with $\subseteq$

The following constraints have the same least solution as the original constraints:

$\lambda x.e$

$\lambda x.e \subseteq L(\lambda x.e)$

$e_1\ e_2$

$\lambda x.e_0 \subseteq L(e_1) \Rightarrow (L(e_2) \subseteq L(x) \wedge L(e_0) \subseteq L(e_1\ e_2))$

Note: Each $L(e)$ is a constraint variable

Each $\lambda x.e$ is a constant

Example ((λx.x) (λy.y)) (λz.z)

$$\lambda x.x \subseteq L(\lambda x.e)$$

$$\lambda y.y \subseteq L(\lambda y.y)$$

$$\lambda z.z \subseteq L(\lambda z.z)$$

$$L(\lambda y.y) \subseteq L(x)$$

$$L(x) \subseteq L((\lambda x.x)\ (\lambda y.y))$$

$$L(\lambda z.z) \subseteq L(y)$$

$$L(y) \subseteq L(((\lambda x.x)\ (\lambda y.y))\ (\lambda z.z))$$

Solution:

$$L(\lambda x.x) = \lambda x.x$$

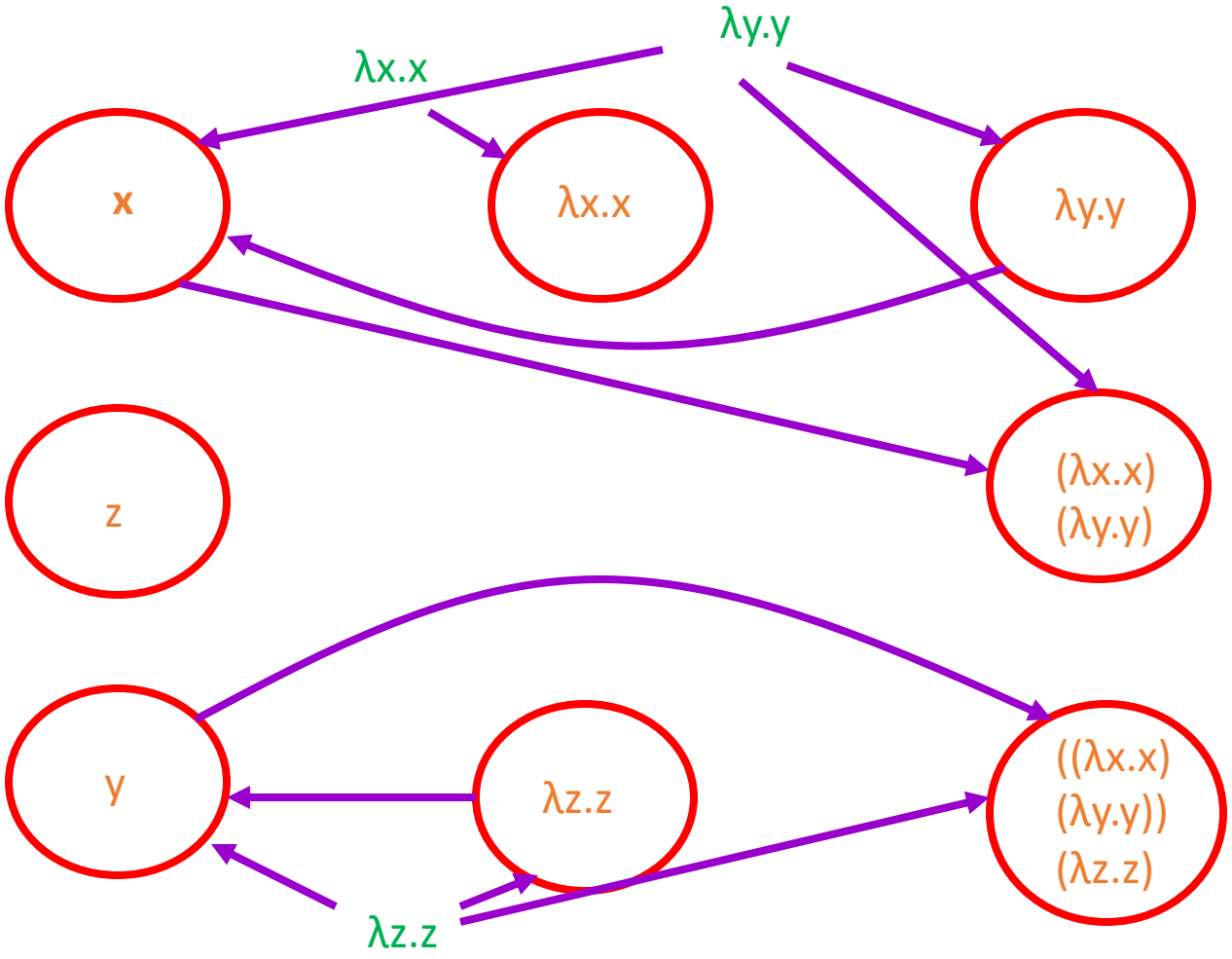
$$L(\lambda y.y) = \lambda y.y$$

$$L(\lambda z.z) = \lambda z.z$$

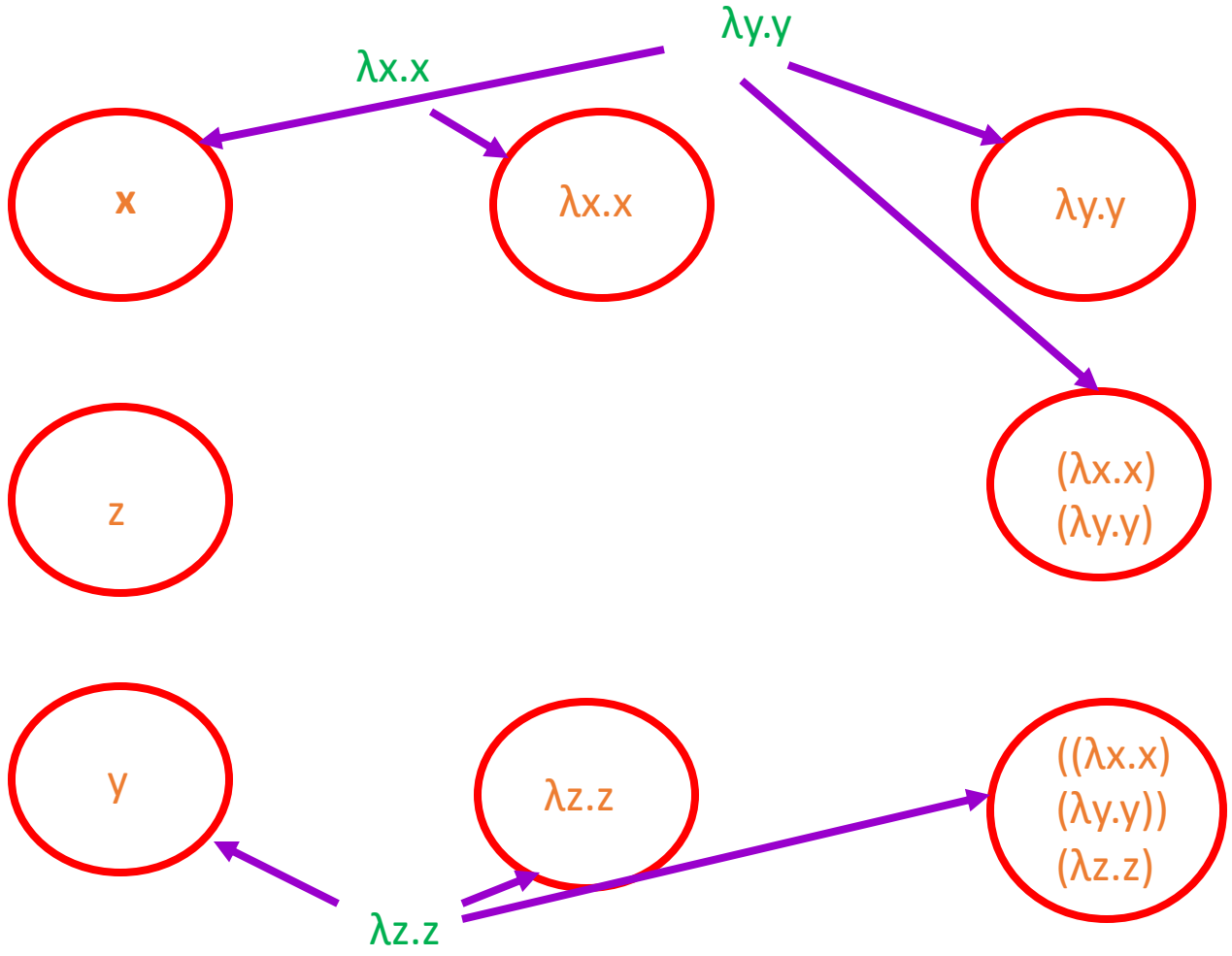
$$L(\lambda y.y) = L(x) = L((\lambda x.x)\ (\lambda y.y))$$

$$L(\lambda z.z) = L(y) = L(((\lambda x.x)\ (\lambda y.y))\ (\lambda z.z))$$

The Example ((λx.x) (λy.y)) (λz.z) with Graphs



