Introduction to Alias Analysis

Last time

– Common Subexpression Elimination

– Partial Redundancy Elimination

Today

– Alias analysis

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)

Otherwise we must be *very* conservative

Constant Propagation Revisited

```
{
    int x, y, a;
    int *p;

    p = &a;
    x = 5;

    y = x + 1;
}
```

Is x constant here?

– Yes, only one value of x reaches this last statement

The Importance of Pointer Analysis

```
{
    int x, y, a;
    int *p;

    p = &a;
    x = 5;
   *p = 23;
    y = x + 1;
}
```

Is x constant here?

– If **p** does not point to **x**, then **x = 5**
– If **p** definitely points to **x**, then **x = 23**
– If **p** might point to **x**, then we have two reaching definitions that reach this last statement, so **x** is not constant

Trivial Pointer Analysis

```
{
    int x, y, a;
    int *p;

    p = &a;
    x = 5;
   *p = 23;
    y = x + 1;
}
```

No analysis

- Assume that nothing *must* alias
- Assume that everything *may* alias everything else
- Yuck!
- Enhance this with type information?

Is x constant here?

- With our trivial analysis, we assume that **`p`** *may* point to **`x`**, so **`x`** is not constant

A Slightly Better Approach (for C)

```
{
    int x, y, a;
    int *p;

    p = &a;
    x = 5;
   *p = 23;
    y = x + 1;
}
```

Address Taken

- 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

Is x constant here?

- With Address Taken, **`*p`** and **`a`** may alias, but neither aliases with **`x`**

Address Taken (cont)

```
{
    int x, y, a;
    int *p, *q;
    q = &x;
    p = &a;
    x = 5;
   *p = 23;
    y = x + 1;
}
```

Is x constant here?

– With Address Taken, we now assume that **`*p`**, **`*q`**, **`a`**, and **`x`** all alias

A Better Points-To Analysis

Goal

– At each program point, compute set of (**`p`**→**`x`**) pairs if **`p`** points to **`x`**

Properties

– Use dataflow analysis

– May information (will look at must information next)

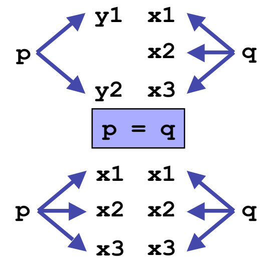
May Points-To Analysis

Domain: $2^{var \times var}$

Direction: forward

Flow functions

- s: `p = &x;`
- s: `p = q;`



Meet function: $\cup$

What if we have pointers to pointers?

- *e.g.*, `int **q;  p = *q;`

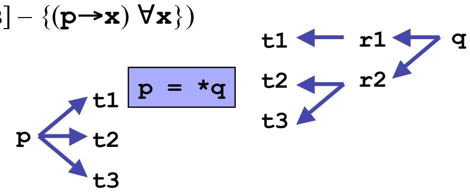
May Points-To Analysis (Pointers to Pointers)

Additional flow functions

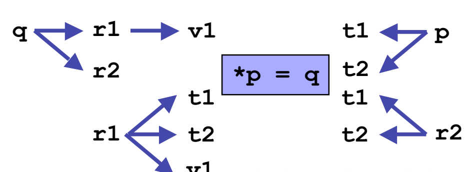
- s: `p = *q;`

  out[s] = {(**p**→**t**) | (**q**→**r**) ∈ in[s] & (**r**→**t**) ∈ in[s]} ∪

  (in[s] – {(**p**→**x**) ∀**x**})



- s: `*q = p;`



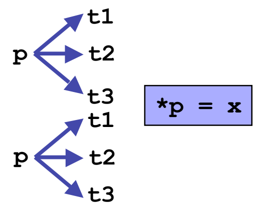
May Points-To Analysis (cont)

What is the flow function for the following statement?

– s: **`x = 3;`**

`*p = x;`

out[s] = in[s]



Dealing with Dynamically Allocated Memory

Issue

– Each allocation creates a new piece of storage
e.g., **`p = new T`**

Proposal?

– Generate (at compile-time) a new “variable” to stand for new storage

Flow function

– s: **`p = new T;`**
out[s] = {(**p**→**newvar**)} ∪ (in[s] – {(**p**→**x**) ∀**x**})

Problem

– Domain is unbounded!

