Interprocedural Analysis part 2; pointer analysis part 1

Claire Le Goues

15-819O: Program Analysis

institute for SOFTWARE RESEARCH

Context-sensitive analysis

- Analyze a function multiple times, differentiating by call site.
  - **Too bad we allow recursive functions!**
- Build a summary: map input to output info, and analyze each function once per context.
  - Different definitions of context give different tradeoffs.
  - Option 1: input dataflow information.
  - Option 2: call strings

Graph reachability: two sentence summary

- It is possible to encode a certain set of dataflow analysis problems in a way that lets you solve them interprocedurally in polynomial time.
- And the problem representation is compact!

isr institute for SOFTWARE RESEARCH

Meet over realizable paths

- Before, collecting dataflow facts that hold at a node by taking the **meet over all paths** (MOP)
- Instead, consider the **meet over all realizable paths** (MRP)
  - …Which is still sound, but more precise.
- This is good, because it means that many analyses can be treated as context-free language reachability problems over directed graphs.

isr institute for SOFTWARE RESEARCH

Interprocedural finite distributive subset (IFDS) problems

- Interprocedural dataflow analysis with:
  - Finite set of data flow facts
  - Distributive dataflow functions ($f(a \sqcap b) = f(a) \sqcap f(b)$)
- Can convert any IFDS problem as a CFL-graph reachability problem, and find the MRP solution with no loss of precision
  - May be some loss of precision phrasing problem as IFDS

Instance of an IFDS problem

- G* is a supergraph
- D is a finite set of dataflow facts
- $F \subseteq 2^D \rightarrow 2^D$ is a set of distributive functions
- $M : E^* \rightarrow F$ is a map from G^*'s edges to dataflow functions
- meet is either union or intersection (assume it's just join; trust me)

institute for SOFTWARE RESEARCH

Or…

- Interprocedural dataflow analysis with
  - Finite set of dataflow facts
  - Distributive dataflow functions ($f(a \sqcup b) = f(a) \sqcup f(b)$)
  - For example?
  - But not?

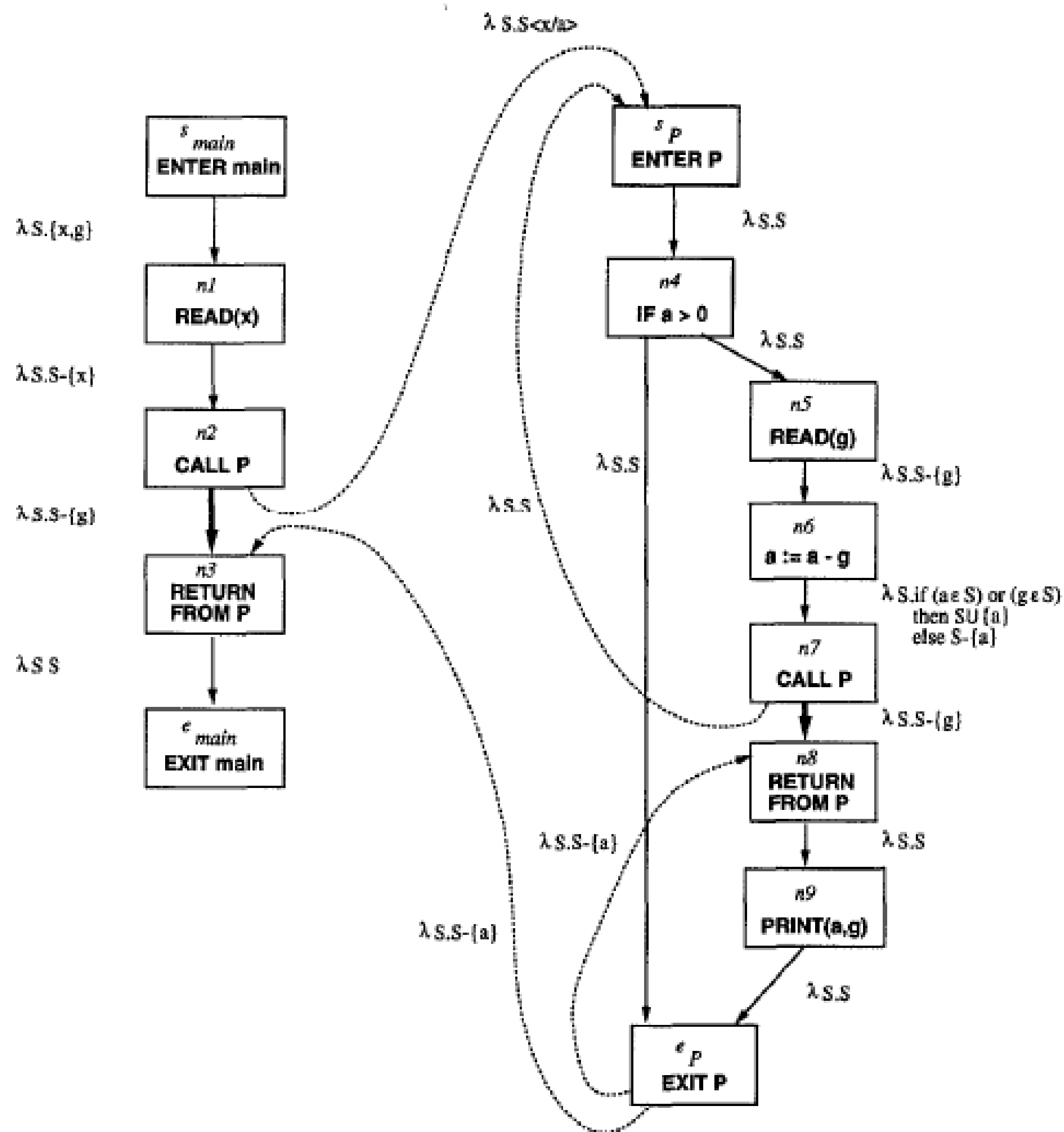
institute for SOFTWARE RESEARCH

```
declare g: integer

program main
begin
  declare x: integer
  read(x)
  call P (x)
end


procedure P (value a : integer)
begin
  if (a > 0) then
    read(g)
    a := a − g
    call P (a)
    print(a, g)
  fi
end
```

