

## Alias/Pointer Analysis Algorithms

### Last time

- Various aspects of pointer analysis
- Example flow-sensitive algorithm

### Today

- Flow and context insensitive analysis
- Context-sensitive pointer analysis
  - Emami invocation graphs
  - Partial Transfer Functions
- The big picture

CS553 Lecture

Alias/Pointer Analysis Algorithms

1

## FICI Example

### Flow-insensitive context-insensitive (FICI)

```
int** foo(int **p, **q)
{
    int **x;

    x = p;           p →
    . . .           q →
    x = q;           x →
    return x;
}

int main()
{
    int **a, *b, *d, *f,
      c, e;         a →
                  b →
                  d → {c, e}
                  f → {c, e}
                  g → {c, e}

    a = foo(&b, &f);
    *a = &c;
    a = foo(&d, &g);
    *a = &e;
}
```

CS553 Lecture

Alias/Pointer Analysis Algorithms

2

## Flow-Insensitive and Context-Insensitive Pointer Analysis

### The defining characteristics

- Ignore the control-flow graph, and assume that statements can execute in any order
- Rather than producing a solution for each program point, produce a single solution that is valid for the whole program

### Flow-insensitive and Context-Insensitive pointer analyses

- **Andersen-style analysis:** the slowest and most precise
- **Steensgaard analysis:** the fastest and least precise
- All other flow-insensitive pointer analyses are hybrids of these two

CS553 Lecture

Alias/Pointer Analysis Algorithms

3

## Andersen 94

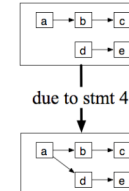
### Overview

- Uses subset constraints
- Cubic complexity in program size,  $O(n^3)$

### Characterization of Andersen

- Whole program
- Flow-insensitive
- Context-insensitive
- May analysis
- Alias representation: points-to
- Heap modeling?
- Aggregate modeling: fields

```
int **a, *b, c, *d, e;
1: a = &b;
2: b = &c;
3: d = &e;
4: a = &d;
```



source: Barbara Ryder's Reference Analysis slides

CS553 Lecture

Alias/Pointer Analysis Algorithms

4

## Steensgaard 96

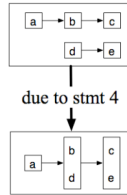
### Overview

- Uses unification constraints
- Almost linear in terms of program size
- Uses fast union-find algorithm
- Imprecision from merging points-to sets

### Characterization of Steensgaard

- Whole program
- Flow-insensitive
- Context-insensitive
- May analysis
- Alias representation: points-to
- Heap modeling: none
- Aggregate modeling: possibly

```
int **a, *b, c, *d, e;
1: a = &b;
2: b = &c;
3: d = &e;
4: a = &d;
```



source: Barbara Ryder's Reference Analysis slides

CS553 Lecture

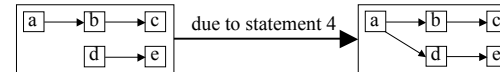
Alias/Pointer Analysis Algorithms

5

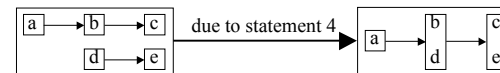
## Andersen vs. Steensgaard

```
int **a, *b, c, *d, e;
1: a = &b;
2: b = &c;
3: d = &e;
4: a = &d;
```

### Andersen-style analysis



### Steensgaard analysis



CS553 Lecture

Alias/Pointer Analysis Algorithms

6

## FSCI Example

### Flow-sensitive context-insensitive (FSCI)

```
int** foo(int **p, **q)
```

```
{
  int **x;
  x = p;
  . . .
  x = q;
  return x;
}
```

We'll see examples of FICS and FSCS later

```
int main()
```

```
{
  int **a, *b, *d, *f,
    c, e;
  a = foo(&b, &f);
  *a = &c;
  a = foo(&d, &g);
  *a = &e;
}
```

p →  
q →  
x<sub>1</sub> →  
x<sub>2</sub> →  
a<sub>1</sub> →  
a<sub>2</sub> →  
f<sub>1</sub> →  
g<sub>1</sub> →  
f<sub>2</sub> →  
g<sub>2</sub> →

