Static Program Analysis

Xiangyu Zhang

The slides are compiled from
Alex Aiken's
Michael D. Ernst's
Sorin Lerner's

A Scary Outline

- Type-based analysis
- Data-flow analysis
- Abstract interpretation
- Theorem proving
- …

The Real Outline

- ❑ The essence of static program analysis
- ❑ The categorization of static program analysis
- ❑ Type-based analysis basics
- ❑ Data-flow analysis basics

The Essence of Static Analysis

- Examine the program text (no execution)
- Build a model of the program state
  - An abstract of the run-time state
- Reason over the possible behaviors.
  - E.g. “run” the program over the abstract state

The Essence of Static Analysis

Each program statement's transfer function indicates how it transforms the (abstract) state

Example: What is the transfer function for

```
y = x++;
```

?

Selecting an abstract domain

Abstract domain: { even, odd, either }

$\langle \text{x is odd; y is odd} \rangle$

```
y = x++;
```

$\langle \text{x is even; y is odd} \rangle$

Selecting an abstract domain

Abstract domain: { even, odd, either }

$\langle$x is odd; y is odd$\rangle$

```
y = x++;
```

$\langle$x is even; y is odd$\rangle$

Abstract domain: { prime, nonprime, anything }

$\langle$x is prime; y is prime$\rangle$

```
y = x++;
```

$\langle$x is anything; y is prime$\rangle$

Selecting an abstract domain

Abstract domain: set of numbers, one set per variable

$\langle \mathrm{x} = \{\ 3, 5, 7\ \}; \mathrm{y} = \{\ 9, 11, 13\ \} \rangle$

```
y = x++;
```

$\langle \mathrm{x} = \{\ 4, 6, 8\ \}; \mathrm{y} = \{\ 3, 5, 7\ \}\ \rangle$

Selecting an abstract domain

Abstract domain: set of numbers, one set per variable

$\langle x = \{ 3, 5, 7 \}; y = \{ 9, 11, 13 \}\rangle$

`y = x++;`

$\langle x = \{ 4, 6, 8 \}; y = \{ 3, 5, 7 \} \rangle$

Abstract domain: set of environments

- environment assigns a variable to a number

$\langle x{=}3,\ y{=}11\rangle, \langle x{=}5,\ y{=}9\rangle, \langle x{=}7,\ y{=}1\rangle$

`y = x++;`

$\langle x{=}4,\ y{=}3\rangle, \langle x{=}6,\ y{=}5\rangle, \langle x{=}8,\ y{=}7\rangle$

Selecting an abstract domain

Abstract domain: set of numbers, one set per variable

$\langle \mathrm{x} = \{ 3, 5, 7 \}; \mathrm{y} = \{ 9, 11, 13 \}\rangle$
`y = x++;`
$\langle \mathrm{x} = \{ 4, 6, 8 \}; \mathrm{y} = \{ 3, 5, 7 \} \rangle$

Abstract domain: set of environments

- environment assigns a variable to a number

$\langle \mathrm{x}{=}3, \mathrm{y}{=}11\rangle, \langle \mathrm{x}{=}5, \mathrm{y}{=}9\rangle, \langle \mathrm{x}{=}7, \mathrm{y}{=}1\rangle$
`y = x++;`
$\langle \mathrm{x}{=}4, \mathrm{y}{=}3\rangle, \langle \mathrm{x}{=}6, \mathrm{y}{=}5\rangle, \langle \mathrm{x}{=}8, \mathrm{y}{=}7\rangle$

Abstract domain: symbolic expression per variable

$\langle x_n = f_1(a_{n-1}, \ldots, z_{n-1}); y_n = f_2(a_{n-1}, \ldots, z_{n-1})\rangle$
`y = x++;`
$\langle x_{n+1} = x_n + 1; y_{n+1} = x_n\rangle$

Categorization

- Flow sensitivity
- Context sensitivity.

Flow Sensitivity

- ❑ Flow sensitive analyses
  - The order of statements matters
  - Need a control flow graph

- ❑ Flow insensitive analyses
  - The order of statements doesn’t matter
  - Analysis is the same regardless of statement order

Example Flow Insensitive Analysis

- What variables does a program modify?

$$G(x := e) = \{x\}$$
$$G(s_1 ; s_2) = G(s_1) \cup G(s_2)$$

- Note $G(s_1;s_2) = G(s_2;s_1)$

The Advantage

- Flow-sensitive analyses require a model of program state at each program point
  - E.g., liveness analysis, reaching definitions, …
- Flow-insensitive analyses require only a single global state
  - E.g., for G, the set of all variables modified

Notes on Flow Sensitivity

- Flow insensitive analyses seem weak, but:

- Flow sensitive analyses are hard to scale to very large programs
  - Additional cost: state size X # of program points

- Beyond 1000's of lines of code, only flow insensitive analyses have been shown to scale (by Alex Aiken)

Context-Sensitive Analysis

- What about analyzing across procedure boundaries?

```
Def f(x){…}
Def g(y){…f(a)…}
Def h(z){…f(b)…}
```

- Goal: Specialize analysis of f to take advantage of
  - f is called with a by g
  - f is called with b by h

Flow Insensitive: Type-Based Analysis

Outline

- A language
  - Lambda calculus
- Types
  - Type checking
  - Type inference
- Applications to software reliability
  - Representation analysis
    - Alias analysis and memory leak analysis.

The Typed Lambda Calculus

- Lambda calculus
  - types are assigned to bound variables.
- Add integers, addition, if-then-else
- Note: Not every expression generated by this grammar is a properly typed term.

$$e = x \mid \lambda x:\tau.e \mid e\ e \mid i \mid e + e \mid \text{if } e\ e\ e$$

Types

- Function types
- Integers
- Type variables
  - Stand for definite, but unknown, types

$$\tau = \alpha \mid \tau \rightarrow \tau \mid \text{int}$$

Function Types

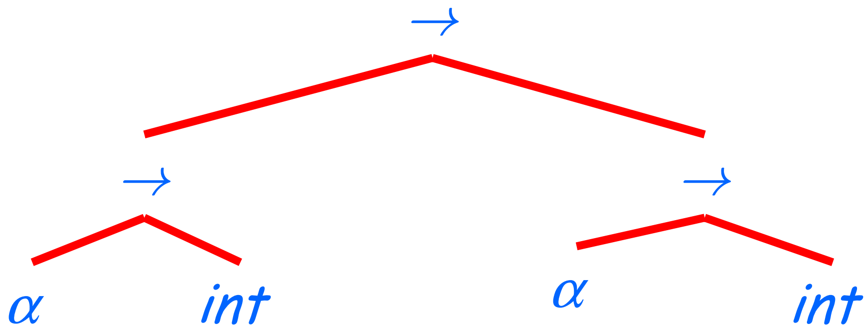
- Intuitively, a type $\tau_1 \rightarrow \tau_2$ stands for the set of functions that map arguments of type τ_1 to results of type τ_2.

- Placeholder for any other structured datatype
  - Lists
  - Trees
  - Arrays

Types are Trees

- Types are terms
- Any term can be represented by a tree
  - The parse tree of the term
  - Tree representation is important in algorithms

$$(\alpha \to \text{int}) \to \alpha \to \text{int}$$



Examples

- We write *e:t* for the statement "*e* has type *t*."

$$\lambda x{:}\alpha x : \alpha \to \alpha$$

Examples

- We write *e:t* for the statement “*e* has type *t*.”

$$\lambda x{:}\alpha.x : \alpha \rightarrow \alpha$$

$$\lambda x{:}\alpha.\lambda y{:}\beta.x : \alpha \rightarrow \beta \rightarrow \alpha$$

Examples

- We write *e:t* for the statement “*e* has type *t*.”

$$\lambda x:\alpha.x:\alpha\to\alpha$$

$$\lambda x:\alpha.\lambda y:\beta.x:\alpha\to\beta\to\alpha$$

$$\lambda f:\alpha\to\beta.\lambda g:\beta\to\gamma.\lambda x:\alpha.g(f\ x):$$

Examples

- We write *e:t* for the statement "*e* has type *t*."

$$\lambda x{:}\alpha.x : \alpha\to\alpha$$

$$\lambda x{:}\alpha.\lambda y{:}\beta.x : \alpha\to\beta\to\alpha$$

$$\lambda f{:}\alpha\to\beta.\lambda g{:}\beta\to\gamma.\lambda x{:}\alpha.g(f\ x) : (\alpha\to\beta)\to(\beta\to\gamma)\to\alpha\to\gamma$$

$$\lambda f{:}\alpha\to\beta\to\gamma.\lambda g{:}\alpha\to\beta.\lambda x{:}\alpha.(f\ x)\ (g\ x) : (\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma$$

Type Environments

- To determine whether the types in an expression are correct we perform *type checking.*
- But we need types for free variables, too!
- A *type environment* is a function from variables to types. The syntax of environments is:

$$A = \emptyset \mid A, x : \tau$$

- The meaning is:

$$(A, x : \tau)(y) = \begin{cases} \tau & \text{if } x = y \\ A(y) & \text{if } x \neq y \end{cases}$$