(a) Example program



(b) Its supergraph G^*

Valid/Realizable paths

- Grammar that represents balanced calls/returns.
- matched ::= “” | e
  | matched matched
  | $(_i$ matched $)_i$
- Grammar of partially balanced parents is the same, without the second $)_i$

isr institute for SOFTWARE RESEARCH

Solution to IFDS problems

- MVP solution: meet-over-all-valid paths solution

$$MVP_n = \bigsqcap_{q \in \mathrm{IVP}(s_{main},\, n)} pf_q(\top) \qquad \text{for each } n \in N^*.$$

- Don't look at the formula – Intuitively it means "union of all solutions from initial state to node n"
- Example: Possibly uninitialized variables problem

isr institute for SOFTWARE RESEARCH

On the board!

FUNCTIONS AS RELATIONS

isr institute for SOFTWARE RESEARCH

Converting it into a graph reachability problem

- Represent functions as relations:

$$R_f =_{df} \{(\mathbf{0}, \mathbf{0})\} \cup \{(\mathbf{0}, y) \mid y \in f(\emptyset)\} \cup \{(x, y) \mid y \in f(\{x\}) \text{ and } y \notin f(\emptyset)\}.$$

Example. The following table shows three functions and their representation relations:

id: $2^{\{a,b\}} \to 2^{\{a,b\}}$ id = $\lambda S.S$	a: $2^{\{a,b\}} \to 2^{\{a,b\}}$ a = $\lambda S.\{a\}$	f: $2^{\{a,b,c\}} \to 2^{\{a,b,c\}}$ f = $\lambda S.(S - \{a\}) \cup \{b\}$
0 a b 0 a b	0 a b 0 a b	0 a b c 0 a b c

