Alias Analysis

Last time
  – Interprocedural analysis

Today
  – Intro to 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 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)
  int i,j, a[128];
  i = j;
  a[i] and a[j] alias

What Can Alias?

Stack storage and globals
  void fun(int pl) {
    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

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

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

<table>
<thead>
<tr>
<th>Fortran</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c)</td>
<td>(7)</td>
</tr>
</tbody>
</table>

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 \(d_3\) reach

Otherwise we must be very conservative

How hard is this problem?

Undecidable
- Landi 1992
- Ramalingan 1994

All solutions are conservative approximations

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.,
    \[
    p = 4a; \\
    a = 3; \quad b = 4; \\
    *q = 5;
    \]
- \(*q\) 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)\}
  - int **a, *b, c, *d, e;
  - 1: a = &b;
  - Completely general
  - 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.,
  \[
  \begin{align*}
  p &= &x; \\
  &\ldots \\
  p &= &y; \\
  \end{align*}
  \]

Flow-insensitive alias analysis
- Compute aliasing information for entire procedure
  - e.g.,
  \[
  \begin{align*}
  p &= &x; \\
  &\ldots \\
  p &= &y; \\
  \end{align*}
  \]

Definiteness of Alias Information

May (possible) alias information
- Indicates what might be true
  - e.g.,
  \[
  \begin{align*}
  &\text{if (c) } p = &i; \\
  \end{align*}
  \]

Must (definite) alias information
- Indicates what is definitely true
  - e.g.,
  \[
  \begin{align*}
  p &= &i; \\
  \end{align*}
  \]

Often need both
- e.g., Consider liveness analysis
  \[
  \begin{align*}
  &1: \text{ *p must alias } v \Rightarrow \text{ def}[s] = \text{ kill}[s] = \{v\} \\
  &2: \text{ *q may alias } v \Rightarrow \text{ use}[s] = \text{ gen}[s] = \{v\} \\
  s: \text{ *p = *q+4; }
  \end{align*}
  \]

Suppose out[s] = \{v\}.
Flow-sensitive May Points-To Analysis

Analogous flow functions
- $\cap$ is $\cup$
- $s$: $p \leftarrow x$
  $out[s] = \{(p \rightarrow x)\} \cup (in[s] - \{(p \rightarrow y) \forall y\})$
- $s$: $p \leftarrow q$
  $out[s] = \{(p \rightarrow t) | (q \rightarrow t) \in in[s]\} \cup (in[s] - \{(p \rightarrow y) \forall y\}))$
- $s$: $p \leftarrow *q$
  $out[s] = \{(r \rightarrow x) \in in[s] & (p \rightarrow t) \in in[s]\} \cup (in[s] - \{(p \rightarrow x) \forall x\})$
- $s$: $*p \leftarrow q$
  $out[s] = \{(r \rightarrow x) \in in[s] & (p \rightarrow t) \in in[s]\} \cup (in[s] - \{(r \rightarrow *) | (p \rightarrow x) \in in[s]\})$

Must Points-To Analysis

Analogous flow functions
- $\cap$ is $\cap$
- $s$: $p \leftarrow x$
  $out_{must}[s] = \{(p \rightarrow x)\} \cup (in_{must}[s] - \{(p \rightarrow x) \forall x\})$
- $s$: $p \leftarrow q$
  $out_{must}[s] = \{(p \rightarrow t) | (q \rightarrow t) \in in_{must}[s]\} \cup (in_{must}[s] - \{(p \rightarrow x) \forall x\})$
- $s$: $p \leftarrow *q$
  $out_{must}[s] = \{(r \rightarrow x) \in in_{must}[s] & (p \rightarrow t) \in in_{must}[s]\} \cup (in_{must}[s] - \{(p \rightarrow x) \forall x\})$
- $s$: $*p \leftarrow q$
  $out_{must}[s] = \{(r \rightarrow x) \in in_{must}[s] & (p \rightarrow t) \in in_{must}[s]\} \cup (in_{must}[s] - \{(r \rightarrow *) | (p \rightarrow x) \in in_{must}[s]\})$

Other Issues (Modeling the Heap)

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
  - $\text{newvar}$: Creates a new variable

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

Problem
- Domain is unbounded!
- Iterative data-flow analysis may not converge

Modeling the Heap (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
  $s$: $p = \text{new } T$
  $out[s] = \{(p \rightarrow \text{stmt})\} \cup (in[s] - \{(p \rightarrow x) \forall x\})$
  - 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
Using Alias Information

Example: reaching definitions
- Compute at each point in the program a set of \((s,v)\) pairs, indicating that statement \(s\) may define variable \(v\).

Flow functions
- \(s: *p = x;\)
  
  \[
  \text{out}_{\text{reach}}[s] = \{(s,x) \mid (p\rightarrow x) \in \text{in}_{\text{may}}[s]\} \cup \{s, y \mid (p\rightarrow y) \in \text{in}_{\text{may}}[s]\}
  \]
- \(s: x = \*p;\)
  
  \[
  \text{out}_{\text{reach}}[s] = \{(s,x)\} \cup \{s, y \mid (p\rightarrow y) \in \text{in}_{\text{must}}[s]\}
  \]
- ...

Function Calls

Question
- How do function calls affect our points-to sets?
  
  e.g., \(\text{p1} = \&x;\)
  
  \[
  \text{p2} = \&\text{p1};
  \]
  
  \[
  \ldots
  \]
  
  \[
  \text{foo()} ;
  \]

Be conservative
- Assume that any reachable pointer may be changed
- Pointers can be “reached” via globals and parameters
  - May pass through objects in the heap
  - Can be changed to anything reachable or something else
- Can we prune aliases using types?

Problem
- Lose a lot of information

Concepts

What is aliasing and how does it arise

Properties of alias analyses
- Definiteness: may or must
- Flow sensitivity: sensitive or insensitive
- Context sensitivity: sensitive or insensitive (interprocedural only)
- Representation: alias pairs, points-to sets

Function calls degrade alias information
- Context-sensitive interprocedural analysis

Next Time

Lecture
- Flow and context insensitive alias analysis