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

```c
{  
    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

```c
{  
    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

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

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)

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

Is x constant here?
– With Address Taken, *p and a may alias, but neither aliases with x
```

No analysis
– Assume that nothing must alias
– Assume that everything may alias everything else
– Yuck!
– Enhance this with type information?
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 \( (p \rightarrow x) \) pairs if \( p \) points to \( x \)

Properties
  – Use dataflow analysis
  – May information (will look at must information next)
May Points-To Analysis

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

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

- $s$: $p = \ast q$;

```
\begin{align*}
\text{out}[s] &= \{(p\rightarrow t) \mid (q\rightarrow r) \in \text{in}[s] \& (r\rightarrow t) \in \text{in}[s]\} \cup \\
&\quad (\text{in}[s] - \{(p\rightarrow x) \forall x\})
\end{align*}
```

- $s$: $\ast q = p$;
May Points-To Analysis (cont)

What is the flow function for the following statement?

- \( s: \ x = 3; \)
- \( \ast p = x; \)
- \( \text{out}[s] = \text{in}[s] \)

Dealing with Dynamically Allocated Memory

Issue
- Each allocation creates a new piece of storage
  \( e.g., \ p = \text{new} \ T \)

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

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

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: \(\text{p} = \text{new} \ T;\)
  out[s] = \{(p->stmt_s)\} \cup (\text{in}[s] - \{(p->x) \ \forall 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: \(\cap\)

Analogous flow functions
s: \(\text{p} = \&x;\)
out_must[s] = \{(p->x)\} \cup (\text{in}_\text{must}[s] - \{(p->x) \ \forall x\})
s: \(\text{p} = q;\)
out_must[s] = \{(p->t) | (q->t) \in \text{in}_\text{must}[s]\} \cup (\text{in}_\text{must}[s] - \{(p->x) \ \forall x\})
s: \(\text{p} = \*q;\)
out_must[s] = \{(p->t) | (q->r) \in \text{in}_\text{must}[s] \& (r->t) \in \text{in}_\text{must}[s]\} \cup (\text{in}_\text{must}[s] - \{(p->x) \ \forall x\})
s: \(\*p = q;\)
out_must[s] = \{(r->t) | (p->r) \in \text{in}_\text{must}[s] \& (q->t) \in \text{in}_\text{must}[s]\} \cup (\text{in}_\text{must}[s] - \{(r->x) \ \forall x\} \ (p->x) \in \text{in}_\text{must}[s])\}

Compute this along with may analysis
- Why?
Definiteness of Alias Information

Often need both

Consider liveness analysis

\[ *p = *q + 4; \]

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

May (possible) alias information

\[ \text{e.g., } \quad \text{if } (c) \ p = \&i; \]

Must (definite) alias information

\[ \text{e.g., } \quad p = \&i; \]

Using Points-To Information

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

To support constant propagation, first run points-to analysis

Then run constant propagation

The point

Since \(*p\) and \(x\) do not alias, \(x\) is constant in this last statement

Points 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
- \(s: \ x = y;\)
  \[
  \text{out}_{\text{reach}}[s] = \{(x, s)\} \cup (\text{in}_{\text{reach}}[s] - \{(x, t) \forall t\})
  \]
- \(s: \ x = *p;\)
  \[
  \text{out}_{\text{reach}}[s] = \{(x, s)\} \cup (\text{in}_{\text{reach}}[s] - \{(x, t) \forall t\})
  \]
- \(s: \ *p = x;\)
  \[
  \text{out}_{\text{reach}}[s] = \{(z, s) | (p \rightarrow z) \in \text{in}_{\text{must-pt}}[s]\} \cup (\text{in}_{\text{reach}}[s] - \{(y, t) \forall t | (p \rightarrow y) \in \text{in}_{\text{must-pt}}[s]\})
  \]
- . . .

Function Calls

```c
{ 
  int x, y, a;
  int *p;
  return p;
}

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

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)*

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

*p and i alias

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

```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)*

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

a[i] and a[j] alias
What Can Alias?

Stack storage and globals

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

Heap allocated objects

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

What Can Alias? (cont)

Arrays

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

Can \(c[i_1]\) and \(c[i_2]\) alias?

<table>
<thead>
<tr>
<th>Fortran</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Fortran array" /></td>
<td><img src="image" alt="Java array" /></td>
</tr>
</tbody>
</table>

```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** [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

**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