- Composition of two relations relates to the compositions of two functions

$$\text{For all } f, g \in 2^D \to 2^D, [\![R_f ; R_g]\!] = g \circ f.$$

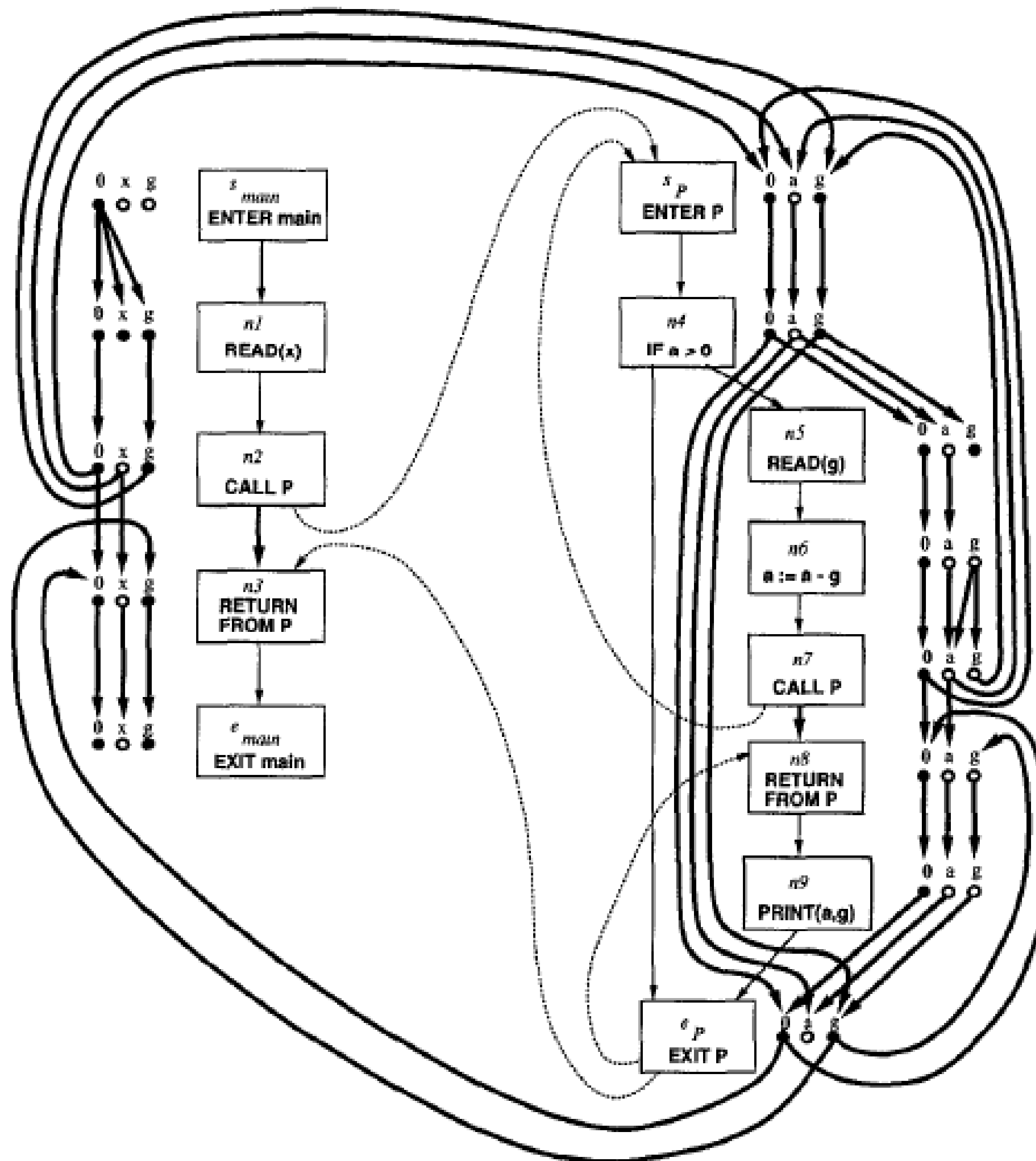
- Using this property, we define exploded supergraph.

isr institute for SOFTWARE RESEARCH

Exploded supergraph

- Let G^* be supergraph (i.e., interprocedural CFP)
- For each node $n \in G^*$, there is node $\langle n,\Lambda\rangle \in G\#$
- For each node $n \in G^*$, and $d \in D$ there is node $\langle n,d\rangle \in G\#$
- For function f associated with edge $a \rightarrow b \in G^*$
  - Edge $\langle a,\Lambda\rangle \rightarrow \langle b,d\rangle$ for every $d \in f(\emptyset)$
  - Edge $\langle a, d1\rangle \rightarrow \langle b,d2\rangle$ for every $d2 \in f(\{d2\})$ - $f(\emptyset)$
  - Edge $\langle a,\Lambda\rangle \rightarrow \langle b, \Lambda\rangle$

isr institute for SOFTWARE RESEARCH



IST SOFTWARE RESEARCH

CFL

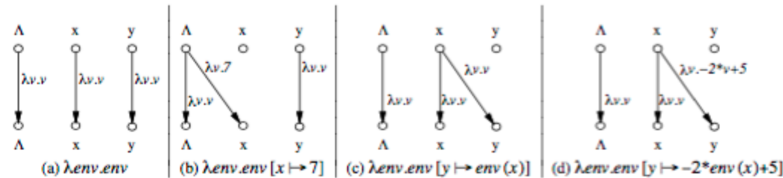
- If L is a CF language over an alphabet, and G a graph with edges from that alphabet.
- Each path in G is a word over the alphabet
- A path is an L-path if it its word is in L
- Reachability problems:
  - All-pairs L-path problem: all pairs of nodes such that there is an L path between them
  - Single-source/target L-path problem: all nodes n_2 such that there is an L-path from a given node n_1 to n_2 (or vice versa).
  - Single source single-target: is there an L-path between two given nodes?
- Why? All CFL reachability problems can be solved in time cubic in the nodes of the graph.