– Iterative data-flow analysis may not converge

Dynamically Allocated Memory (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: **`p = new T;`**
  out[s] = {(**p**→**stmt$_s$**)} ∪ (in[s] − {(**p**→**x**) ∀**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

Must Points-To Analysis

Meet function: ∩

Analogous flow functions

- s: **`p = &x;`**
  $out_{must}[s]$ = {(**p**→**x**)} ∪ ($in_{must}[s]$ − {(**p**→**x**) ∀**x**})
- s: **`p = q;`**
  $out_{must}[s]$ = {(**p**→**t**) | (**q**→**t**) ∈ $in_{must}[s]$} ∪ ($in_{must}[s]$ − {(**p**→**x**) ∀**x**)})
- s: **`p = *q;`**
  $out_{must}[s]$ = {(**p**→**t**) | (**q**→**r**) ∈ $in_{must}[s]$ & (**r**→**t**) ∈ $in_{must}[s]$} ∪
  (in_{must} [s] − {(**p**→**x**) ∀**x**)})
- s: **`*p = q;`**
  $out_{must}[s]$ = {(**r**→**t**) | (**p**→**r**) ∈ $in_{must}[s]$ & (**q**→**t**) ∈ in_{must} [s]} ∪
  ($in_{must}[s]$ − {(**r**→**x**) ∀**x** | (**p**→**r**) ∈ $in_{must}[s]$})

Compute this along with may analysis

- Why?

Definiteness of Alias Information

Often need both

- Consider liveness analysis

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

Recall: in[s] = use[s] ∪ (out[s] – def[s])

(1) `*p` *must* alias `v` ⇒ def[s] = kill[s] = {`v`}

Suppose out[s] = {`v`}

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

Using Points-To Information

```
{
    int x, y, a;
    int *p, *q;
    q = &x;        {(q→x)}
    p = &a;        {(q→x)}(p→a)}
    x = 5;         {(q→x)}(p→a)}
   *p = 23;        {(q→x)}(p→a)}
    y = x + 1;     {(q→x)}(p→a)}
}
```

To support constant propagation, first run points-to analysis

Then run constant propagation

- Since `*p` and `x` do not alias, `x` is constant in this last statement

The point

- Pointer analysis is an enabling analysis

Using Points-To Information (cont)

Example: reaching definitions

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

Flow functions

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

Function Calls

```
{
    int x, y, a;
    int *p;

    p = &a;
    x = 5;
    foo(&x);
    y = x + 1;
}
```

```
foo (int *p)
{
    return p;
}
```

Does the function call modify x?

- With our intra-procedural analysis, we don't know
- Make worst case assumptions
  - Assume that any reachable pointer may be changed
  - Pointers can be "reached" via globals and parameters
    - May pass through objects in the heap
- More next time

Let's Take a Step Back

We've been talking about pointers

– Are there other ways for memory locations to alias one another?

How else can we represent alias information?

How Do Aliases Arise?

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];
}
```

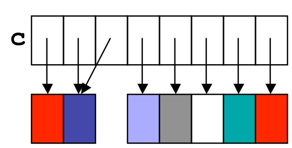
do $\mathbf{b[c[i_1]]}$ and $\mathbf{b[c[i_2]]}$ alias for any two iterations $\mathbf{i_1}$ and $\mathbf{i_2}$?

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

Fortran

c	7	1	4	2	3	1	9	0

Java



Representations of Aliasing

Points-to pairs [Emami94]
- Pairs where the first member points to the second
  e.g., (`a -> b`), (`b -> c`)

Alias pairs [Shapiro & Horwitz 97]
- Pairs that refer to the same memory
  e.g., `(*a,b), (*b,c), (**a,c)`
- Completely general
- May be less concise than points-to pairs

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

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

How hard is this problem?

Undecidable
- Landi 1992
- Ramalingan 1994

All solutions are conservative approximations

Is this problem solved?
- Numerous papers in this area
- Haven't we solved this problem yet? [Hind 2001]

Concepts

What is aliasing and how does it arise?

Properties of alias analyses

- Definiteness: may or must
- Representation: alias pairs, points-to sets

Function calls degrade alias information

- Context-sensitive interprocedural analysis

Next Time

Lecture

- Interprocedural analysis