PRE Example

B1: `a := b + c`   B2: `b := b + 1`

B3: `a := b + c`

	B1	B2	B3
transparent	{b+c}		{b+c}
locally_available	{b+c}		{b+c}
locally_anticipated	{b+c}	{b+1}	{b+c}
available_in			
available_out	{b+c}		{b+c}
partially_available_in			{b+c}
partially_available_out	{b+c}		{b+c}
anticipated_out	{b+c}	{b+c}	
anticipated_in	{b+c}	{b+1}	{b+c}
ppout	{b+c}	{b+c}	
ppin			{b+c}
insert		{b+c}	
delete			{b+c}

Introduction to Alias Analysis

Last time

- Partial Redundancy Elimination

Today

- Alias analysis

Alias Analysis (aka Pointer Analysis)

Goal: Statically identify aliases

- Can memory references 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? (pointing to `y = x + 1;`)

- 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) 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 may alias

A Better Points-To Analysis

Goal

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

Properties

– Use data-flow analysis

– May information (will look at must information next)

May Points-To Analysis

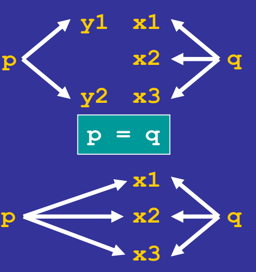
Domain: $2^{\text{var} \times \text{var}}$

Direction: forward

Flow functions

– s: `p = &x;`

– s: `p = q;`



Meet function: $\cup$

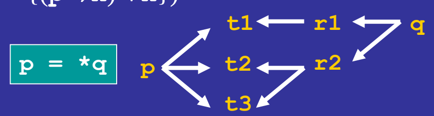
What if we have pointers to pointers?

– *e.g.*, `int **q;  p = *q;`

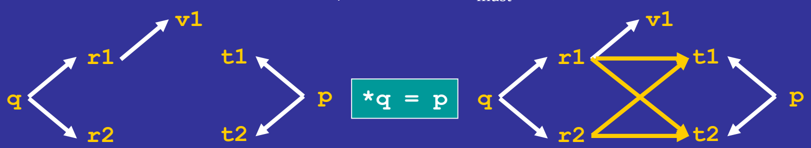
May Points-To Analysis (Pointers to Pointers)

Additional flow functions

- s: `p = *q;`
  $out[s] = \{(\mathbf{p} \rightarrow \mathbf{t}) \mid (\mathbf{q} \rightarrow \mathbf{r}) \in in[s] \;\&\; (\mathbf{r} \rightarrow \mathbf{t}) \in in[s]\} \;\cup$
  $(in[s] - \{(\mathbf{p} \rightarrow \mathbf{x})\ \forall \mathbf{x}\})$



- s: `*q = p;`
  $out[s] = \{(\mathbf{r} \rightarrow \mathbf{t}) \mid (\mathbf{q} \rightarrow \mathbf{r}) \in in[s] \;\&\; (\mathbf{p} \rightarrow \mathbf{t}) \in in[s]\} \cup$
  $(in[s] - \{(\mathbf{r} \rightarrow \mathbf{x})\ \forall \mathbf{x} \mid (\mathbf{q} \rightarrow \mathbf{r}) \in in_{must}[s]\})$



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 name to represent each new allocation
- `newvar`: Creates a new variable

Flow function

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

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: $2^{(Var \cup Stmt) \times (Var \cup Stmt)}$ rather than $2^{Var \times Var}$
- 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
- This dimension is often referred to as "heap naming"

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**→*) | (**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

To support constant propagation, first run points-to analysis

```
{
    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)}
}
```

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

Integrated Pointer Analysis

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

– s: `x = *p;`
$out_{reach}[s] = \{(\mathbf{x},s\} \cup (in_{reach}[s] - \{(\mathbf{x},t) \; \forall t\}$

– . . .

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 Wednesday

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

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

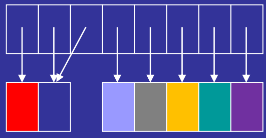
Fortran

c

7	1	4	2	3	1	9	0

Java

c



Representations of Aliasing

Points-to pairs [Emami94]

– Pairs where the first member points to the second
  e.g., (**a -> b**), (**b -> c**)

Alias pairs

– 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

[Shapiro & Horwitz 97]

```
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