High points of algorithm

- Worklist algorithm.
- Computes summary edges that capture the partial effects of functions on datflow information.
- Tabulation algorithm in paper is a dynamic programming approach.

Interprocedural Distributive Environment (IDE) problems

- Interprocedural dataflow analysis with:
  - Dataflow info at program point represented as a finite **environment** (i.e., mapping from variables/locations to finite height domain of values)
  - Transfer function distributive "environment transformer":



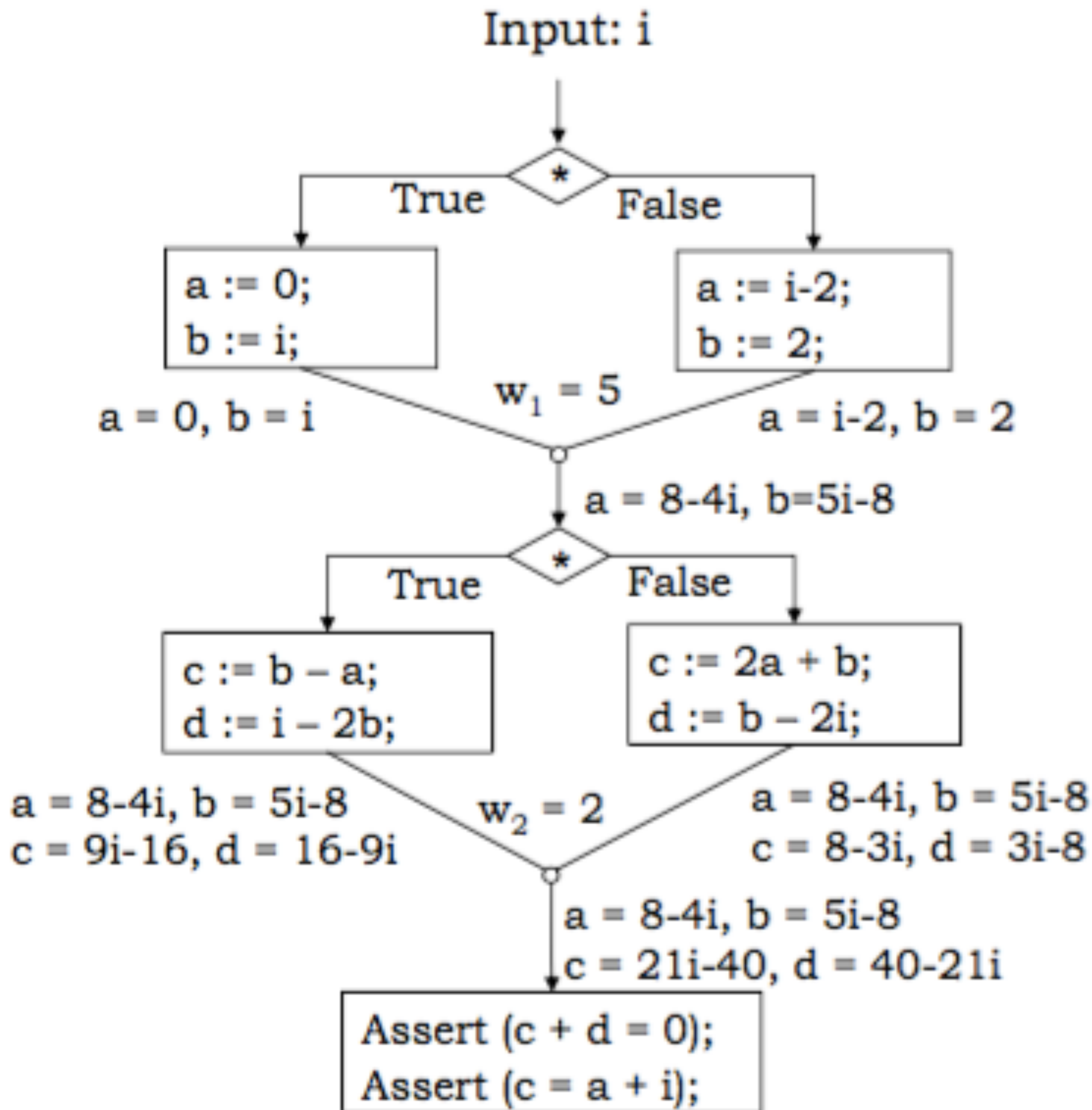
- E.g., copy constant propagation
  - interprets assignment statements such as x=7 and y=x
- E.g. linear constant propagation
  - also interprets assignment statements such as y = 5*z + 9

