Pointer Analysis

CS 252r Fall 2015
Today: pointer analysis

• What is it? Why? Different dimensions
• Andersen analysis
• Steensgaard analysis
• Pointer analysis for Java
Pointer analysis

• What memory locations can a pointer expression refer to?

• Alias analysis: When do two pointer expressions refer to the same storage location?

• E.g.,
  
  ```
  int x;
  p = &x;
  q = p;
  ```

  • *p and *q alias,
  
  as do x and *p, and x and *q
• Aliasing can arise due to
  • Pointers
    • e.g., int *p, i;  p = &i;
  • Call-by-reference
    • void m(Object a, Object b) { … }
      m(x,x); // a and b alias in body of m
      m(x,y); // y and b alias in body of m
  • Array indexing
    • int i,j,a[100];
      i = j; // a[i] and a[j] alias
Why do we want to know?

- Pointer analysis tells us what memory locations code uses or modifies
- Useful in many analyses
- E.g., Available expressions
  - \(*p = a + b;\)
  - \(y = a + b;\)
  - If \(*p\) aliases \(a\) or \(b\), then second computation of \(a+b\) is not redundant
- E.g., Constant propagation
  - \(x = 3; \; *p = 4; \; y = x;\)
  - Is \(y\) constant? If \(*p\) and \(x\) do not alias each other, then yes. If \(*p\) and \(x\) always alias each other, then yes. If \(*p\) and \(x\) sometimes alias each other, then no.
Some dimensions of pointer analysis

• Intraprocedural / interprocedural
• Flow-sensitive / flow-insensitive
• Context-sensitive / context-insensitive
• Definiteness
  • May versus must
• Heap modeling
• Representation
Flow-sensitive vs flow-insensitive

- **Flow-sensitive** pointer analysis computes for each program point what memory locations pointer expressions may refer to.
- **Flow-insensitive** pointer analysis computes what memory locations pointer expressions may refer to, at any time in program execution.
- Flow-sensitive pointer analysis is (traditionally) too expensive to perform for whole program.
  - Flow-insensitive pointer analyses typically used for whole program analyses.
  - Recent work shows flow-sensitivity can scale: *Flow-sensitive pointer analysis for millions of lines of code* by Hardekopf and Lin, CGO 11.
Flow-sensitive pointer analysis is hard

<table>
<thead>
<tr>
<th>Alias Mechanism</th>
<th>Intraprocedural May Alias</th>
<th>Intraprocedural Must Alias</th>
<th>Interprocedural May Alias</th>
<th>Interprocedural Must Alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Formals, No Pointers, No Structures</td>
<td>-</td>
<td>-</td>
<td>Polynomial [1, 5]</td>
<td>Polynomial [1, 5]</td>
</tr>
<tr>
<td>Single level pointers, No Reference Formals, No Structures</td>
<td>Polynomial</td>
<td>Polynomial</td>
<td>Polynomial</td>
<td>Polynomial</td>
</tr>
<tr>
<td>Single level pointers, Reference Formals, No Pointer Reference Formals, No Structures</td>
<td>-</td>
<td>-</td>
<td>Polynomial</td>
<td>Polynomial</td>
</tr>
<tr>
<td>Multiple level pointers, No Reference Formals, No Structures</td>
<td>( \mathcal{NP} )-hard</td>
<td>Complement is ( \mathcal{NP} )-hard</td>
<td>( \mathcal{NP} )-hard</td>
<td>Complement is ( \mathcal{NP} )-hard</td>
</tr>
<tr>
<td>Single level pointers, Pointer Reference Formals, No Structures</td>
<td>-</td>
<td>-</td>
<td>( \mathcal{NP} )-hard</td>
<td>Complement is ( \mathcal{NP} )-hard</td>
</tr>
<tr>
<td>Single level pointers, Structures, No Reference Formals</td>
<td>( \mathcal{NP} )-hard [14]</td>
<td>Complement is ( \mathcal{NP} )-hard</td>
<td>( \mathcal{NP} )-hard [14]</td>
<td>Complement is ( \mathcal{NP} )-hard</td>
</tr>
</tbody>
</table>

Table 1: Alias problem decomposition and classification

*Pointer-induced Aliasing: A Problem Classification*, Landi and Ryder, POPL 1990
Context sensitivity