The Solution for ((λx.x) (λy.y)) (λz.z)



The solution is given by the edges whose source is a lambda.

Set Constraints

- A finite set of *constructors*

$$a, b, c, f, g, h \in C$$

- Each constructor c has an *arity* a(c)
  - Can be 0

- Terms
  - $T = \{ f(t_1,\ldots,t_{a(f)}) \mid f \in C, t_i \in T \}$

Set Constraints

$L \rightarrow 0 \mid x \mid c(L_1,\ldots,L_n) \mid L_1 \cup L_2$

$R \rightarrow 1 \mid x \mid c(R_1,\ldots,R_n) \mid R_1 \cap R_2$

Constraints: $L \subseteq R$ or $c \subseteq R \Rightarrow L' \subseteq R'$

Solving

$S, L_1 \cup L_2 \subseteq R$	→	$S, L_1 \cup L_2 \subseteq R, L_1 \subseteq R, L_2 \subseteq R$
$S, L \subseteq R_1 \cap R_2$	→	$S, L \subseteq R_1 \cap R_2, L \subseteq R_1, L \subseteq R_2$
$S, c(L_1,\ldots,L_n) \subseteq c(R_1,\ldots,R_n)$	→	$S, c(L_1,\ldots,L_n) \subseteq c(R_1,\ldots,R_n), L_1 \subseteq R_1, \ldots, L_n \subseteq R_n$
$S, c \subseteq R \Rightarrow L' \subseteq R', c \subseteq R$	→	$S, c \subseteq R \Rightarrow L' \subseteq R', c \subseteq R, L' \subseteq R'$
$S, L \subseteq x, x \subseteq R$	→	$S, L \subseteq x, x \subseteq R, L \subseteq R$