institute for SOFTWARE RESEARCH

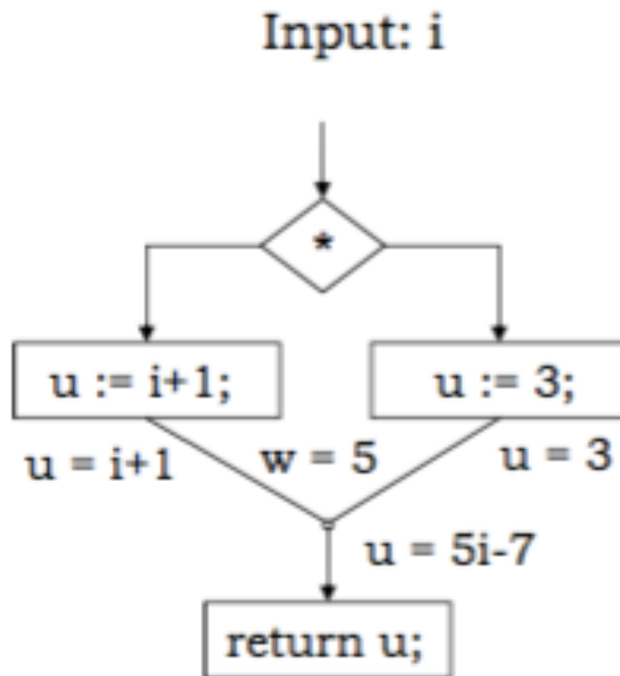
Solving

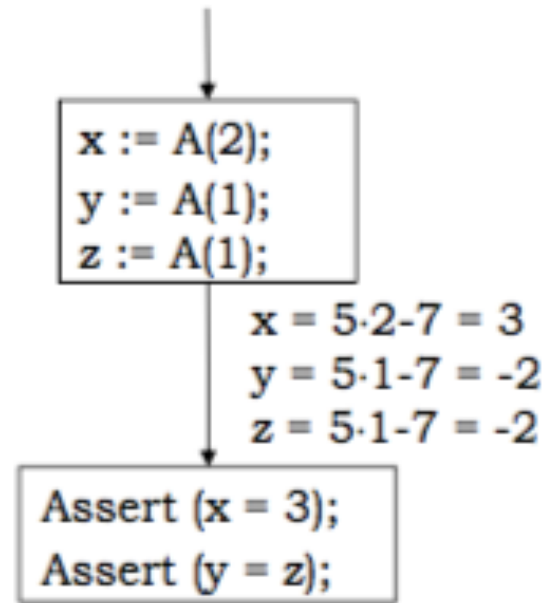
- First pass computes **jump functions** and **summary functions**
  - Summaries of paths within a procedure and of procedure calls, respectively
- Second pass uses these functions to compute environments at program points
- More details in “Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation” by Sagiv, Reps, and Horwitz, 1996.

RANDOM INTERPRETATION?



titute for
FTWARE
RESEARCH





isr institute for SOFTWARE RESEARCH

Pointer (or points-to) analysis

```
I ::= ...
    | p := & x
    | p := q
    | *p := q
    | p := *q
```


isr institute for SOFTWARE RESEARCH

What memory locations can a pointer expression refer to?

- Alias analysis: When do two pointer expressions refer to the same storage location?
  1. int x;
  2. p = &x;
  3. q = p;
- *p and *q alias, as do x and *p, and x and *q

institute for SOFTWARE RESEARCH

Why do we want to know?

- Pointer analysis tells us what memory locations code uses or modifies, and is useful in many other analyses:
  - Available expressions: *p = a + b; y = a + b;
    - If *p aliases a or b, then second computation of a +b is not redundant
  - Constant propagation: x = 3; *p = 4; y = x;
    - Is y constant?
      - If *p and x do not alias each other, then yes.
      - If *p and x always alias each other, then yes.
      - If *p and x sometimes alias each other, then no.

