

Alias Analysis

Last time

- Interprocedural analysis

Today

- Intro to alias analysis (pointer analysis)

Aliasing

What is aliasing?

- When two expressions denote the same **mutable** memory location
- e.g., `p = new Object;`
`q = p;` \Rightarrow `*p` and `*q` alias

How do aliases arise?

- Pointers
- Call by reference (parameters can alias each other or non-locals)
- Array indexing
- C **union**, Pascal variant records, Fortran **EQUIVALENCE** and **COMMON** blocks

Aliasing Examples

Pointers (e.g., in C)

```
int *p, i;  
p = &i;
```

`*p` and `i` alias

Parameter passing by reference (e.g., in Pascal)

```
procedure procl(var a:integer; var b:integer);  
...  
procl(x,x);  
procl(x,glob);
```

`a` and `b` alias in body of `procl`

`b` and `glob` alias in body of `procl`

Array indexing (e.g., in C)

```
int i,j, a[128];  
i = j;
```

`a[i]` and `a[j]` alias

What Can Alias?

Stack storage and globals

```
void fun(int p1) {  
    int i, j, temp;  
    ...  
}
```

do `i`, `j`, or `temp` alias?

Heap allocated objects

```
n = new Node;  
n->data = x;  
n->next = new Node;  
...
```

do `n` and `n->next` alias?

What Can Alias? (cont)

Arrays

```
for (i=1; i<=n; i++) {
    b[c[i]] = a[i];
}
```

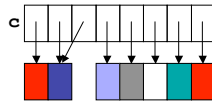
do $b[c[i_1]]$ and $b[c[i_2]]$ alias for any two iterations i_1 and i_2 ?

Can $c[i_1]$ and $c[i_2]$ alias?

Fortran

c 7 1 4 2 3 1 9 0

Java



Alias Analysis

Goal: Statically identify aliases

- Can memory reference m and n access the same state at program point p ?
- What program state can memory reference m access?

Why is alias analysis important?

- Many analyses need to know *what* storage is read and written

e.g., available expressions (CSE)

```
*p = a + b;
y = a + b;
```

If $*p$ aliases a or b , the second expression is not redundant (CSE fails)

- e.g., Reaching definitions (constant propagation)

d_1 : $x = 3$;

d_2 : $*p = 4$;

d_3 : $y = x$;

If $*p$ aliases x , d_2 reaches this point; otherwise, both d_1 and d_2 reach

Otherwise we must be very conservative

How hard is this problem?

Undecidable

- Landi 1992
- Ramalingam 1994

All solutions are conservative approximations

Is this problem solved?

- Why haven't we solved this problem? [Hind 2001]
- Still a number of open issues
 - large programs
 - partial programs
 - modeling the heap (shape analysis)
 - ...

Trivial Alias Analyses

Easiest approach

- Assume that nothing *must* alias
- Assume that everything *may* alias everything else
- Yuck!

Address taken: A slightly better approach (for C)

- Assume that nothing *must* alias
- Assume that all pointer dereferences *may* alias each other
- Assume that variables whose addresses are taken (and globals) *may* alias all pointer dereferences

e.g.,

```
p = &a;
```

```
. . .
```

```
a = 3; b = 4;
```

```
*q = 5;
```

$*q$ and a may alias, so a may be 3 or 5, but $*q$ does not alias b , so b is 4

Enhance with type information?

Properties of Alias Analysis

Scope: Intraprocedural (per procedure) or Interprocedural (whole program)

Representation

- Alias pairs?
- Points-to sets?
- Others...?

Flow sensitivity: Sensitive versus insensitive?

Context sensitivity: Sensitive versus insensitive?

Definiteness: May versus must?

Heap Modeling?

Aggregate Modeling?

Representations of Aliasing

Equivalence sets

- All memory references in the same set are aliases
- e.g., $\{ *a, b \}$, $\{ *b, c, **a \}$

Alias pairs

- Pairs that refer to the same memory
 - e.g., $(*a, b)$, $(*b, c)$, $(**a, c)$
 - Completely general
- ```
int **a, *b, c, *d, e;
1: a = &b;
2: b = &c;
```

### Points-to pairs [Emami94]

- Pairs where the first member points to the second
- e.g.,  $( a \rightarrow b )$ ,  $( b \rightarrow c )$
- Possibly more compact than alias pairs