Type Checking Rules

- ❑ Type checking is done by structural induction.
  - One inference rule for each form
  - Assumptions contain types of free variables
  - A term is *well-typed* if ∅ | e: τ

$$\frac{A(x) = \tau}{A \vdash x : \tau} \qquad \frac{A, x : \tau \vdash e : \tau'}{A \vdash \lambda x : \tau . e : \tau \to \tau'} \qquad \frac{\begin{array}{c} A \vdash e_1 : \tau \to \tau' \\ A \vdash e_2 : \tau \end{array}}{A \vdash e_1 \; e_2 : \tau'}$$

$$\frac{}{A \vdash \mathrm{i} : \mathrm{int}} \qquad \frac{\begin{array}{c} A \vdash e_1 : \mathrm{int} \\ A \vdash e_2 : \mathrm{int} \end{array}}{A \vdash e_1 + e_2 : \mathrm{int}} \qquad \frac{\begin{array}{c} A \vdash e_1 : \mathrm{int} \\ A \vdash e_2 : \tau \\ A \vdash e_3 : \tau \end{array}}{A \vdash \mathrm{if} \; e_1 \; e_2 \; e_3 : \tau}$$

Example

??? $\emptyset \vdash \lambda \boldsymbol{x}:\alpha.\lambda \boldsymbol{y}:\beta.\boldsymbol{x}:\alpha \rightarrow \beta \rightarrow \alpha$

Example

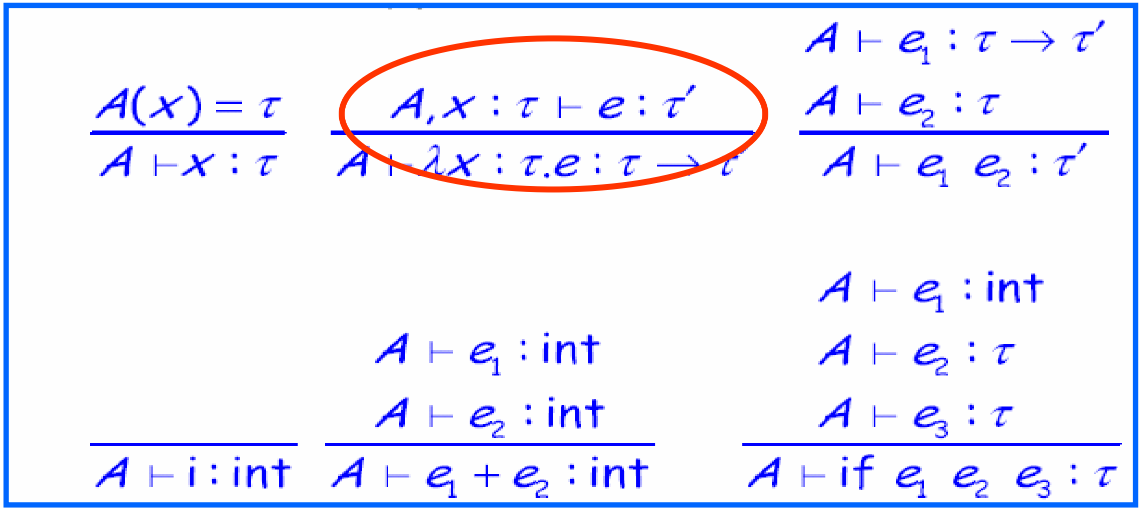
$$x:\alpha, y:\beta \vdash x:\alpha$$

$$\frac{A(x)=\tau}{A \vdash x:\tau} \qquad \frac{A, x:\tau \vdash e:\tau'}{A \vdash \lambda x:\tau.e : \tau \rightarrow \tau'} \qquad \frac{\begin{array}{c} A \vdash e_1 : \tau \rightarrow \tau' \\ A \vdash e_2 : \tau \end{array}}{A \vdash e_1\ e_2 : \tau'}$$

$$\frac{}{A \vdash i : \text{int}} \qquad \frac{\begin{array}{c} A \vdash e_1 : \text{int} \\ A \vdash e_2 : \text{int} \end{array}}{A \vdash e_1 + e_2 : \text{int}} \qquad \frac{\begin{array}{c} A \vdash e_1 : \text{int} \\ A \vdash e_2 : \tau \\ A \vdash e_3 : \tau \end{array}}{A \vdash \text{if } e_1\ e_2\ e_3 : \tau}$$

Example

$$\dfrac{\dfrac{x:\alpha, y:\beta \vdash x:\alpha}{x:\alpha \vdash \lambda y:\beta.x:\beta \to \alpha}}{\quad}$$

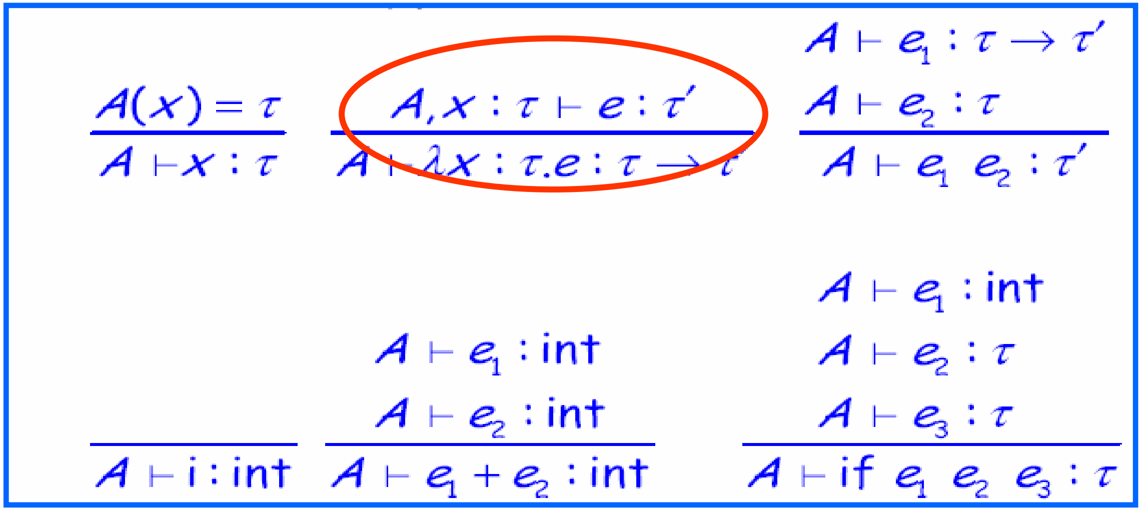


$$\frac{A(x)=\tau}{A \vdash x:\tau} \qquad \frac{A, x:\tau \vdash e:\tau'}{A \vdash \lambda x:\tau.e:\tau \to \tau'} \qquad \frac{\begin{array}{c}A \vdash e_1:\tau \to \tau' \\ A \vdash e_2:\tau\end{array}}{A \vdash e_1\ e_2:\tau'}$$

$$\frac{}{A \vdash i:\text{int}} \qquad \frac{\begin{array}{c}A \vdash e_1:\text{int} \\ A \vdash e_2:\text{int}\end{array}}{A \vdash e_1+e_2:\text{int}} \qquad \frac{\begin{array}{c}A \vdash e_1:\text{int} \\ A \vdash e_2:\tau \\ A \vdash e_3:\tau\end{array}}{A \vdash \text{if } e_1\ e_2\ e_3:\tau}$$

Example

$$\dfrac{\dfrac{x:\alpha, y:\beta \vdash x:\alpha}{x:\alpha \vdash \lambda y:\beta.x:\beta \to \alpha}}{\emptyset \vdash \lambda x:\alpha.\lambda y:\beta.x:\alpha \to \beta \to \alpha}$$

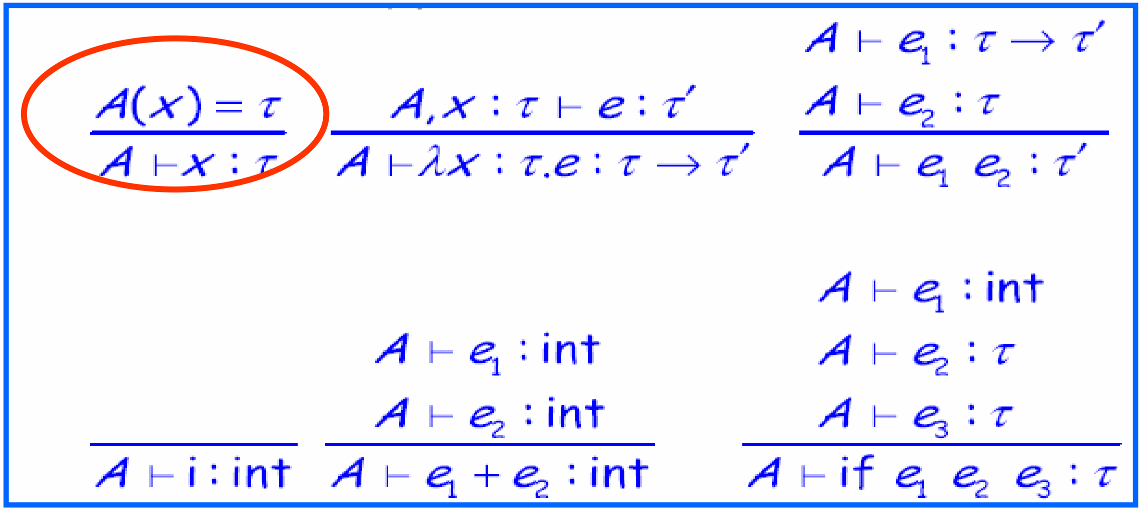


$$\frac{A(x)=\tau}{A \vdash x:\tau} \qquad \frac{A, x:\tau \vdash e:\tau'}{A \vdash \lambda x:\tau.e:\tau \to \tau'} \qquad \frac{\begin{array}{c}A \vdash e_1:\tau \to \tau' \\ A \vdash e_2:\tau\end{array}}{A \vdash e_1\ e_2:\tau'}$$

$$\frac{}{A \vdash i:\mathrm{int}} \qquad \frac{\begin{array}{c}A \vdash e_1:\mathrm{int} \\ A \vdash e_2:\mathrm{int}\end{array}}{A \vdash e_1 + e_2:\mathrm{int}} \qquad \frac{\begin{array}{c}A \vdash e_1:\mathrm{int} \\ A \vdash e_2:\tau \\ A \vdash e_3:\tau\end{array}}{A \vdash \mathrm{if}\ e_1\ e_2\ e_3:\tau}$$