- Also difficult, but success in scaling up to hundreds of thousands LOC
  - BDDs see Whaley and Lam PLDI 2004
  - Doop, Bravenboer and Smaragdakis OOPSLA 2009
  - ...
Definiteness

- May analysis: aliasing that may occur during execution
  - (cf. must-not alias, although often has different representation)
- Must analysis: aliasing that must occur during execution
- Sometimes both are useful
  - E.g., Consider liveness analysis for *p = *q + 4;
  - If *p must alias x, then x in kill set for statement
  - If *q may alias y, then y in gen set for statement
Representation

• Possible representations
  • Points-to pairs: first element points to the second
    • e.g., (p → b), (q → b)
      *p and b alias, as do *q and b, as do *p and *q
  • Pairs that refer to the same memory
    • e.g., (*p,b), (*q,b), (*p,*q), (**r, b)
      • General, may be less concise than points-to pairs
  • Equivalence sets: sets that are aliases
    • e.g., {*p,*q,b}
Modeling memory locations

- We want to describe what memory locations a pointer expression may refer to
- How do we model memory locations?
  - For global variables, no trouble, use a single “node”
  - For local variables, use a single “node” per context
    - i.e., just one node if context insensitive
  - For dynamically allocated memory
    - Problem: Potentially unbounded locations created at runtime
    - Need to model locations with some finite abstraction
Modeling dynamic memory locations

• Common solution:
  • For each allocation statement, use one node per context
  • (Note: could choose context-sensitivity for modeling heap locations to be less precise than context-sensitivity for modeling procedure invocation)

• Other solutions:
  • One node for entire heap
  • One node for each type
  • Nodes based on analysis of “shape” of heap
• Let’s consider flow-insensitive may pointer analysis

• Assume program consists of statements of form
  • \( p = \&a \) (address of, includes allocation statements)
  • \( p = q \)
  • \( \*p = q \)
  • \( p = \*q \)

• Assume pointers \( p, q \in P \) and address-taken variables \( a, b \in A \) are disjoint
  • Can transform program to make this true
  • For any variable \( v \) for which this isn’t true, add statement \( p_v = \&a_v \), and replace \( v \) with \( \*p_v \)

• Want to compute relation \( \text{pts} : P \cup A \rightarrow 2^A \)
  • Essentially points to pairs
Andersen-style pointer analysis

- View pointer assignments as **subset constraints**
- Use constraints to propagate points-to information

<table>
<thead>
<tr>
<th>Constraint type</th>
<th>Assignment</th>
<th>Constraint</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td><code>a = &amp;b</code></td>
<td><code>a \supseteq \{b\}</code></td>
<td><code>loc(b) \in pts(a)</code></td>
</tr>
<tr>
<td>Simple</td>
<td><code>a = b</code></td>
<td><code>a \supseteq b</code></td>
<td><code>pts(a) \supseteq pts(b)</code></td>
</tr>
<tr>
<td>Complex</td>
<td><code>a = *b</code></td>
<td><code>a \supseteq *b</code></td>
<td><code>\forall v \in pts(b). pts(a) \supseteq pts(v)</code></td>
</tr>
<tr>
<td>Complex</td>
<td><code>*a = b</code></td>
<td><code>*a \supseteq b</code></td>
<td><code>\forall v \in pts(a). pts(v) \supseteq pts(b)</code></td>
</tr>
</tbody>
</table>
Andersen-style pointer analysis

• Can solve these constraints directly on sets $\text{pts}(p)$

\begin{align*}
p &= & & \&a; & p \supseteq \{a\} \\
q &= & & p; & q \supseteq p \\
p &= & & \&b; & p \supseteq \{b\} \\
r &= & & p; & r \supseteq p
\end{align*}

\begin{align*}
\text{pts}(p) &= & & \{a, \; b\} \\
\text{pts}(q) &= & & \emptyset, \; b \\
\text{pts}(r) &= & & \emptyset, \; b \\
\text{pts}(a) &= & & \emptyset \\
\text{pts}(b) &= & & \emptyset
\end{align*}
Another example