No solutions if $c(\ldots) \subseteq 0$, $1 \subseteq c(\ldots)$, or $c(\ldots) \subseteq d(\ldots)$

Add Integers ...

$$e \rightarrow x \mid \lambda x.e \mid e\ e \mid i$$

Extend L(…) With Integers

$\lambda x.e$

$\lambda x.e \subseteq L(\lambda x.e)$

i $\quad i \subseteq L(i)$

Idea: Treat integers like lambdas and track their flow.

$e_1\ e_2$

for each $\lambda x.e \subseteq L(e_1)$

$L(e_2) \subseteq L(x)$

$L(e) \subseteq L(e_1\ e_2)$

Application: Type Inference!

$\lambda x.e$

$\lambda x.e \subseteq L(\lambda x.e)$

i $\quad$ $i \subseteq L(i)$

$e_1\ e_2$

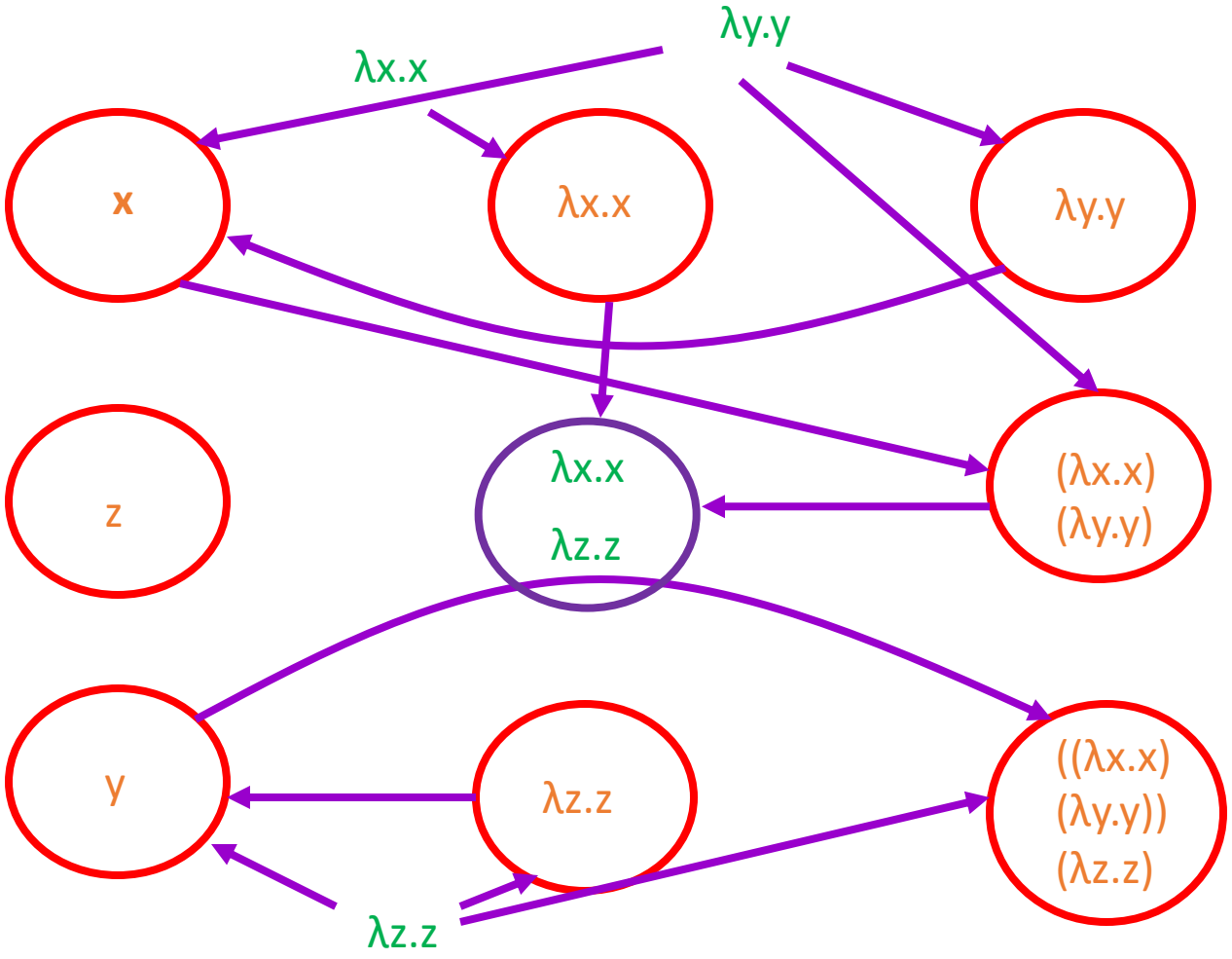
for each $\lambda x.e \subseteq L(e_1)$

$L(e_2) \subseteq L(x)$

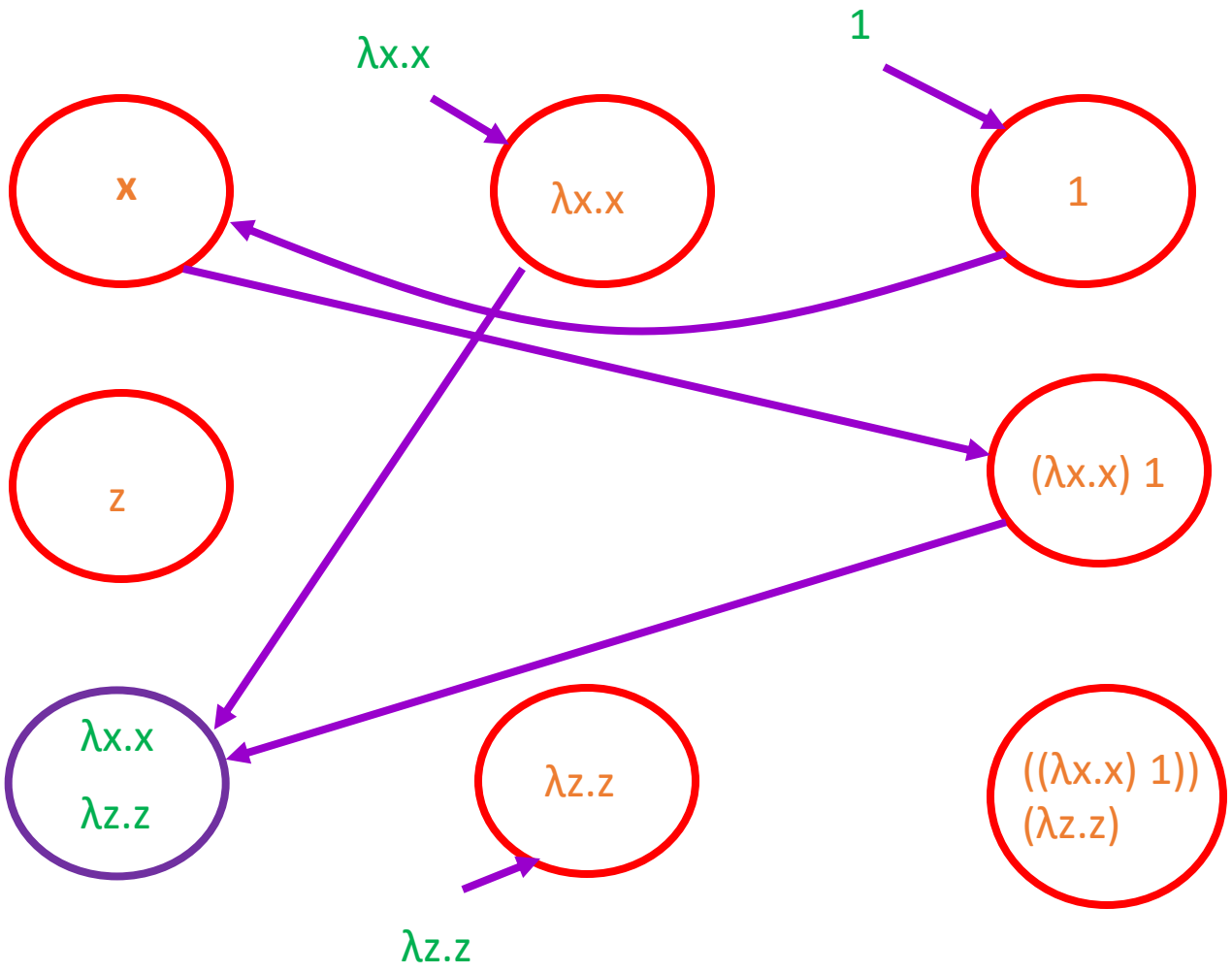
$L(e) \subseteq L(e_1\ e_2)$

$L(e_1) \subseteq \{\lambda x.e \mid \lambda x.e$ is a lambda abstraction in the program$\}$

An Example ((λx.x) (λy.y)) (λz.z)



An Example ((λx.x) 1) (λz.z)



There is a path from an integer constant to the set of lambdas – a type error.

Discussion

- Idea: For each application $e_1\ e_2$, check if $i \subseteq L(e_1)$
  - If yes, it's a type error
- More general than simply typed lambda calculus
  - Intuition: Relax = constraints to $\subseteq$
- Note every pure lambda (with no integers) has a type

Summary So Far

- Set constraints use subset relationships instead of equality

- Natural interpretation as a graph
  - Nodes are sets
  - Directed edges are inclusion constraints
  - Solving the constraints = adding edges to the graph
  - Dynamic transitive closure $O(n^3)$

- Many applications of closure analysis in functional programming
  - Often the first analysis to be done

The Next Step

- So far we've only considered sets of atoms
  - Sets where the elements have no structure

- Now consider sets where the elements can be sets of data types
  - E.g., cons(A,B) is the cross product of cons(a,b) for every a in A and b in B

Recall: Simple Type Inference Rules

$$\frac{}{A, x: \alpha_x \vdash x : \alpha_x} \quad [\text{Var}]$$

$$\frac{A, x: \alpha_x \vdash e : t}{A \vdash \lambda x: \alpha_x . e : \alpha_x \to t} \quad [\text{Abs}]$$

$$\frac{t = t' \to \beta \qquad A \vdash e_1 : t \qquad A \vdash e_2 : t'}{A \vdash e_1\, e_2 : \beta} \quad [\text{App}]$$