Not Straightforward

$$x:\alpha, y:\beta \vdash x:\alpha$$



$$\frac{A(x)=\tau}{A \vdash x:\tau} \qquad \frac{A, x:\tau \vdash e:\tau'}{A \vdash \lambda x:\tau.e:\tau \to \tau'} \qquad \frac{\begin{array}{c} A \vdash e_1:\tau \to \tau' \\ A \vdash e_2:\tau \end{array}}{A \vdash e_1\ e_2:\tau'}$$

$$\frac{}{A \vdash i:\mathrm{int}} \qquad \frac{\begin{array}{c} A \vdash e_1:\mathrm{int} \\ A \vdash e_2:\mathrm{int} \end{array}}{A \vdash e_1 + e_2:\mathrm{int}} \qquad \frac{\begin{array}{c} A \vdash e_1:\mathrm{int} \\ A \vdash e_2:\tau \\ A \vdash e_3:\tau \end{array}}{A \vdash \mathrm{if}\ e_1\ e_2\ e_3:\tau}$$

Type Checking Algorithm

- There is a simple algorithm for type checking
- Observe that there is only one possible “shape” of the type derivation
  - only one inference rule applies to each form.

$$\dfrac{\dfrac{? \vdash x : ?}{? \vdash \lambda y : \beta . x : ?}}{\emptyset \vdash \lambda x : \alpha . \lambda y : \beta . x : ?}$$

Algorithm (Cont.)

- Walk the proof tree from the root to the leaves, generating the correct environments.
- Assumptions are simply gathered from lambda abstractions.

$$\dfrac{\dfrac{x:\alpha, y:\beta \vdash x : ?}{x:\alpha \vdash \lambda y:\beta.x : ?}}{\emptyset \vdash \lambda x:\alpha.\lambda y:\beta.x : ?}$$

Algorithm (Cont.)

- In a walk from the leaves to the root, calculate the type of each expression.
- The types are completely determined by the type environment and the types of subexpressions.

$$\dfrac{\dfrac{x:\alpha, y:\beta \vdash x:\alpha}{x:\alpha \vdash \lambda y:\beta.x:\beta \to \alpha}}{\emptyset \vdash \lambda x:\alpha.\lambda y:\beta.x:\alpha \to \beta \to \alpha}$$

A Bigger Example

$$\dfrac{\dfrac{\dfrac{x:\alpha\to\alpha, y:\beta \vdash x:\alpha\to\alpha}{x:\alpha\to\alpha \vdash \lambda y:\beta.x:\beta\to\alpha\to\alpha}}{\emptyset \vdash \lambda x:\alpha\to\alpha.\lambda y:\beta.x:(\alpha\to\alpha)\to\beta\to\alpha\to\alpha} \quad \dfrac{z:\alpha \vdash z:\alpha}{\emptyset \vdash \lambda z:\alpha.z:\alpha\to\alpha}}{\emptyset \vdash (\lambda x:\alpha\to\alpha.\lambda y:\beta.x)\ \lambda z:\alpha.z:(\alpha\to\alpha)\to\beta\to\alpha\to\alpha}$$

What Do Types Mean?

- **Thm.** If $A \vdash e:\tau$ and $e \rightarrow^{*}_{\beta} d$, then $A \vdash d:\tau$
  - Evaluation preserves types.

- This is the basis of a claim that there can be no runtime type errors
  - functions applied to data of the wrong type
    - Adding to a function
    - Using an integer as a function

Type Inference

- The *type erasure* of *e* is *e* with all type information removed (i.e., the untyped term).
- Is an untyped term the erasure of some simply typed term? And what are the types?
- This is a *type inference* problem. We must infer, rather than check, the types.

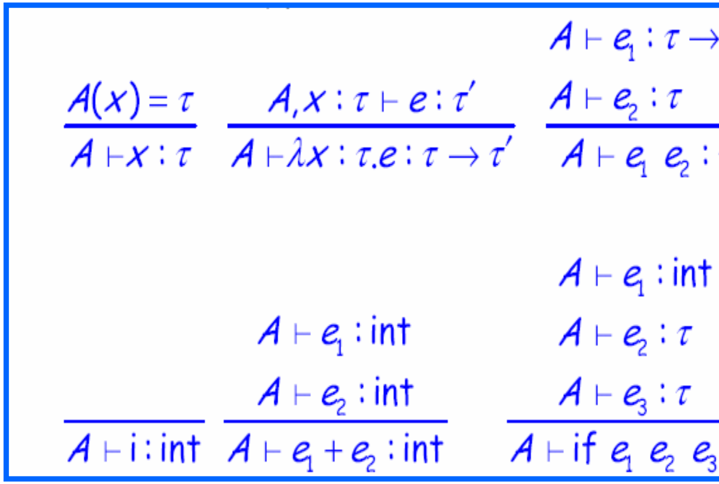
Type Inference

- recast the type rules in an equivalent form
- typing in the new rules reduces to a constraint satisfaction problem
- the constraint problem is solvable via term unification.

New Rules

$$\frac{A(x)=\alpha_x}{A\vdash x:\alpha_x} \qquad \frac{A,x:\alpha_x\vdash e:\tau}{A\vdash \lambda x.e:\alpha_x\to\tau} \qquad \frac{\begin{array}{c}A\vdash e_1:\tau_1\\ A\vdash e_2:\tau_2\\ \tau_1=\tau_2\to\beta\end{array}}{A\vdash e_1\ e_2:\beta}$$

$$\frac{}{A\vdash \mathrm{i}:\mathrm{int}} \qquad \frac{\begin{array}{c}A\vdash e_1:\tau_1\\ A\vdash e_2:\tau_2\\ \tau_1=\tau_2=\mathrm{int}\end{array}}{A\vdash e_1+e_2:\mathrm{int}} \qquad \frac{\begin{array}{c}A\vdash e_1:\tau_1\\ A\vdash e_2:\tau_2\\ A\vdash e_3:\tau_3\\ \tau_1=\mathrm{int}\quad \tau_2=\tau_3\end{array}}{A\vdash \mathrm{if}\ e_1\ e_2\ e_3:\tau_2}$$



$$\frac{A(x)=\tau}{A\vdash x:\tau} \qquad \frac{A,x:\tau\vdash e:\tau'}{A\vdash \lambda x:\tau.e:\tau\to\tau'} \qquad \frac{\begin{array}{c}A\vdash e_1:\tau\to\\ A\vdash e_2:\tau\end{array}}{A\vdash e_1\ e_2:}$$

$$\frac{}{A\vdash \mathrm{i}:\mathrm{int}} \qquad \frac{\begin{array}{c}A\vdash e_1:\mathrm{int}\\ A\vdash e_2:\mathrm{int}\end{array}}{A\vdash e_1+e_2:\mathrm{int}} \qquad \frac{\begin{array}{c}A\vdash e_1:\mathrm{int}\\ A\vdash e_2:\tau\\ A\vdash e_3:\tau\end{array}}{A\vdash \mathrm{if}\ e_1\ e_2\ e_3}$$

- Sidestep the problems by introducing explicit unknowns and constraints

New Rules

$$\frac{A(x)=\alpha_x}{A \vdash x : \alpha_x} \qquad \frac{A, x : \alpha_x \vdash e : \tau}{A \vdash \lambda x.e : \alpha_x \to \tau} \qquad \frac{\begin{array}{c} A \vdash e_1 : \tau_1 \\ A \vdash e_2 : \tau_2 \\ \tau_1 = \tau_2 \to \beta \end{array}}{A \vdash e_1\ e_2 : \beta}$$

$$\frac{}{A \vdash i : \text{int}} \qquad \frac{\begin{array}{c} A \vdash e_1 : \tau_1 \\ A \vdash e_2 : \tau_2 \\ \tau_1 = \tau_2 = \text{int} \end{array}}{A \vdash e_1 + e_2 : \text{int}} \qquad \frac{\begin{array}{c} A \vdash e_1 : \tau_1 \\ A \vdash e_2 : \tau_2 \\ A \vdash e_3 : \tau_3 \\ \tau_1 = \text{int} \quad \tau_2 = \tau_3 \end{array}}{A \vdash \text{if } e_1\ e_2\ e_3 : \tau_2}$$

$$\frac{A(x)=\tau}{A \vdash x : \tau} \qquad \frac{A, x : \tau \vdash e : \tau'}{A \vdash \lambda x : \tau.e : \tau \to \tau'} \qquad \frac{\begin{array}{c} A \vdash e_1 : \tau \to \tau' \\ A \vdash e_2 : \tau \end{array}}{A \vdash e_1\ e_2 : \tau'}$$