CS553 Lecture

Alias/Pointer Analysis Algorithms

7

## Flow-sensitive May Points-To Analysis

### Analogous flow functions

- $\square$  is  $\cup$
- s:  $p = \&x$ ;  
out[s] =  $\{(p \rightarrow x)\} \cup (\text{in}[s] - \{(p \rightarrow y) \forall y\})$
- s:  $p = q$ ;  
out[s] =  $\{(p \rightarrow t) \mid (q \rightarrow t) \in \text{in}[s]\} \cup (\text{in}[s] - \{(p \rightarrow y) \forall y\})$
- s:  $p = *q$ ;  
out[s] =  $\{(p \rightarrow t) \mid (q \rightarrow x) \in \text{in}[s] \ \& \ (x \rightarrow t) \in \text{in}[s]\} \cup (\text{in}[s] - \{(p \rightarrow x) \forall x\})$
- s:  $*p = q$ ;  
out[s] =  $\{(x \rightarrow t) \mid (p \rightarrow x) \in \text{in}[s] \ \& \ (q \rightarrow t) \in \text{in}[s]\} \cup (\text{in}[s] - \{(x \rightarrow x) \forall x \mid (p \rightarrow x) \in \text{in}_{\text{must}}[s]\})$

CS553 Lecture

Alias/Pointer Analysis Algorithms

8

## Must Points-To Analysis

### Analogous flow functions

- $\cap$  is  $\cap$
- s:  $p = \&x$ ;  
 $out_{must}[s] = \{(p \rightarrow x)\} \cup (in_{must}[s] - \{(p \rightarrow x) \forall x\})$
- s:  $p = q$ ;  
 $out_{must}[s] = \{(p \rightarrow t) \mid (q \rightarrow t) \in in_{must}[s]\} \cup (in_{must}[s] - \{(p \rightarrow x) \forall x\})$
- s:  $p = *q$ ;  
 $out_{must}[s] = \{(p \rightarrow t) \mid (q \rightarrow x) \in in_{must}[s] \& (x \rightarrow t) \in in_{must}[s]\} \cup (in_{must}[s] - \{(p \rightarrow x) \forall x\})$
- s:  $*p = q$ ;  
 $out_{must}[s] = \{(x \rightarrow t) \mid (p \rightarrow x) \in in_{must}[s] \& (q \rightarrow t) \in in_{must}[s]\} \cup (in_{must}[s] - \{(x \rightarrow *) \mid (p \rightarrow x) \in in_{must}[s]\})$

### Compute along with may analysis

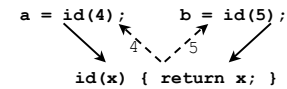
CS553 Lecture

Alias/Pointer Analysis Algorithms

9

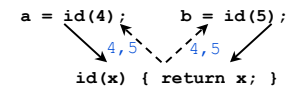
## Recall Context Sensitivity

### Is x constant?



### Context-sensitive analysis

- Computes an answer for every callsite:
  - $x$  is 4 in the first call
  - $x$  is 5 in the second call



CS553 Lecture

Alias/Pointer Analysis Algorithms

10

## FICS Example

### Flow-insensitive context-sensitive (FICS)

```

int** foo(int **p, **q)
{
    int **x;           p1 →
    x = p;             p2 →
    . . .              q1 →
    x = q;             q2 →
    return x;         x1 →
}
int main()
{
    int **a, *b, *d, *f,
      c, e;           x2 →
    a = foo(&b, &f);   a →
    *a = &c;           b →
    a = foo(&d, &g);   d →
    *a = &e;           f →
}
                    g →
    
```

CS553 Lecture

Alias/Pointer Analysis Algorithms

11

## FSCS Example

### Flow-sensitive context-sensitive (FSCS)

```

int** foo(int **p, **q)
{
    int **x;           p11 →
    x = p;             q11 →
    . . .              p21 →
    x = q;             q21 →
    return x;         x11 →
}
int main()
{
    int **a, *b, *d, *f,
      c, e;           x21 →
    a = foo(&b, &f);   x22 →
    *a = &c;           a11 →
    a = foo(&d, &g);   a12 →
    *a = &e;           f11 →
}
                    g11 →
    
```

**P<sub>11</sub>** is circled in red, with an arrow pointing to it labeled "first callsite" and another arrow pointing to the first call site in the code labeled "first def".

