Alias Analysis

Last time

– Reuse optimization

Today

– Alias analysis (pointer analysis)

Next time

– More alias analysis (pointer analysis)

Aliasing

What is aliasing?

– When two expressions denote the same **mutable** memory location

– *e.g.*, `p = new Object;`
`q = p;` ⇒ `*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 proc1(var a:integer; var b:integer);
. . .
proc1(x,x);
proc1(x,glob);
```

`a` and `b` alias in body of `proc1`

`b` and `glob` alias in body of `proc1`

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];
}
```

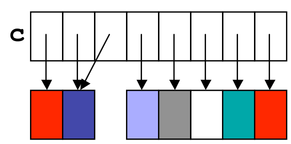
do `b[c[i`$_1$`]]` and `b[c[i`$_2$`]]` alias for any two interations `i`$_1$ and `i`$_2$?

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
- Ramalingan 1994

All solutions are conservative approximations

Is this problem solved?

- Why haven't we solved this problem? [Hind 2001]
- Wednesday and next week we will look at some open issues

Alias/Pointer Analysis Survey

Today

- Address Taken
- Steensgaard (unification)

Tomorrow

- Anderson (inclusion)
- Emami

Next Week

- Burk
- Choi

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

```
[Shapiro & Horwitz 97]

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 -> b`), (`b -> 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] ∪ (out[s] – def[s])

- *e.g.,* Consider liveness analysis

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

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

Suppose out[s] = {**v**}

FIAlias [Landi & Ryder] equivalent to Steensgaard

Overview

- Put all interesting memory references in separate equivalence sets
- Merge equivalence sets based on pointer assignments
- Merge equivalence sets based on type 2 alias effects, (e.g., merging *a with d will cause merge of equiv sets with b and d, and those with e and c)

Characterization of Steensgaard

- Whole program
- Flow-insensitive
- Context-insensitive
- May analysis
- Alias representation: equivalence sets
- Heap modeling?
- Aggregate modeling?

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

Next Time

Reading

- [Emami95]

Lecture

- Alias Analysis II
  - Andersen
  - Emami