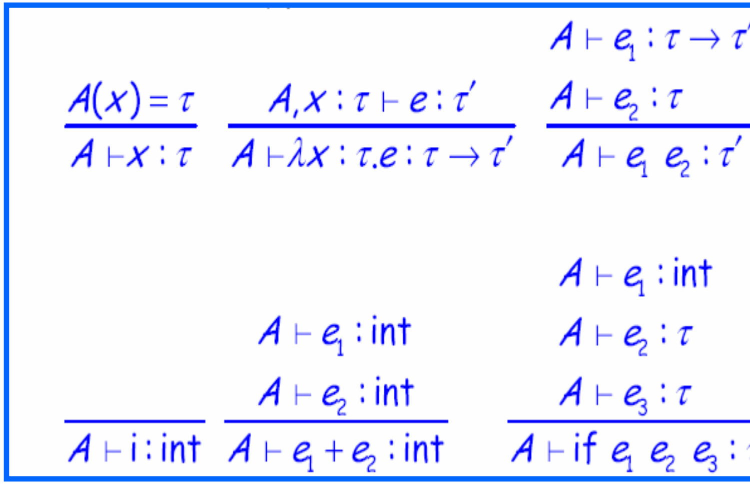
$$\frac{}{A \vdash i : \text{int}} \qquad \frac{\begin{array}{c} A \vdash e_1 : \text{int} \\ A \vdash e_2 : \text{int} \end{array}}{A \vdash e_1 + e_2 : \text{int}} \qquad \frac{\begin{array}{c} A \vdash e_1 : \text{int} \\ A \vdash e_2 : \tau \\ A \vdash e_3 : \tau \end{array}}{A \vdash \text{if } e_1\ e_2\ e_3 :}$$

- Type assumption for variable x is a fresh variable α_x

New Rules

$$\frac{A(x)=\alpha_x}{A \vdash x : \alpha_x} \qquad \frac{A, x:\alpha_x \vdash e:\tau}{A \vdash \lambda x.e : \alpha_x \to \tau} \qquad \frac{\begin{array}{c} A \vdash e_1 : \tau_1 \\ A \vdash e_2 : \tau_2 \\ \tau_1 = \tau_2 \to \beta \end{array}}{A \vdash e_1\ e_2 : \beta}$$

$$\frac{}{A \vdash i : \text{int}} \qquad \frac{\begin{array}{c} A \vdash e_1 : \tau_1 \\ A \vdash e_2 : \tau_2 \\ \tau_1 = \tau_2 = \text{int} \end{array}}{A \vdash e_1 + e_2 : \text{int}} \qquad \frac{\begin{array}{c} A \vdash e_1 : \tau_1 \\ A \vdash e_2 : \tau_2 \\ A \vdash e_3 : \tau_3 \\ \tau_1 = \text{int} \quad \tau_2 = \tau_3 \end{array}}{A \vdash \text{if } e_1\ e_2\ e_3 : \tau_2}$$



$$\frac{A(x)=\tau}{A \vdash x : \tau} \qquad \frac{A, x:\tau \vdash e:\tau'}{A \vdash \lambda x:\tau.e : \tau \to \tau'} \qquad \frac{\begin{array}{c} A \vdash e_1 : \tau \to \tau' \\ A \vdash e_2 : \tau \end{array}}{A \vdash e_1\ e_2 : \tau'}$$

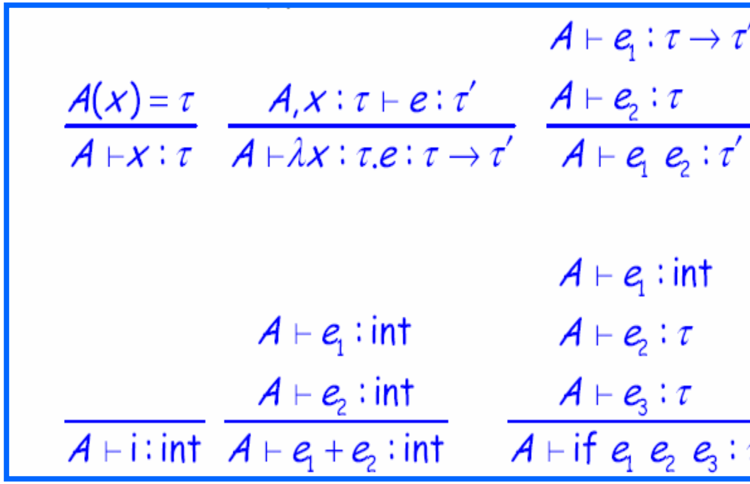
$$\frac{}{A \vdash i : \text{int}} \qquad \frac{\begin{array}{c} A \vdash e_1 : \text{int} \\ A \vdash e_2 : \text{int} \end{array}}{A \vdash e_1 + e_2 : \text{int}} \qquad \frac{\begin{array}{c} A \vdash e_1 : \text{int} \\ A \vdash e_2 : \tau \\ A \vdash e_3 : \tau \end{array}}{A \vdash \text{if } e_1\ e_2\ e_3 :}$$

- Hypotheses are all arbitrary
  - Can always complete a derivation, pending constraint resolution

New Rules

$$\frac{x) = \alpha_x}{\vdash x : \alpha_x} \qquad \frac{A, x : \alpha_x \vdash e : \tau}{A \vdash \lambda x.e : \alpha_x \to \tau} \qquad \frac{\begin{array}{c} A \vdash e_1 : \tau_1 \\ A \vdash e_2 : \tau_2 \\ \boxed{\tau_1 = \tau_2 \to \beta} \end{array}}{A \vdash e_1\ e_2 : \beta}$$

$$\frac{}{\vdash i : \text{int}} \qquad \frac{\begin{array}{c} A \vdash e_1 : \tau_1 \\ A \vdash e_2 : \tau_2 \\ \boxed{\tau_1 = \tau_2 = \text{int}} \end{array}}{A \vdash e_1 + e_2 : \text{int}} \qquad \frac{\begin{array}{c} A \vdash e_1 : \tau_1 \\ A \vdash e_2 : \tau_2 \\ A \vdash e_3 : \tau_3 \\ \boxed{\tau_1 = \text{int} \quad \tau_2 = \tau_3} \end{array}}{A \vdash \text{if } e_1\ e_2\ e_3 : \tau_2}$$



$$\frac{A(x) = \tau}{A \vdash x : \tau} \qquad \frac{A, x : \tau \vdash e : \tau'}{A \vdash \lambda x : \tau.e : \tau \to \tau'} \qquad \frac{\begin{array}{c} A \vdash e_1 : \tau \to \tau' \\ A \vdash e_2 : \tau \end{array}}{A \vdash e_1\ e_2 : \tau'}$$

$$\frac{}{A \vdash i : \text{int}} \qquad \frac{\begin{array}{c} A \vdash e_1 : \text{int} \\ A \vdash e_2 : \text{int} \end{array}}{A \vdash e_1 + e_2 : \text{int}} \qquad \frac{\begin{array}{c} A \vdash e_1 : \text{int} \\ A \vdash e_2 : \tau \\ A \vdash e_3 : \tau \end{array}}{A \vdash \text{if } e_1\ e_2\ e_3 :}$$

- ❑ Equality conditions represented as side constraints

Solutions of Constraints

- The new rules generate a system of type equations.
- Intuitively, a solution of these equations gives a derivation.
- A solution is a substitution Vars → Types such that the equations are satisfied.

Example

$$\alpha = \beta \to \gamma$$
$$\alpha = \gamma \to \beta$$
$$\beta = \mathsf{int}$$

- A solution is

$$\alpha = \mathsf{int} \to \mathsf{int},\ \beta = \mathsf{int},\ \gamma = \mathsf{int}$$

Solving Type Equations

- Term equations are a unification problem.
  - Solvable in near-linear time using a union-find based algorithm.
- No solutions $\alpha = T[\alpha]$ are permitted
  - The *occurs check.*
  - The check is omitted if we allow infinite types.

Unification

- Four rules.
- If no inconsistency or occurs check violation found, system has a solution.
  - int = x → y

$$
\begin{aligned}
S \cup \{\alpha = \alpha\} &\Rightarrow S \\
S \cup \{\alpha = \tau\} &\Rightarrow S[\tau/\alpha] \cup \{\alpha \cong \tau\} \\
S \cup \{\tau_1 \rightarrow \tau_2 = \tau_3 \rightarrow \tau_4\} &\Rightarrow S \cup \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \\
S \cup \{\text{int} = \text{int}\} &\Rightarrow S
\end{aligned}
$$

Syntax

- We distinguish *solved* equations $\alpha \cong \tau$
- Each rule manipulates only unsolved equations.

$$
\begin{array}{lcl}
S \cup \{\alpha = \alpha\} & \Rightarrow & S \\
S \cup \{\alpha = \tau\} & \Rightarrow & S[\tau/\alpha] \cup \{\alpha \cong \tau\} \\
S \cup \{\tau_1 \rightarrow \tau_2 = \tau_3 \rightarrow \tau_4\} & \Rightarrow & S \cup \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \\
S \cup \{\mathrm{int} = \mathrm{int}\} & \Rightarrow & S
\end{array}
$$