A Small Change

$$\frac{}{A, x: \alpha_x \vdash x : \alpha_x}\ [\text{Var}]$$

$$\frac{A, x: \alpha_x \vdash e : t}{A \vdash \lambda x.e : \alpha_x \to t}\ [\text{Abs}]$$

$$\frac{t \subseteq t' \to \beta \qquad A \vdash e_1 : t \qquad A \vdash e_2 : t'}{A \vdash e_1\, e_2 : \beta}\ [\text{App}]$$

Recall: Solving the (Equality) Constraints

Apply the following rewrite rules until no new constraints can be added

$S, t = \alpha \Rightarrow S, t = \alpha, \alpha = t$ [Reflexivity]

$S, \alpha = t_1, \alpha = t_2 \Rightarrow S, \alpha = t_1, \alpha = t_2, t_1 = t_2$ [Transitivity]

$S, t_1 \to t_2 = t_3 \to t_4 \Rightarrow S, t_1 \to t_2 = t_3 \to t_4, t_1 = t_3, t_2 = t_4$ [Structure]

Solving the Subset Constraints

Apply the following rewrite rules until no new constraints can be added

$S, t_1 \subseteq t_2, t_2 \subseteq t_3 \quad \Rightarrow \quad S, t_1 \subseteq t_2, t_2 \subseteq t_3, t_1 \subseteq t_3$ [Transitivity]

$S, t_1 \to t_2 \subseteq t_3 \to t_4 \quad \Rightarrow \quad S, t_1 \to t_2 \subseteq t_3 \to t_4, t_1 \,?\, t_3, t_2 \,?\, t_4$ [Subtyping]

What is the subtyping rule for $t_1 \to t_2 \subseteq t_3 \to t_4$?

Start With Something Else ...

What is the subtyping rule for $cons(A,B) \subseteq cons(A',B')$?

$$cons(A,B) \subseteq cons(A',B') \Rightarrow A \subseteq A' \wedge B \subseteq B'$$

What is the subtyping rule for $t_1 \to t_2 \subseteq t_3 \to t_4$?

$$t_1 \to t_2 \subseteq t_3 \to t_4 \Rightarrow t_3 \subseteq t_1 \wedge t_2 \subseteq t_4$$

Terminology

- We say that the function type constructor is
  - *Contravariant* in the first argument (the domain)
  - *Covariant* in the second argument (the range)

- Constructors in set constraints have fixed variance
  - Covariant, contravariant, or invariant in every argument position
  - Examples: cons(+,+) - → +

- Note that contravariance adds nothing to the graph representation
  - The directed edge is just added in the opposite direction for contravariant relationships
  - And in both directions for invariant relationships

Solving the Subset Constraints

Apply the following rewrite rules until no new constraints can be added

$S, t_1 \subseteq t_2, t_2 \subseteq t_3 \quad \Rightarrow \quad S, t_1 \subseteq t_2, t_2 \subseteq t_3, t_1 \subseteq t_3$ [Transitivity]

$S, t_1 \to t_2 \subseteq t_3 \to t_4 \quad \Rightarrow \quad S, t_1 \to t_2 \subseteq t_3 \to t_4, t_3 \subseteq t_1, t_2 \subseteq t_4$ [Subtyping]

Comparisons

- The lambda calculus with type equality
- The lambda calculus with subtyping
- Which is equivalent to closure analysis

Recall

$\lambda x.e$

$\lambda x.e \subseteq L(\lambda x.e)$

$i \qquad i \subseteq L(i)$

$e_1\ e_2$

for each $\lambda x.e \subseteq L(e_1)$

$L(e_2) \subseteq L(x)$

$L(e) \subseteq L(e_1\ e_2)$

$L(e_1) \subseteq \{\lambda x.e \mid \lambda x.e \text{ is a lambda abstraction in the program}\}$

The Analogy ...

$e_1\ e_2$

for each $\lambda x.e \subseteq L(e_1)$

$L(e_2) \subseteq L(x)$

$L(e) \subseteq L(e_1\ e_2)$

$L(e_1) \subseteq \{\lambda x.e \mid \lambda x.e$ is a lambda abstraction in the program$\}$

$$\frac{t \subseteq t' \to \beta \qquad A \vdash e_1 : t \qquad A \vdash e_2 : t'}{A \vdash e_1\ e_2 : \beta}$$

For each $t_1 \to t_2 \subseteq t$
We have $t_1 \to t_2 \subseteq t' \to \beta$
So $t' \subseteq t_1$ and $t_2 \subseteq \beta$

Observe $L(e_2) \equiv t'$, $L(x) \equiv t_1$, $L(e) \equiv t_2$, $L(e_1\ e_2) \equiv \beta$

Control Flow Graphs in OO Languages

- Consider a method call $e_0.f(e_1,\ldots,e_n)$
- To build a control-flow graph, we need to know which f methods may be called
  - Depends on the class of e_0 at runtime
- The problem:
  - For each expression, estimate the set of classes it could evaluate to at run time

An OO Language

P ::= C_1 . . . C_n E

C ::= class ClassId M_1 . . . M_n

M ::= MId(Id) E

E ::= Id := E | E.MId(E) | E;E | new ClassId | if E E E

Example Program

```
class A
    foo(x) x.bar(x)
    bar(y)  z := new B;  z.bar(y)

class B
    bar(w) w.bar(new B)
```