- Can solve these constraints directly on sets $pts(p)$

<table>
<thead>
<tr>
<th>Variable</th>
<th>Constraint</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p = &amp;a$</td>
<td>$p \supseteq {a}$</td>
<td>$pts(p) = {a}$</td>
</tr>
<tr>
<td>$q = &amp;b$</td>
<td>$q \supseteq {b}$</td>
<td>$pts(q) = {b}$</td>
</tr>
<tr>
<td>$*p = q;$</td>
<td>$*p \supseteq q$</td>
<td></td>
</tr>
<tr>
<td>$r = &amp;c;$</td>
<td>$r \supseteq {c}$</td>
<td>$pts(r) = {c}$</td>
</tr>
<tr>
<td>$s = p;$</td>
<td>$s \supseteq p$</td>
<td>$pts(s) = \emptyset$</td>
</tr>
<tr>
<td>$t = *p;$</td>
<td>$t \supseteq *p$</td>
<td></td>
</tr>
<tr>
<td>$*s = r;$</td>
<td>$*s \supseteq r$</td>
<td>$pts(t) = {b}, c$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$pts(a) = {b}, c$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$pts(b) = \emptyset$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$pts(c) = \emptyset$</td>
</tr>
</tbody>
</table>
How precise?

\[ p = &a \]
\[ q = &b \]
\[ *p = q; \]
\[ r = &c; \]
\[ s = p; \]
\[ t = *p; \]
\[ *s = r; \]

\[ \text{pts}(p) = \{a\} \]
\[ \text{pts}(q) = \{b\} \]
\[ \text{pts}(r) = \{c\} \]
\[ \text{pts}(s) = \{a\} \]
\[ \text{pts}(t) = \{b,c\} \]
\[ \text{pts}(a) = \{b,c\} \]
\[ \text{pts}(b) = \emptyset \]
\[ \text{pts}(c) = \emptyset \]
Andersen-style as graph closure

- Can be cast as a graph closure problem
- One node for each \( \text{pts}(p) \), \( \text{pts}(a) \)

<table>
<thead>
<tr>
<th>Assgmt.</th>
<th>Constraint</th>
<th>Meaning</th>
<th>Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = &amp; b )</td>
<td>( a \supseteq {b} )</td>
<td>( b \in \text{pts}(a) )</td>
<td>no edge</td>
</tr>
<tr>
<td>( a = b )</td>
<td>( a \supseteq b )</td>
<td>( \text{pts}(a) \supseteq \text{pts}(b) )</td>
<td>( b \rightarrow a )</td>
</tr>
<tr>
<td>( a = \ast b )</td>
<td>( a \supseteq \ast b )</td>
<td>( \forall v \in \text{pts}(b). \text{pts}(a) \supseteq \text{pts}(v) )</td>
<td>no edge</td>
</tr>
<tr>
<td>( \ast a = b )</td>
<td>( \ast a \supseteq b )</td>
<td>( \forall v \in \text{pts}(a). \text{pts}(v) \supseteq \text{pts}(b) )</td>
<td>no edge</td>
</tr>
</tbody>
</table>

- Each node has an associated points-to set
- Compute transitive closure of graph, and add edges according to complex constraints
Workqueue algorithm

• Initialize graph and points to sets using base and simple constraints
• Let $W = \{ v \mid \text{pts}(v) \neq \emptyset \}$ (all nodes with non-empty points to sets)

While $W$ not empty
  • $v \leftarrow$ select from $W$
  • for each $a \in \text{pts}(v)$ do
    • for each constraint $p \supseteq *v$
      ‣ add edge $a \rightarrow p$, and add $a$ to $W$ if edge is new
    • for each constraint $*v \supseteq q$
      ‣ add edge $q \rightarrow a$, and add $q$ to $W$ if edge is new
  • for each edge $v \rightarrow q$ do
    • $\text{pts}(q) = \text{pts}(q) \cup \text{pts}(v)$, and add $q$ to $W$ if $\text{pts}(q)$ changed
Same example, as graph

\begin{align*}
p &= \&a & p \supseteq \{a\} \\
q &= \&b & q \supseteq \{b\} \\
*p &= q; & *p \supseteq q \\
r &= \&c; & r \supseteq \{c\} \\
s &= p; & s \supseteq p \\
t &= *p; & t \supseteq *p \\
*s &= r; & *s \supseteq r
\end{align*}