Rules 1 and 4

- Rules 1 and 4 eliminate trivial constraints.
- Rule 1 is applied in preference to rule 2
  - the only such possible conflict

$$S \cup \{\alpha = \alpha\} \Rightarrow S$$

$$S \cup \{\alpha = \tau\} \Rightarrow S[\tau/\alpha] \cup \{\alpha \cong \tau\}$$

$$S \cup \{\tau_1 \to \tau_2 = \tau_3 \to \tau_4\} \Rightarrow S \cup \{\tau_1 = \tau_3, \tau_2 = \tau_4\}$$

$$S \cup \{\text{int} = \text{int}\} \Rightarrow S$$

Rule 2

- Rule 2 eliminates a variable from all equations but one (which is marked as solved).
  - Note the variable is eliminated from all unsolved as well as solved equations

$$\begin{array}{lll}
S \cup \{\alpha = \alpha\} & \Rightarrow & S \\
S \cup \{\alpha = \tau\} & \Rightarrow & S[\tau/\alpha] \cup \{\alpha \doteqdot \tau\} \\
S \cup \{\tau_1 \to \tau_2 = \tau_3 \to \tau_4\} & \Rightarrow & S \cup \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \\
S \cup \{\mathrm{int} = \mathrm{int}\} & \Rightarrow & S
\end{array}$$

Rule 3

- Rule 3 applies structural equality to non-trivial terms.
- Note rule 4 is a degenerate case of rule 3 for a type constructor of arity zero.

$$
\begin{aligned}
S \cup \{\alpha = \alpha\} &\quad\Rightarrow\quad S \\
S \cup \{\alpha = \tau\} &\quad\Rightarrow\quad S[\tau/\alpha] \cup \{\alpha \cong \tau\} \\
S \cup \{\tau_1 \rightarrow \tau_2 = \tau_3 \rightarrow \tau_4\} &\quad\Rightarrow\quad S \cup \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \\
S \cup \{\mathrm{int} = \mathrm{int}\} &\quad\Rightarrow\quad S
\end{aligned}
$$

Correctness

- Each rule preserves the set of solutions.
  - Rules 1 and 4 eliminate trivial constraints.
  - Rule 2 substitutes equals for equals.
  - Rule 3 is the definition of equality on function types.

$$\begin{aligned}
S \cup \{\alpha = \alpha\} &\Rightarrow S \\
S \cup \{\alpha = \tau\} &\Rightarrow S[\tau/\alpha] \cup \{\alpha \cong \tau\} \\
S \cup \{\tau_1 \rightarrow \tau_2 = \tau_3 \rightarrow \tau_4\} &\Rightarrow S \cup \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \\
S \cup \{\mathrm{int} = \mathrm{int}\} &\Rightarrow S
\end{aligned}$$

Termination

- Rules 1 and 4 reduce the number of equations.
- Rule 2 reduces the number of variables in unsolved equations.
- Rule 3 decreases the height of terms.

$$
\begin{array}{lll}
S \cup \{\alpha = \alpha\} & \Rightarrow & S \\
S \cup \{\alpha = \tau\} & \Rightarrow & S[\tau/\alpha] \cup \{\alpha \cong \tau\} \\
S \cup \{\tau_1 \rightarrow \tau_2 = \tau_3 \rightarrow \tau_4\} & \Rightarrow & S \cup \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \\
S \cup \{\mathrm{int} = \mathrm{int}\} & \Rightarrow & S
\end{array}
$$

Termination (Cont.)

- Rules 1, 3, and 4 always terminate
  - because terms must eventually be reduced to height 0.
- Eventually rule 2 is applied, reducing the number of variables.

$$
\begin{array}{lcl}
S \cup \{\alpha = \alpha\} & \Rightarrow & S \\
S \cup \{\alpha = \tau\} & \Rightarrow & S[\tau/\alpha] \cup \{\alpha \cong \tau\} \\
S \cup \{\tau_1 \to \tau_2 = \tau_3 \to \tau_4\} & \Rightarrow & S \cup \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \\
S \cup \{\mathrm{int} = \mathrm{int}\} & \Rightarrow & S
\end{array}
$$

A Nitpick

- We really need one more operation.
- $\tau = \alpha$ should be flipped to $\alpha = \tau$ if τ is not a variable.
  - Needed to ensure rule 2 applies whenever possible.
  - We just assume equations are maintained in this “normal form”.

Solutions

- The final system is a solution.
  - There is one equation $\alpha \cong \tau$ for each variable.
  - This is a substitution with all the solutions of the original system
- Must also perform occurs check to guarantee there are no recursive constraints.

Example

ewrites

$$\alpha = \beta \rightarrow \gamma,\ \alpha = \gamma \rightarrow \beta,\ \beta = \text{int}$$

Example

ewrites

$$\frac{\alpha = \beta \rightarrow \gamma,\ \alpha = \gamma \rightarrow \beta,\ \beta = \mathsf{int}}{\alpha = \mathsf{int} \rightarrow \gamma,\ \alpha = \gamma \rightarrow \mathsf{int},\ \beta \cong \mathsf{int}}$$

Example

ewrites

$$\begin{array}{c} \alpha = \beta \to \gamma,\ \alpha = \gamma \to \beta,\ \beta = \mathsf{int} \\ \hline \alpha = \mathsf{int} \to \gamma,\ \alpha = \gamma \to \mathsf{int},\ \beta \cong \mathsf{int} \\ \hline \gamma \to \mathsf{int} = \mathsf{int} \to \gamma,\ \alpha \cong \gamma \to \mathsf{int},\ \beta \cong \mathsf{int} \end{array}$$

Example

ewrites

$$\begin{array}{c}
\alpha = \beta \rightarrow \gamma,\ \alpha = \gamma \rightarrow \beta,\ \beta = \mathsf{int} \\
\hline
\alpha = \mathsf{int} \rightarrow \gamma,\ \alpha = \gamma \rightarrow \mathsf{int},\ \beta \cong \mathsf{int} \\
\hline
\gamma \rightarrow \mathsf{int} = \mathsf{int} \rightarrow \gamma,\ \alpha \cong \gamma \rightarrow \mathsf{int},\ \beta \cong \mathsf{int} \\
\hline
\gamma = \mathsf{int},\ \mathsf{int} = \gamma,\ \alpha \cong \gamma \rightarrow \mathsf{int},\ \beta \cong \mathsf{int}
\end{array}$$

Example

ewrites

$$\begin{array}{c}
\alpha = \beta \rightarrow \gamma,\ \alpha = \gamma \rightarrow \beta,\ \beta = \mathsf{int} \\
\hline
\alpha = \mathsf{int} \rightarrow \gamma,\ \alpha = \gamma \rightarrow \mathsf{int},\ \beta \cong \mathsf{int} \\
\hline
\gamma \rightarrow \mathsf{int} = \mathsf{int} \rightarrow \gamma,\ \alpha \cong \gamma \rightarrow \mathsf{int},\ \beta \cong \mathsf{int} \\
\hline
\gamma = \mathsf{int},\ \mathsf{int} = \gamma,\ \alpha \cong \gamma \rightarrow \mathsf{int},\ \beta \cong \mathsf{int} \\
\hline
\mathsf{int} = \mathsf{int},\ \gamma \cong \mathsf{int},\ \alpha \cong \mathsf{int} \rightarrow \mathsf{int},\ \beta \cong \mathsf{int}
\end{array}$$

Example

ewrites

$$\begin{array}{c}
\alpha = \beta \rightarrow \gamma,\ \alpha = \gamma \rightarrow \beta,\ \beta = \mathsf{int} \\ \hline
\alpha = \mathsf{int} \rightarrow \gamma,\ \alpha = \gamma \rightarrow \mathsf{int},\ \beta \cong \mathsf{int} \\ \hline
\gamma \rightarrow \mathsf{int} = \mathsf{int} \rightarrow \gamma,\ \alpha \cong \gamma \rightarrow \mathsf{int},\ \beta \cong \mathsf{int} \\ \hline
\gamma = \mathsf{int},\ \mathsf{int} = \gamma,\ \alpha \cong \gamma \rightarrow \mathsf{int},\ \beta \cong \mathsf{int} \\ \hline
\mathsf{int} = \mathsf{int},\ \gamma \cong \mathsf{int},\ \alpha \cong \mathsf{int} \rightarrow \mathsf{int},\ \beta \cong \mathsf{int} \\ \hline
\gamma \cong \mathsf{int},\ \alpha \cong \mathsf{int} \rightarrow \mathsf{int},\ \beta \cong \mathsf{int}
\end{array}$$

An Example of Failure

$$\begin{array}{c}
\alpha = \beta \to \gamma,\ \alpha = \gamma \to (\beta \to \beta),\ \beta = \mathsf{int} \\
\hline
\alpha = \mathsf{int} \to \gamma,\ \alpha = \gamma \to (\mathsf{int} \to \mathsf{int}),\ \beta \cong \mathsf{int} \\
\hline
\gamma \to (\mathsf{int} \to \mathsf{int}) = \mathsf{int} \to \gamma,\ \alpha \cong \gamma \to \mathsf{int},\ \beta \cong \mathsf{int} \\
\hline
\gamma = \mathsf{int},\ \mathsf{int} \to \mathsf{int} = \gamma,\ \alpha \cong \gamma \to \mathsf{int},\ \beta \cong \mathsf{int} \\
\hline
\mathsf{t} \to \mathsf{int} = \mathsf{int},\ \gamma \cong \mathsf{int} \to \mathsf{int},\ \alpha \cong \mathsf{int} \to \mathsf{int},\ \beta \cong \mathsf{i}
\end{array}$$

Notes

- The algorithm produces the *most general unifier* of the equations.
  - All solutions are preserved.

- Less general solutions are all substitution instances of the most general solution.