Constraints

id := e

$C(e) \subseteq C(id)$

$C(e) \subseteq C(id := e)$

$e_1; e_2$

$C(e_2) \subseteq C(e_1; e_2)$

new A

$A \subseteq C(\text{new } A)$

if e_1 e_2 e_3

$C(e_2) \subseteq C(\text{if } e_1\ e_2\ e_3)$

$C(e_3) \subseteq C(\text{if } e_1\ e_2\ e_3)$

$e_0.f(e_1)$

for each class A with a method f(x) e

$A \subseteq C(e_0) \Rightarrow$

$C(e_1) \subseteq C(x) \wedge$

$C(e) \subseteq C(e_0.f(e_1))$

Example Program w/Constraints

```
class A
    foo(x) (new A).bar(x)
    bar(y)  z := new B;  z.bar(z)

class B
    bar(w) w.bar(new B)
```

$A \subseteq C(\text{new } A)$

$C(x) \subseteq C(y)$

$C(z := \text{new } B;\ z.bar(z)) \subseteq C((\text{new } A).bar(x))$

$B \subseteq C(\text{new } B)$

$C(\text{new } B) \subseteq C(z)$

$C(\text{new } B) \subseteq C(z := \text{new } B)$

$B \subseteq C(z)$

$C(z) \subseteq C(w)$

$C(w.bar(\text{new } B)) \subseteq C(z.bar(y))$

$B \subseteq C(w)$

$C(\text{new } B) \subseteq C(w)$

$C(w.bar(\text{new } B)) \subseteq C(w.bar(\text{new } B))$

Notes

- Receiver class analysis of OO languages and control flow analysis of functional languages are the same problem

- Receiver class analysis is important in practice
  - Heavily object-oriented code pays a high price for the indirection in method calls
  - If we can show that only one method can be called, the function can be statically bound
    - Or even inlined and optimized

Type Safety

- Notice that our OO language is untyped
  - We can run (new A).f(0) even if A has no f method
  - Gives a runtime error

- By adding upper bounds to the constraints, we can make receiver class analysis into a type inference procedure for our language

Type Inference

id := e

$C(e) \subseteq C(id)$

$C(e) \subseteq C(id := e)$

$e_1; e_2$

$C(e_2) \subseteq C(e_1; e_2)$

new A

$A \subseteq C(\text{new } A)$

if e_1 e_2 e_3

$C(e_2) \subseteq C(\text{if } e_1\ e_2\ e_3)$

$C(e_3) \subseteq C(\text{if } e_1\ e_2\ e_3)$

$C(e_1) \subseteq \text{Bool}$

$e_0.f(e_1)$

for each class A with a method f(x) e

$A \subseteq C(e_0) \Rightarrow$

$C(e_1) \subseteq C(x) \wedge$

$C(e) \subseteq C(e_0.f(e_1))$

$C(e_0) \subseteq \{ A \mid A \text{ has an f method} \}$

Example Revisited: Type Safety

```
class A
    foo(x) (new A).bar(x)
    bar(y)  z := new B;  z.bar(z)

class B
    bar(w) w.bar(new B)
```

$A \subseteq C(\text{new } A)$

$C(x) \subseteq C(y)$

$C(z := \text{new } B;\ z.bar(z)) \subseteq C((\text{new } A).bar(x))$

$B \subseteq C(\text{new } B)$

$C(\text{new } B) \subseteq C(z)$

$C(\text{new } B) \subseteq C(z := \text{new } B)$

$B \subseteq C(z)$

$C(z) \subseteq C(w)$

$C(w.bar(\text{new } B)) \subseteq C(z.bar(y))$

$B \subseteq C(w)$

$C(\text{new } B) \subseteq C(w)$

$C(w.bar(\text{new } B)) \subseteq C(w.bar(\text{new } B))$

Type Inference (Cont.)

- These constraints may not have a solution
- If there is a solution, every dispatch will succeed at runtime
- Note: Requires a whole-program analysis

Alias Analysis

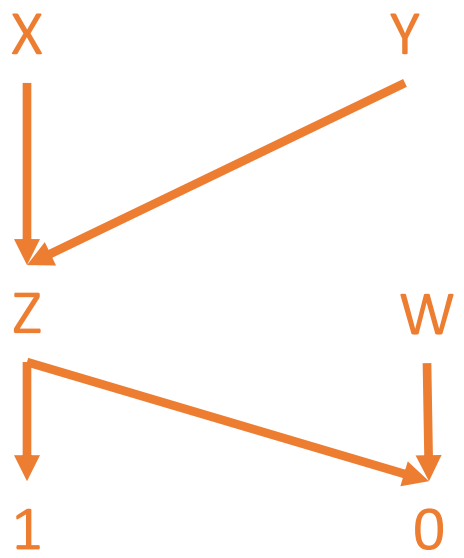
- In languages with side effects, we want to know which locations may have aliases
  - More than one “name”
  - More than one pointer to them