Aliasing can arise due to

- Pointers, e.g., int *p, i; p = &i;
- Call-by-reference

  void m(Object a, Object b) { … }

  m(x,x); // a and b alias in body of m

  m(x,y); // y and b alias in body of m
- Array indexing:

  int i,j,a[100];

  i = j; // a[i] and a[j] alias

Many approaches to this

- Pointer/alias analysis had a major boom in research interest.
- Two “big names”/approaches to know: Andersen and Steensgaard.

isr institute for SOFTWARE RESEARCH

Dimensions of pointer analysis

- Intraprocedural / interprocedural
- Flow-sensitive / flow-insensitive
- Context-sensitive / context-insensitive
- Definiteness
- May versus must
- Heap modeling
- Representation

Flow-sensitive vs flow-insensitive

- Flow-sensitive: computes for each program point what memory locations pointer expressions may refer to.
  - Traditionally too expensive to perform for whole program.
- Flow-insensitive: computes what memory locations pointer expressions may refer to, at any time in program execution.
  - Typically used for whole program analyses.

Flow sensitive analysis is *hard*

Alias Mechanism	Intraprocedural May Alias	Intraprocedural Must Alias	Interprocedural May Alias	Interprocedural Must Alias
Reference Formals, No Pointers, No Structures	–	–	Polynomial[1, 5]	Polynomial[1, 5]
Single level pointers, No Reference Formals, No Structures	Polynomial	Polynomial	Polynomial	Polynomial
Single level pointers, Reference Formals, No Pointer Reference Formals, No Structures	–	–	Polynomial	Polynomial
Multiple level pointers, No Reference Formals, No Structures	$\mathcal{NP}$-hard	Complement is $\mathcal{NP}$-hard	$\mathcal{NP}$-hard	Complement is $\mathcal{NP}$-hard
Single level pointers, Pointer Reference Formals, No Structures	–	–	$\mathcal{NP}$-hard	Complement is $\mathcal{NP}$-hard
Single level pointers, Structures, No Reference Formals	$\mathcal{NP}$-hard[14]	Complement is $\mathcal{NP}$-hard	$\mathcal{NP}$-hard[14]	Complement is $\mathcal{NP}$-hard

Table 1: Alias problem decomposition and classification

Pointer-inducing aliasing: A Problem Classification, Landi and Ryder, POPL 1990

isr institute for SOFTWARE RESEARCH

Context sensitivity

- Also difficult, but success in scaling up to hundreds of thousands LOC
- BDDs, see Whaley and Lam PLDI 2004
  Doop, Bravenboer and Smaragdakis OOPSLA 2009

isr institute for SOFTWARE RESEARCH

Definiteness

- *May* analysis: aliasing that may occur during execution
  - (cf. must-not alias, although often has different representation)
- *Must* analysis: aliasing that must occur during execution
- Sometimes both are useful: consider liveness analysis for *p = *q + 4;
  - If *p must alias x, then x in kill set for statement
  - If *q may alias y, then y in gen set for statement

isr institute for SOFTWARE RESEARCH

May vs must analysis

```
1. z := 1
2. if cond
3.       p := y
4. else
5.       p = &z;
6. *p := 2
7. print z
```


isr institute for SOFTWARE RESEARCH

Possible representations

- Points-to pairs: first element points to the second
  - e.g., (p → b), (q → b)
  - *p and b alias, as do *q and b, as do *p and *q
- Pairs that refer to the same memory
  - e.g., (*p,b), (*q,b), (*p,*q), (**r, b)
  - General, may be less concise than points-to pairs
- Equivalence sets: sets that are aliases
  - e.g., {*p,*q,b}

Problem statement

- Consider flow-insensitive may pointer analysis
- Assume pointers $p,q \in P$ and address-taken variables $a,b \in A$ are disjoint
  - Can transform program to make this true: For any variable v for which this isn't true, add statement pv = &av, replace v with *pv
- Want to compute relation $pts : P \cup A \rightarrow 2^A$
  - Essentially *points to* pairs

institute for SOFTWARE RESEARCH

Andersen-style pointer analysis

- View pointer assignments as *subset constraints*
- Use constraints to propagate points-to information, solve them:

- p = &a;
- q = p;
- p = &b;
- r = p;

- $p \supseteq \{a\}$
- $q \supseteq p$
- $p \supseteq \{b\}$
- $r \supseteq p$

isr institute for SOFTWARE RESEARCH