CS553 Lecture

Alias/Pointer Analysis Algorithms

12

## Emami 1994

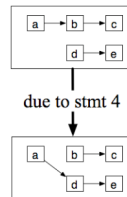
### Overview

- Uses invocation graph for context-sensitivity
- Can be exponential in program size
- Handles function pointers

### Characterization of Emami

- Whole program
- Flow-sensitive
- Context-sensitive
- May and must analysis
- Alias representation: points-to
- Heap modeling: one heap variable
- Aggregate modeling of fields and arrays

```
int **a, *b, c, *d, e;  
1: a = &b;  
2: b = &c;  
3: d = &e;  
4: a = &d;
```



CS553 Lecture

Alias/Pointer Analysis Algorithms

13

## Partial Transfer Functions [Wilson et. al. 95]

### Key idea

- Exploit commonality among contexts
- Provide one procedure summary (PTF) for all contexts that share the same input/output aliasing relationships

CS553 Lecture

Alias/Pointer Analysis Algorithms

14

## Partial Transfer Functions – Example

```
main() {  
  int *a,*b,c,d;  
  a = &c;  
  b = &d;  
  swap(&a, &b); // S0  
  for (i = 0; i<2; i++) {  
    bar(&a,&a); // S1  
    bar(&b,&b); // S2  
    bar(&a,&b); // S3  
    bar(&b,&a); // S4  
  }  
}  
void bar(int **i, int **j) { swap(i,j); }  
void swap(int **x, int **y) {  
  int *temp = *x;  
  *x = *y;  
  *y = temp;  
}
```

How many contexts do we care about?

- Two: the formals either alias or they do not alias

In practice

- Only need 1 or 2 PTF's per procedure
- Complex to implement

15

## The Big Picture

### Where do we lose precision?

- Let's revisit our running example

CS553 Lecture

Alias/Pointer Analysis Algorithms

16

## Revisiting Our Earlier Example (cont)

### Flow-insensitive context-insensitive (FICI)

```

int** foo(int **p, **q)
{
    int **x;

    x = p;
    . . .
    x = q;
    return x;
}

int main()
{
    int **a, *b, *d, *f,
        c, e;

    a = foo(&b, &f);
    *a = &c;
    a = foo(&d, &g);
    *a = &e;
}

```

p → {b, d}  
 q → {f, g}  
 x → {b, d, f, g}

a → {b, d, f, g}  
 b → {c, e}  
 d → {c, e}  
 f → {c, e}  
 g → {c, e}

CS553 Lecture

Alias/Pointer Analysis Algorithms

17

## Revisiting Our Earlier Example (cont)

### Flow-sensitive context-insensitive (FSCI)

```

int** foo(int **p, **q)
{
    int **x;

    x = p;
    . . .
    x = q;
    return x;
}

int main()
{
    int **a, *b, *d, *f,
        c, e;

    a = foo(&b, &f);
    *a = &c;
    a = foo(&d, &g);
    *a = &e;
}

```

p → {b, d}  
 q → {f, g}  
 x<sub>1</sub> → {b, d}  
 x<sub>2</sub> → {f, g}

a<sub>1</sub> → {f, g}  
 a<sub>2</sub> → {f, g}  
 f<sub>1</sub> → {c}  
 g<sub>1</sub> → {c}  
 f<sub>2</sub> → {c, e} (weak update)  
 g<sub>2</sub> → {c, e} (weak update)

CS553 Lecture

Alias/Pointer Analysis Algorithms

18

## Strong vs. Weak Updates

### Strong update

- When we know precisely what an assignment through a pointer refers to, the assignment kills old information
- Such cases are analogous to assignments to scalars

```

int a;
a := 5;
a := 6;

```

{a=5}  
 {a=6}

```

int *a, b;
a := &b;
b := 5;
*a := 6;

```

{a->b}  
 {a->b, b=5}  
 {a->{b}, b=6}

CS553 Lecture

Alias/Pointer Analysis Algorithms

19

## Strong vs. Weak Updates

### Weak update

- When we do not know what an assignment through a pointer refers to, we cannot use that assignment to kill old facts
- So the imprecision spreads

```

int *a, b, c;
if (blah)
    a := &b;
else
    a := &c;
b := 5;
*a := 6;