- There exists more efficient algorithm, amortized time complexity is close to linear

Application – Treating Program Property as A Type

- INT, BOOL, and STRING are types, and
  - “ALLOCATED” and “FREED” can also be treated as types.

$$\frac{e_1 : \tau \qquad e_2 : \tau}{e_1 = e_2 : \tau}$$

For example, p=q

Uses

- Find bugs
  - Every equivalence class with a *malloc* should have a *free*
- Alias analysis
- Implemented for C in a tool Lackwit
  - O'Callahan & Jackson

Where is Type Inference Strong?

- Handles data structures smoothly
- Works in infinite domains
  - Set of types is unlimited
- No forwards/backwards distinction
- Type polymorphism good fit for context sensitivity

Where is Type Inference Weak?

- No flow sensitivity
  - Equality-based analysis only gets equivalence classes

- Context-sensitive analyses don’t always scale
  - Type polymorphism can lead to exponential blowup in constraints

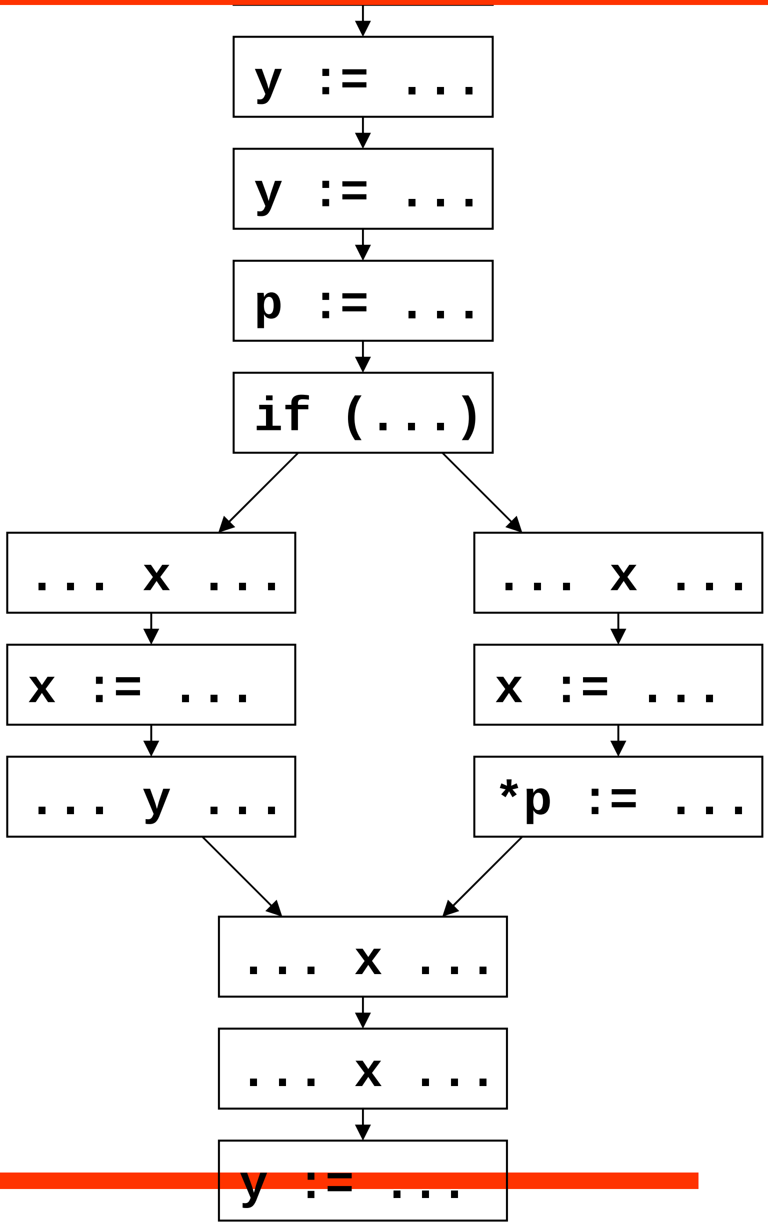
Flow Sensitive: Data Flow Analysis

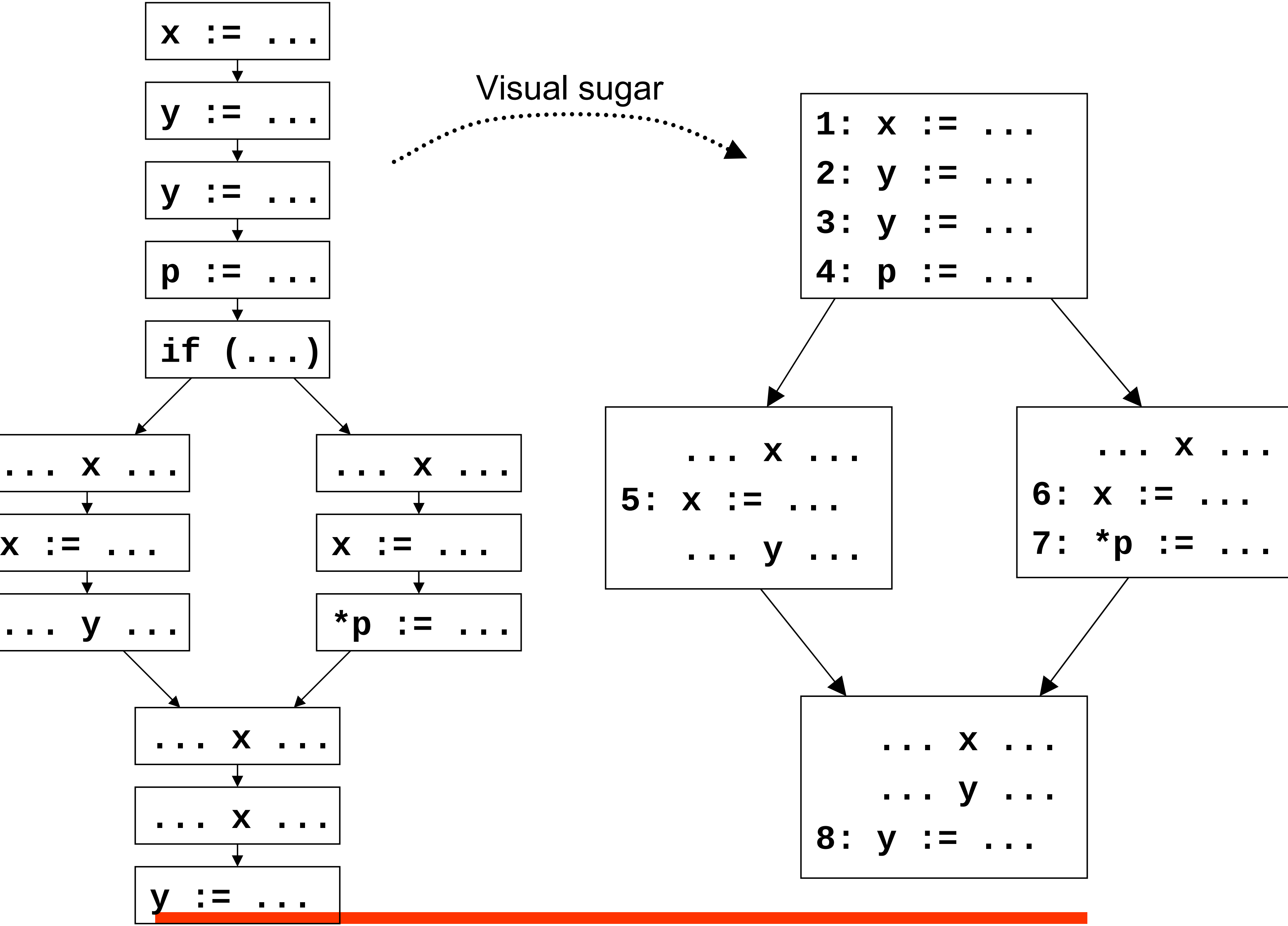
An example DFA: reaching definitions

- For each use of a variable, determine what assignments could have set the value being read from the variable
- Information useful for:
  - performing constant and copy prop
  - detecting references to undefined variables
  - presenting “def/use chains” to the programmer
  - building other representations, like the program dependence graph
- Let’s try this out on an example

Example CFG

```
x := ...
y := ...
y := ...
p := ...
if (...) {
   ... x ...
   x := ...
   ... y ...
}
else {
   ... x ...
   x := ...
   *p := ...
}
... x ...
... y ...
y := ...
```





```
x := ...
y := ...
y := ...
p := ...
if (...)
... x ...
x := ...
... y ...
... x ...
x := ...
*p := ...
... x ...
... x ...
y := ...
```