- E.g.,

```
Y = &Z
X = Y
*X = W     // changes the value of *Y
*Y
```

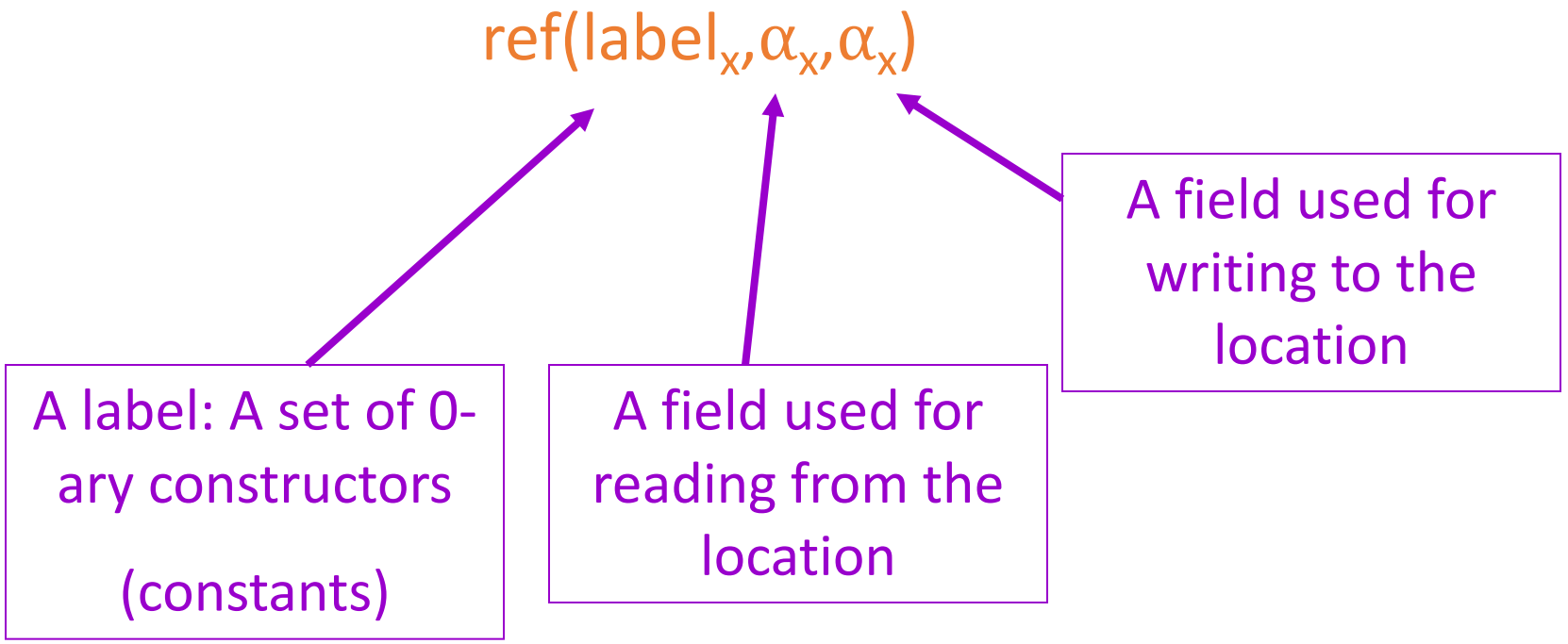
Picture

```
Y = &Z
X = Y
*X = W     // changes the value of *Y
*Y
```