```

{a->{b, c}}  
 {a->{b, c}, b=5}  
 {a->{b, c}, b=5 □ b=6}

Does not kill {b=5} because \*a might update c and not b

CS553 Lecture

Alias/Pointer Analysis Algorithms

20

## Imprecision

---

### Weak updates

- Occur more often in flow-insensitive and context-insensitive analyses

### The callgraph

- When function pointers are used, pointer analysis is needed to build the callgraph
- Imprecision in pointer analysis leads to imprecision in the callgraph
  - A conservative callgraph has more edges than a less conservative callgraph
- Imprecision in the callgraph leads to further imprecision in the pointer analysis

### The basic issue

- The need for approximation

## Approximations

---

### Many ways to approximate

- Recall that the constraint graph has nodes representing variables and edges representing constraints
- The many dimensions of pointer analysis represent different ways of collapsing the constraint graph

### Flow-insensitive

- Andersen:
  - Collapse all constraints (assignments) pertaining to a given variable into a single node
- Steensgaard:
  - Collapse all nodes that have been assigned to one another into a single node
  - Allows information to flow from rhs to lhs as well as from lhs to rhs

## More Approximations

---

### Context-insensitive analysis

- Collapse all constraints arising from different callsites of a procedure into a single node

### Partial Transfer Functions

- Collapse constraints for all callsites of a procedure that share the same aliasing relationships

### Field-insensitive

- Collapse all fields of a structure into a single node

### Field-based

- Collapse all instances of a struct type into one node per field
- Example: one node for all instances of `student.name`, and another node for all instances of `student.gpa`

## Yet More Approximations

---

### Address Taken

- Collapse all objects that have their address taken into a single node
- Assume that all pointers point to this node

### Heap naming

- One heap:
  - Collapse all heap objects into a single node
- Static allocation site
  - Collapse all instances of objects that are allocated at the same program location into a single node

## Concepts

---

### Partial Transfer Functions

- Exploit commonality among contexts

### Sources of imprecision

## Next Time

---

### Next lecture

- Finding Bugs paper

## Binary Decision Diagrams (BDDs)

---

### A data structure

- Extensively used in the model-checking community

### Benefits

- Compactly represents sets and relations
- Operations are proportional to the size of the BDD, not the size of the set or relation

### How does this apply to pointer analysis?

## Andersen-Style Pointer Analysis – Recap

---

### Program

$a := \&b$   
 $c := a$   
 $a := \&d$   
 $e := a$

### Constraints

$a \supseteq \{ b, d \}$   
 $c \supseteq a$   
 $e \supseteq a$

### Points-to Relations

$a \rightarrow \{ b, d \}$   
 $c \rightarrow \{ b, d \}$   
 $e \rightarrow \{ b, d \}$

We've reached a fixed point

### Base constraints

- Used to initialize the points-to sets
- Ex:  $a := \&b$
- Not needed after initialization

### Complex constraints

- Involve pointer dereferences
- Ex:  $*a := c$

### Simple constraints

- Involve variable names only
- Ex:  $c := a$

### Procedure calls

- Insert constraints for copying parameters and return values

## Andersen-Style Pointer Analysis

### Represent two sets

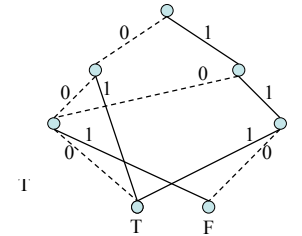
- $C = \{ (a,b) \mid a \supseteq b \}$  // Constraints
- $P = \{ (a,b) \mid a \rightarrow b \}$  // Points-to sets

### Iterate until we reach a fixed point:

- $S = \{ (a,c) \mid \exists b. ((a,b) \in C \wedge (b,c) \in P) \}$  // Propagate constraints
- $P := P \cup S$

## Binary Decision Diagrams (BDDs)

000, 010, 011, 100, 111



## Symbolic Pointer Analysis

### Encode relations as BDDs

- $C = \{ (a,b) \mid a \supseteq b \}$
- $P = \{ (a,b) \mid a \rightarrow b \}$

### Possible strategies

- Encode both  $C$  and  $P$  as BDDs
- Encode  $P$  as a BDD, but not  $C$
- Encode  $C$  as a BDD, but not  $P$

### Recent work

- Success for Java [Whaley and Lam '04]
  - Can analyze 600K lines of code
- Less successful for C—an order of magnitude smaller programs
- Has not yet been applied to flow-sensitive analyses