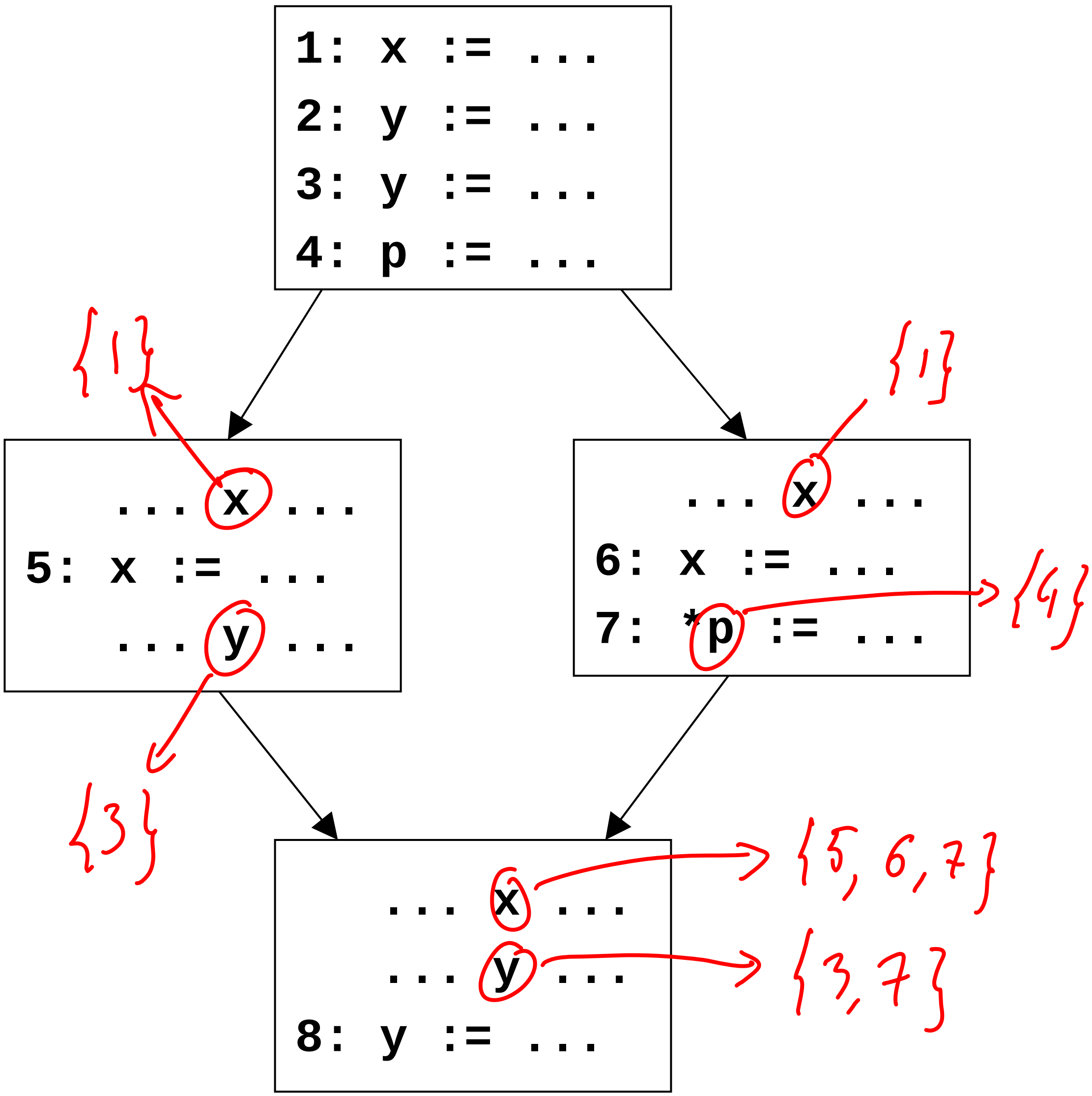
Visual sugar

```
1: x := ...
2: y := ...
3: y := ...
4: p := ...
```

```
   ... x ...
5: x := ...
   ... y ...
```

```
   ... x ...
6: x := ...
7: *p := ...
```

```
   ... x ...
   ... y ...
8: y := ...
```



```
1: x := ...
2: y := ...
3: y := ...
4: p := ...
```

```
   ... x ...
5: x := ...
   ... y ...
```

```
   ... x ...
6: x := ...
7: *p := ...
```

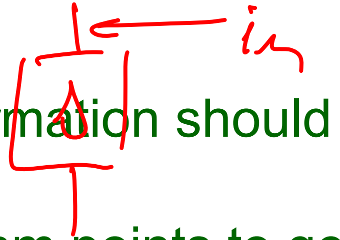
```
   ... x ...
   ... y ...
8: y := ...
```

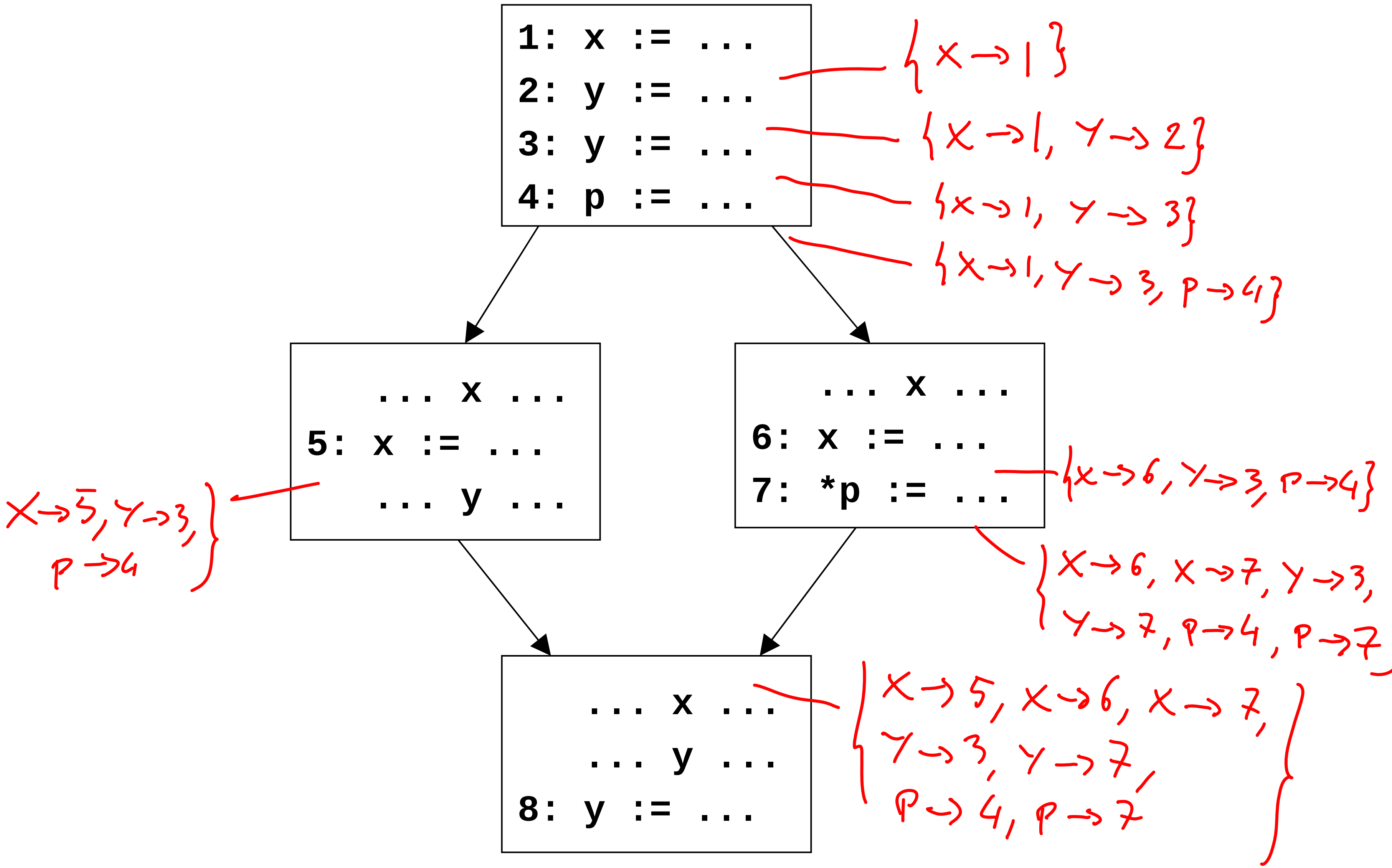
Safety

- Safety:
  - can have more bindings than the “true” answer, but can’t miss any

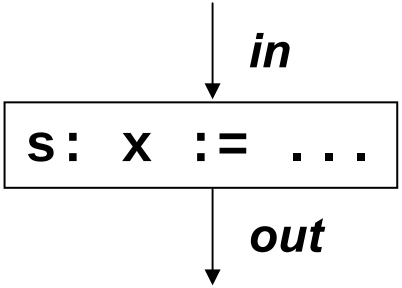
Reaching definitions generalized

- Computed information at a program point is a set of var → stmt bindings
    - eg: $\{ x \rightarrow s_1, x \rightarrow s_2, y \rightarrow s_3 \}$
- How do we get the previous info we wanted?
    - if a var x is used in a stmt whose incoming info is *in*, then: $\{ s \mid (x \rightarrow s) \in in \}$
- This is a common pattern
    - generalize the problem to define what information should be computed at each program point
    - use the computed information at the program points to get the original info we wanted

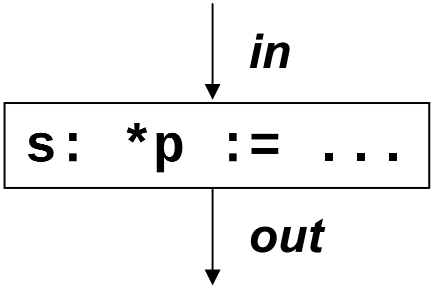




Constraints for reaching definitions

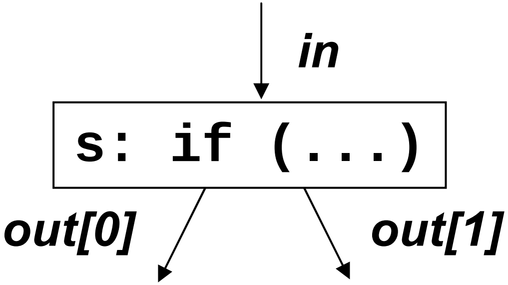


$$out = in - \{ x \rightarrow s' \mid s' \in stmts \} \cup \{ x \rightarrow s \}$$



$$\begin{aligned} out = in - \{ x \rightarrow s' \mid x \in \text{must-point-to}(p) \ \wedge \\ s' \in stmts \} \\ \cup \{ x \rightarrow s \mid x \in \text{may-point-to}(p) \} \end{aligned}$$