## Flow Sensitivity of Alias Analysis

### Flow-sensitive alias analysis

- Compute aliasing information at each program point

e.g.,

```
p = &x;
```

```
...
```

```
p = &y;
```

**\*p and x alias here**

**\*p and y alias here**

### Flow-insensitive alias analysis

- Compute aliasing information for entire procedure

e.g.,

```
p = &x;
```

```
...
```

```
p = &y;
```

**\*p may alias x or y  
in this procedure**

## Definiteness of Alias Information

### May (possible) alias information

- Indicates what might be true

e.g.,

```
if (c) p = &i;
```

**\*p and i may alias**

### Must (definite) alias information

- Indicates what is definitely true

e.g.,

```
p = &i;
```

**\*p and i must alias**

### Often need both

Recall:  $in[s] = use[s] \cup (out[s] - def[s])$

- e.g., Consider liveness analysis

```
s: *p = *q+4;
```

- (1) **\*p must alias v**  $\Rightarrow$   $def[s] = kill[s] = \{v\}$
- (2) **\*q may alias v**  $\Rightarrow$   $use[s] = gen[s] = \{v\}$

Suppose  $out[s] = \{v\}$

## Flow-sensitive May Points-To Analysis

### Analogous flow functions

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

## Must Points-To Analysis

### Analogous flow functions

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

### Compute along with may analysis

## Other Issues (Modeling the Heap)

### Issue

- Each allocation creates a new piece of storage  
*e.g.*,  $\mathbf{p} = \text{new } \mathbf{T}$

### Proposal?

- Generate (at compile-time) a new “variable” to stand for new storage
- **newvar**: Creates a new variable

### Flow function

- s:  $\mathbf{p} = \text{new } \mathbf{T}$ ;  
 $\text{out}[s] = \{(\mathbf{p} \rightarrow \text{newvar})\} \cup (\text{in}[s] - \{(\mathbf{p} \rightarrow \mathbf{x}) \forall \mathbf{x}\})$

### Problem

- Domain is unbounded!
- Iterative data-flow analysis may not converge

## Modeling the Heap (cont)

### Simple solution

- Create a summary “variable” (node) for each allocation statement
- Domain:  $2^{(\text{Var} \cup \text{Stmt}) \times (\text{Var} \cup \text{Stmt})}$  rather than  $2^{\text{Var} \times \text{Var}}$
- *Monotonic* flow function  
s:  $\mathbf{p} = \text{new } \mathbf{T}$ ;  
 $\text{out}[s] = \{(\mathbf{p} \rightarrow \text{stmt}_i)\} \cup (\text{in}[s] - \{(\mathbf{p} \rightarrow \mathbf{x}) \forall \mathbf{x}\})$
- Less precise (but finite)

### Alternatives

- Summary node for entire heap
- Summary node for each type
- K-limited summary
  - Maintain distinct nodes up to k links removed from root variables

## Using Alias Information

### Example: reaching definitions

- Compute at each point in the program a set of  $(s, v)$  pairs, indicating that statement  $s$  may define variable  $v$

### Flow functions

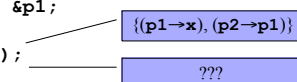
- $s: *p = x;$   
 $out_{reach}[s] = \{(s, z) \mid (p \rightarrow z) \in in_{may-pt}[s]\} \cup$   
 $(in_{reach}[s] - \{(t, y) \mid (p \rightarrow y) \in in_{must-pt}[s]\})$
- $s: x = *p;$   
 $out_{reach}[s] = \{(s, x)\} \cup (in_{reach}[s] - \{(t, x) \mid \forall t\})$
- ...

## Function Calls

### Question

- How do function calls affect our points-to sets?

*e.g.*,  
`p1 = &x;`  
`p2 = &p1;`  
...  
`foo();`



### Be conservative

- Assume that any reachable pointer may be changed
- Pointers can be “reached” via globals and parameters
  - May pass through objects in the heap
- Can be changed to anything reachable or something else
- Can we prune aliases using types?

### Problem

- Lose a lot of information

## Concepts

### What is aliasing and how does it arise

#### Properties of alias analyses

- Definiteness: may or must
- Flow sensitivity: sensitive or insensitive
- Context sensitivity: sensitive or insensitive (interprocedural only)
- Representation: alias pairs, points-to sets

#### Function calls degrade alias information

- Context-sensitive interprocedural analysis

## Next Time

### Lecture

- Flow and context insensitive alias analysis