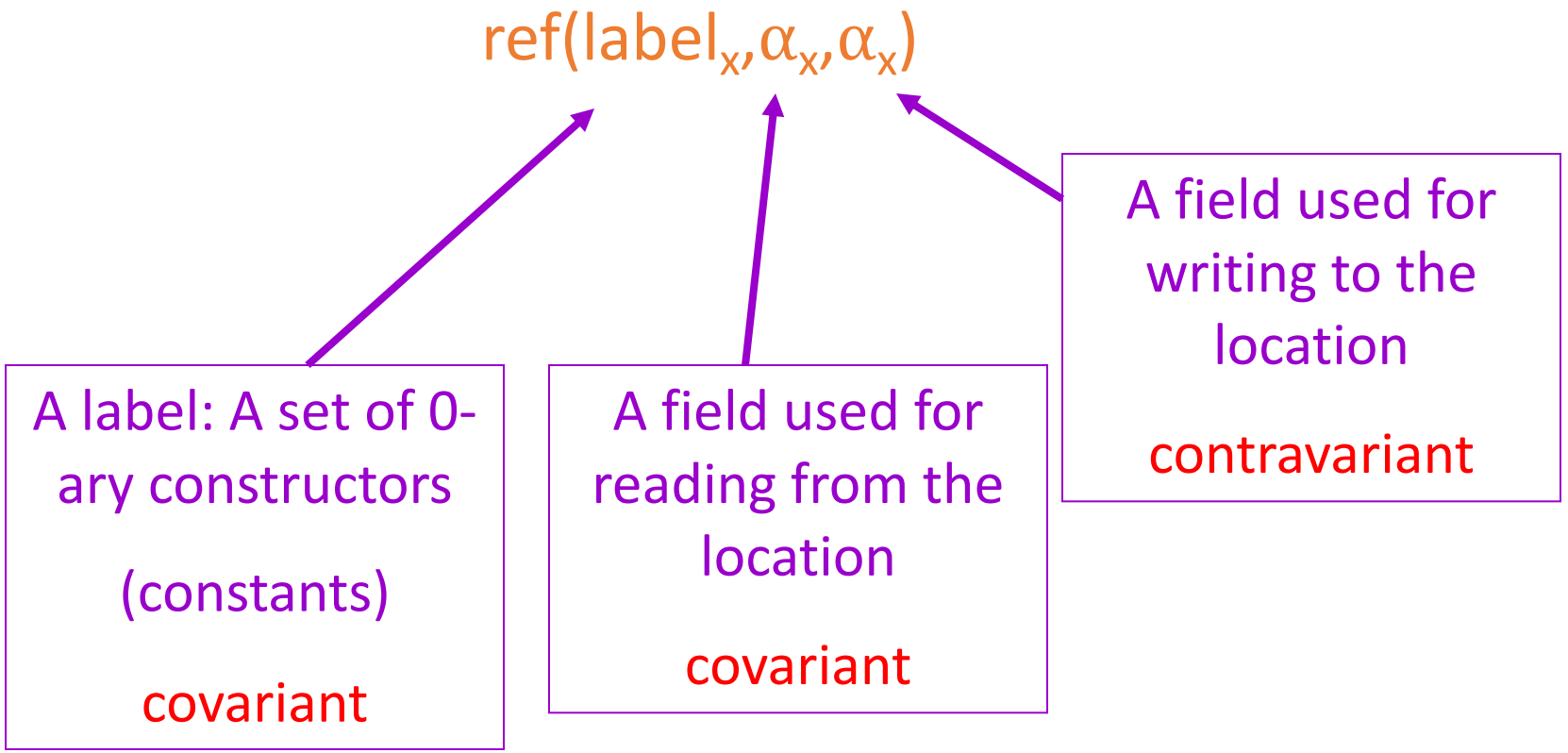
The Encoding of a Location

- For a program variable x:



The Encoding of a Location

- For a program variable x:



$$\mathrm{ref}(l_1, t_1, t_2) \subseteq \mathrm{ref}(l_2, t_3, t_4) \Rightarrow$$
$$\mathrm{ref}(l_1, t_1, t_2) \subseteq \mathrm{ref}(l_2, t_3, t_4), l_1 \subseteq l_2, t_1 \subseteq t_3, t_4 \subseteq t_2$$

A Pointer Language

P ::= S … S

S ::= E = E | E

E ::= *E | &E | x

Inference Rules

$$\frac{}{x : \mathrm{ref}(l_x, \alpha_x, \alpha_x)} \quad [\mathrm{Var}]$$

$$\frac{e : t}{\&e : \mathrm{ref}(0, t, t)} \quad [\mathrm{Ref}]$$

$$\frac{\begin{array}{l} e : t \\ t \subseteq \mathrm{ref}(1, \alpha, 0) \\ \alpha \text{ fresh} \end{array}}{*e : \alpha} \quad [\mathrm{Deref}]$$

$$\frac{\begin{array}{ll} e_1 : t_1 & e_2 : t_2 \\ t_1 \subseteq \mathrm{ref}(1, 1, \alpha) & \alpha \text{ fresh} \\ t_2 \subseteq \mathrm{ref}(1, \beta, 0) & \beta \text{ fresh} \\ \beta \subseteq \alpha & \end{array}}{e_1 = e_2 : t_2} \quad [\mathrm{Assign}]$$

Example

Y = &Z
X = Y
*X = W
*Y

Y : ref(l_y, Y, Y)
X : ref(l_x, X, X)
W: ref(l_w, W, W)
Z: ref(l_z, Z, Z)

&Z : ref(0, ref(l_z, Z, Z), ref(l_z, Z, Z))

Y = &Z :

ref(l_y, Y, Y) ⊆ ref(1,1,A)

ref(0, ref(l_z, Z, Z), ref(l_z, Z, Z)) ⊆ ref(1,B,0)

B ⊆ A

A ⊆ Y

ref(l_z, Z, Z) ⊆ B

ref(l_z, Z, Z) ⊆ Y

X = Y:

ref(l_x, X, X) ⊆ ref(1,1,C)

ref(l_y, Y, Y) ⊆ ref(1,D,0)

D ⊆ C

C ⊆ ref(l_x, X, X)

ref(l_y, Y, Y) ⊆ D

ref(l_y, Y, Y) ⊆ ref(l_x, X, X)

X ⊆ Y, Y ⊆ X

*X:

ref(l_x, X, X) ⊆ ref(1, E, 0)

X ⊆ E

*X = W:

E ⊆ ref(1,1,F)

ref(l_w, W, W) ⊆ ref(1,G,0)

G ⊆ F

ref(l_z, Z, Z) ⊆ ref(1,1,F)

F ⊆ Z

W ⊆ G

W ⊆ Z

*Y:

ref(l_y, Y, Y) ⊆ ref(1, H, 0)

Y ⊆ H

In Practice

- Many natural inclusion-based analysis problems are equivalent to dynamic transitive closure

- Not trivial to implement well
  - $O(n^3)$ suggests it may be slow
  - Many naïve implementations have failed
  - But today these algorithms can scale to millions of lines code

Applications

- Used heavily in compilers, bug-finding tools
  - Reason about data flow in a program
  - Useful for finding opportunities for optimizations
    - E.g., a dynamic dispatch that has exactly one possible target

- Not used in type checking or inference
  - The types are harder to read and interpret

Summary

- Set constraints have many applications in static analysis
  - Closure analysis
  - Receiver class analysis
  - Alias analysis
  - And more ...
- Used often in programming tools