Constraints for reaching definitions



in

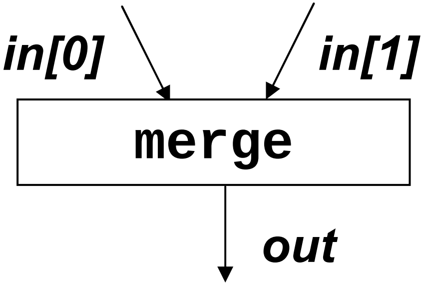
`s: if (...)`

out[0] ***out[1]***

$out\,[\,0\,] = in \wedge$

$out\,[\,0\,] = in$

more generally: $\forall\, i\, .\, out\,[\,i\,] = in$



in[0] ***in[1]***

`merge`

out

$out = in\,[\,0\,] \,\cup\, in\,[\,1\,]$

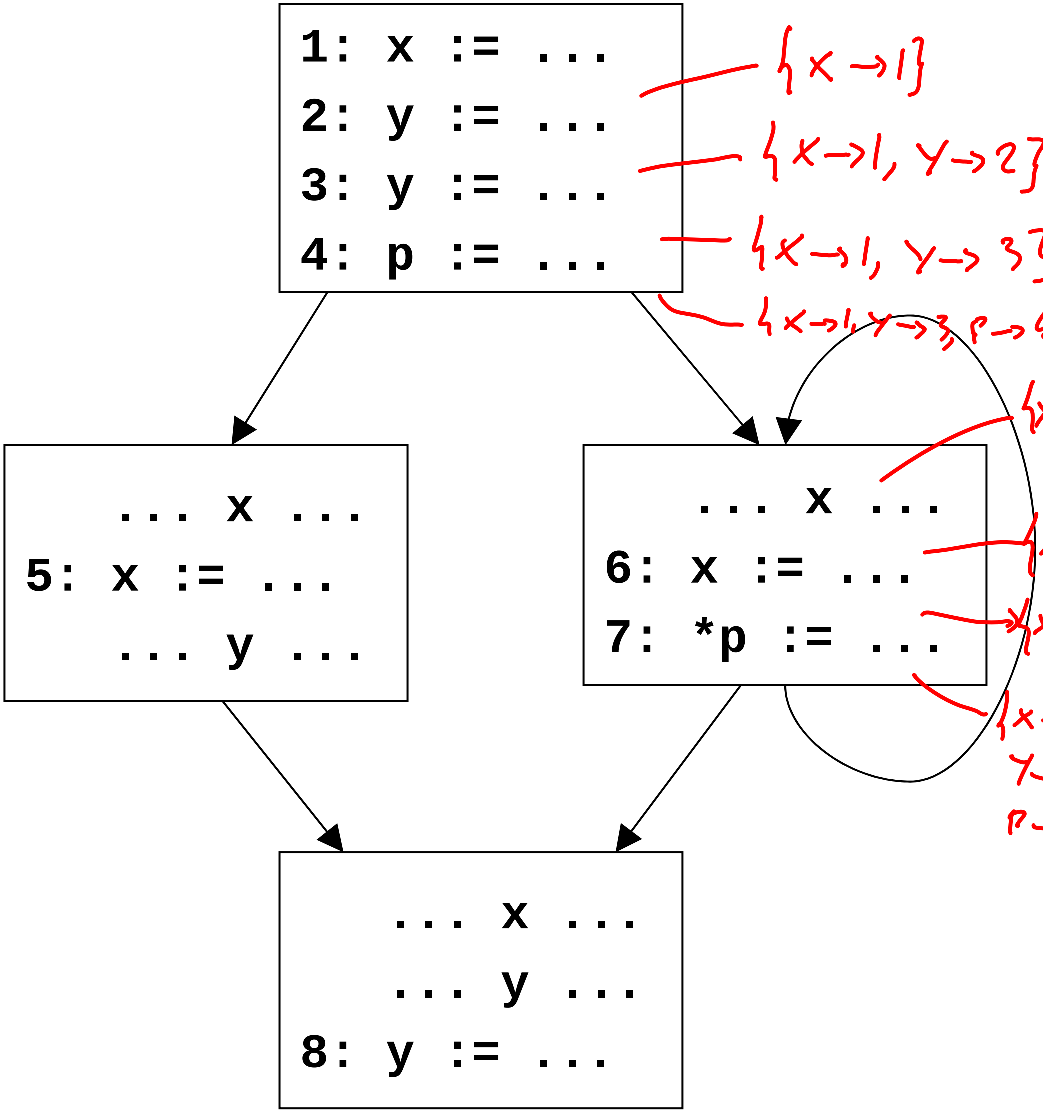
more generally: $out = \bigcup_i in\,[\,i\,]$

Flow functions

- The constraint for a statement kind s often have the form: out = F_s(in)
- F_s is called a ~~flow~~ function
  - other names for it: dataflow function, transfer function
- Given information *in* before statement s, F_s(*in*) returns information after statement s

The Problem of Loops

- If there is no loop, the topological order can be adopted to evaluate transfer functions of statements.
- What if loops?



```
1: x := ...
2: y := ...
3: y := ...
4: p := ...
```

```
   ... x ...
5: x := ...
   ... y ...
```

```
   ... x ...
6: x := ...
7: *p := ...
```

```
   ... x ...
   ... y ...
8: y := ...
```

{x→1}

{x→1, y→2}

{x→1, y→3}

{x→1, y→3, p→4}

{x→1, y→3, p→4}

{x→1, y→3, p→4}

{x→6, y→3, p→4}

{x→6, x→7, y→3, y→7, p→4, p→7}

{x→1, x→6, x→7, y→3, y→7, p→4, p→

"

"

"

Solution: iterate!

- Initialize all sets to the empty
- Store all nodes onto a worklist
- while worklist is not empty:
  - remove node n from worklist
  - apply flow function for node n
  - update the appropriate set, and add nodes whose inputs have changed back onto worklist

Termination

- How do we know the algorithm terminates?

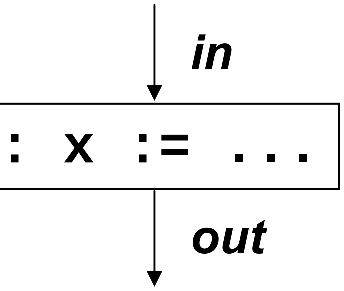
- Because
  - operations are *monotonic*
  - the domain is finite

Monotonicity

- Operation *f* is monotonic if

$$X \subseteq Y \Rightarrow f(x) \subseteq f(y)$$

- We require that all operations be monotonic
  - Easy to check for the set operations
  - Easy to check for all transfer functions; recall:



$$out = in - \{ x \to s' \mid s' \in stmts \} \cup \{ x \to s \}$$

Termination again

- To see the algorithm terminates
  - All variables start empty
  - Variables and rhs’s only increase with each update
  - Sets can only grow to a max finite size
- Together, these imply termination

What Else In DFA

- May vs. must
- Backward vs. Forward
- Lattice
  - Mere goal: help prove the termination of the analysis
  - To show the domain is finite (has finite height)

Where is Dataflow Analysis Useful?

- Best for flow-sensitive, context-insensitive, distributive problems on small pieces of code
  - E.g., the examples we've seen and many others

- Extremely efficient algorithms are known
  - Use different representation than control-flow graph, but not fundamentally different

Where is Dataflow Analysis Weak?

- Lots of places

Data Structures

- Not good at analyzing data structures
- Works well for atomic values
  - Labels, constants, variable names
- Not easily extended to arrays, lists, trees, etc.

The Heap

- Good at analyzing flow of values in local variables
- No notion of the heap in traditional dataflow applications
  - Aliasing

Context Sensitivity

- Standard dataflow techniques for handling context sensitivity don’t scale well

Flow Sensitivity (Beyond Procedures)

- Flow sensitive analyses are standard for analyzing single procedures

- Not used (or not aware of uses) for whole programs
  - Too expensive

The Call Graph

- Dataflow analysis requires a call graph
  - Or something close

- Inadequate for higher-order programs
  - First class functions
  - Object-oriented languages with dynamic dispatch

- Call-graph hinders algorithmic efficiency

Coming Back: The Essence of Static Analysis

- Examine the program text (no execution)
- Build a model of the program state
  - An abstract of the run-time state
- Reason over the possible behaviors.
  - E.g. “run” the program over the abstract state
- The property an analysis needs to promise is that it TERMINATES
  - Slogan of most researchers:

Finite Lattices + Monotonic Functions = Program Analysis

Tips on Designing Analysis

- Program analysis is a formalization of INTUITIVE insights.
  - Type inference
  - Reaching definition
  - …
- Steps
  - Look at the code (segment), gain insights;
  - More systematic: manually “runs” through the code with your abstraction.
  - Works? Good, lets do formalization.

Next Lecture

- Dynamic Program Analysis