Introduction to Alias Analysis

**Last time**
- Partial Redundancy Elimination

**Today**
- Alias analysis

<table>
<thead>
<tr>
<th>transparent</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{b+c}</td>
<td>{b+c}</td>
<td>{b+c}</td>
</tr>
<tr>
<td>locally_available</td>
<td>{b+c}</td>
<td>{b+c}</td>
<td>{b+c}</td>
</tr>
<tr>
<td>locally_anticipated</td>
<td>{b+c}</td>
<td>{b+1}</td>
<td>{b+c}</td>
</tr>
<tr>
<td>available_in</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>available_out</td>
<td>{b+c}</td>
<td>{b+c}</td>
<td>{b+c}</td>
</tr>
<tr>
<td>partially_available_in</td>
<td></td>
<td></td>
<td>{b+c}</td>
</tr>
<tr>
<td>partially_available_out</td>
<td>{b+c}</td>
<td>{b+c}</td>
<td>{b+c}</td>
</tr>
<tr>
<td>anticipated_out</td>
<td>{b+c}</td>
<td>{b+c}</td>
<td>{b+c}</td>
</tr>
<tr>
<td>anticipated_in</td>
<td>{b+c}</td>
<td>{b+1}</td>
<td>{b+c}</td>
</tr>
<tr>
<td>pin</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pin</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>{b+c}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td></td>
<td></td>
<td>{b+c}</td>
</tr>
</tbody>
</table>
**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)
    
    ```c
    *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

```
{  
    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;  

    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;  
}  

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 \((p \rightarrow x)\) pairs if \(p\) points to \(x\)

Properties
- Use data-flow analysis
- May information (will look at must information next)

March 2, 2015
Introduction to Alias Analysis

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., \(\text{int **q}; p = *q;\)
May Points-To Analysis (Pointers to Pointers)

Additional flow functions
- $s$: $p = *q$:
  $$\text{out}[s] = \{(p \rightarrow t) \mid (q \rightarrow x) \in \text{in}[s] \land (r \rightarrow t) \in \text{in}[s]\} \cup (\text{in}[s] - \{(p \rightarrow x) \forall x\})$$

- $s$: $*q = p$:
  $$\text{out}[s] = \{(r \rightarrow t) \mid (q \rightarrow x) \in \text{in}[s] \land (p \rightarrow t) \in \text{in}[s]\} \cup (\text{in}[s] - \{(r \rightarrow x) \forall x \mid (q \rightarrow x) \in \text{in}_{\text{new}}[s]\})$$

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 name to represent each new allocation
  - $\text{newvar}$: Creates a new variable

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: $p = \text{new } T$;
  out$[s] = \{(p\rightarrow \text{stmt})\} \cup (\text{in}[s] - \{(p\rightarrow x) \forall x\})$
- 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
  - This dimension is often referred to as “heap naming”

Must Points-To Analysis

Meet function: $\bigcap$

Analogous flow functions
- s: $p = \&x$;
  out$\text{must}[s] = \{(p\rightarrow x)\} \cup (\text{in}_{\text{must}}[s] - \{(p\rightarrow x) \forall x\})$
- s: $p = q$;
  out$\text{must}[s] = \{(p\rightarrow t) \mid (q\rightarrow t) \in \text{in}_{\text{must}}[s]\} \cup (\text{in}_{\text{must}}[s] - \{(p\rightarrow x) \forall x\})$
- s: $p = *q$;
  out$\text{must}[s] = \{(p\rightarrow t) \mid (q\rightarrow r) \in \text{in}_{\text{must}}[s] \land (r\rightarrow t) \in \text{in}_{\text{must}}[s]\} \cup (\text{in}_{\text{must}}[s] - \{(p\rightarrow x) \forall x\})$
- s: $*p = q$;
  out$\text{must}[s] = \{(r\rightarrow t) \mid (p\rightarrow r) \in \text{in}_{\text{must}}[s] \land (q\rightarrow t) \in \text{in}_{\text{must}}[s]\} \cup (\text{in}_{\text{must}}[s] - \{(r\rightarrow *) \mid (p\rightarrow r) \in \text{in}_{\text{must}}[s]\})$

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

Often need both

- Consider liveness analysis

\[ \text{s: } *p = *q + 4; \]

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

May (possible) alias information

- Indicates what might be true
  
  \[ \text{e.g.,} \quad \text{if (c) } p = \&i; \]

Must (definite) alias information

- Indicates what is definitely true
  
  \[ \text{e.g.,} \quad p = \&i; \]

Using Points-To Information

To support constant propagation, first run points-to analysis

\[
\{ \begin{align*}
\text{int } x, y, a; \\
\text{int } *p, *q; \\
q &= \&x; \\
p &= \&a; \\
x &= 5; \\
*p &= 23; \\
y &= x + 1
\end{align*} \}
\]

\{ \begin{align*}
\{\text{\(q\mapsto x\)}\} \\
\{\text{\(p\mapsto a\)}\} \\
\{\text{\(q\mapsto x\)}, \text{\(p\mapsto a\)}\} \\
\{\text{\(q\mapsto x\)}, \text{\(p\mapsto a\)}\} \\
\{\text{\(q\mapsto x\)}, \text{\(p\mapsto a\)}\} \\
\{\text{\(q\mapsto x\)}, \text{\(p\mapsto a\)}\}
\end{align*} \}

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
– \(s: \ *p = x;\)
  \[\text{out}_{\text{reach}}[s] = \{(z,s) \mid (p \rightarrow z) \in \text{in}_{\text{may-quant}}[s]\} \cup \]

– \(s: \ x = \ *p;\)
  \[\text{out}_{\text{reach}}[s] = \{(x,s) \cup \text{in}_{\text{reach}}[s] - \{(x,t) \forall t}\} \cup \]

– . . .

Function Calls

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

```c
foo (int *p)
{
 int x, y, a;
 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;

Parameter passing by reference (e.g., in Pascal)
procedure procl(var a:integer; var b:integer);
... procl(x,x);
procl(x,glob);

Array indexing (e.g., in C)
int i,j, a[128];
i = j;

*p and i alias
a and b alias in body of procl
b and glob alias in body of procl
a[i] and a[j] alias
What Can Alias?

Stack storage and globals
void fun(int p1) {
    int i, j, temp;
    ... 
}

Heap allocated objects
n = new Node;
n->data = x;
n->next = new Node;
...

What Can Alias? (cont)

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

Can c[i1] and c[i2] alias?

Fortran

Java

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Representations of Aliasing

**Points-to pairs** [Emami 94]
- 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