\underline{W:} p q r s a

- Initialize graph and points to sets using base and simple constraints
- Let $W = \{ v \mid \text{pts}(v) \neq \emptyset \}$ (all nodes with non-empty points to sets)
- While $W$ not empty
  - $v \leftarrow$ select from $W$
  - for each $a \in \text{pts}(v)$ do
    - for each constraint $p \supseteq ^*v$
      - add edge $a \rightarrow p$, and add $a$ to $W$ if edge is new
    - for each constraint $*v \supseteq q$
      - add edge $q \rightarrow a$, and add $q$ to $W$ if edge is new
    - for each edge $v \rightarrow q$ do
      - $\text{pts}(q) = \text{pts}(q) \cup \text{pts}(v)$, and add $q$ to $W$ if $\text{pts}(q)$ changed
Same example, as graph

\[
\begin{align*}
  p &= \&a \\
  q &= \&b \\
  *p &= q; \\
  r &= \&c; \\
  s &= p; \\
  t &= *p; \\
  *s &= r;
\end{align*}
\]

- Initialize graph and points to sets using base and simple constraints
- Let \( W = \{ v | \text{pts}(v) \neq \emptyset \} \) (all nodes with non-empty points to sets)
- While \( W \) not empty
  - \( v \leftarrow \text{select from } W \)
  - for each \( a \in \text{pts}(v) \) do
    - for each constraint \( p \supseteq *v \)
      - add edge \( a \rightarrow p \), and add \( a \) to \( W \) if edge is new
    - for each constraint \( *v \supseteq q \)
      - add edge \( q \rightarrow a \), and add \( q \) to \( W \) if edge is new
    - for each edge \( v \rightarrow q \) do
      - \( \text{pts}(q) = \text{pts}(q) \cup \text{pts}(v) \), and add \( q \) to \( W \) if \( \text{pts}(q) \) changed
Cycle elimination

- Andersen-style pointer analysis is $O(n^3)$, for number of nodes in graph (Actually, quadratic in practice [Sridharan and Fink, SAS 09])
  - Improve scalability by reducing $n$
- Cycle elimination
  - Important optimization for Andersen-style analysis
  - Detect strongly connected components in points-to graph, collapse to single node
    - Why? All nodes in an SCC will have same points-to relation at end of analysis
  - How to detect cycles efficiently?
    - Some reduction can be done statically, some on-the-fly as new edges added
Steensgaard-style analysis

- Also a constraint-based analysis
- Uses **equality constraints** instead of subset constraints
  - Originally phrased as a type-inference problem
- Less precise than Andersen-style, thus more scalable

<table>
<thead>
<tr>
<th>Constraint type</th>
<th>Assignment</th>
<th>Constraint</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>a = &amp;b</td>
<td>a ⊇ {b}</td>
<td>loc(b) ∈ pts(a)</td>
</tr>
<tr>
<td>Simple</td>
<td>a = b</td>
<td>a = b</td>
<td>pts(a) = pts(b)</td>
</tr>
<tr>
<td>Complex</td>
<td>a = *b</td>
<td>a = *b</td>
<td>∀v ∈ pts(b). pts(a) = pts(v)</td>
</tr>
<tr>
<td>Complex</td>
<td>*a = b</td>
<td>*a = b</td>
<td>∀v ∈ pts(a). pts(v) = pts(b)</td>
</tr>
</tbody>
</table>
Implementing Steensgaard-style analysis

• Can be efficiently implemented using Union-Find algorithm
  • Nearly linear time: $O(n\alpha(n))$
  • Each statement needs to be processed just once
• Different languages use pointers differently

• *Scaling Java Points-To Analysis Using SPARK* Lhotak & Hendren CC 2003
  • Most C programs have many more occurrences of the address-of (&) operator than dynamic allocation
    • & creates stack-directed pointers; malloc creates heap-directed pointers
  • Java allows no stack-directed pointers, many more dynamic allocation sites than similar-sized C programs
  • Java strongly typed, limits set of objects a pointer can point to
    • Can improve precision
  • Call graph in Java depends on pointer analysis, and vice-versa (in context sensitive pointer analysis)
  • Dereference in Java only through field store and load
  • And more…
    • Larger libraries in Java, more entry points in Java, can’t alias fields in Java, ...
Object-sensitive pointer analysis

  - Context-sensitive interprocedural pointer analysis
  - For context, use stack of receiver objects
  - (More next week?)

- Lhotak and Hendren. *Context-sensitive points-to analysis: is it worth it?* CC 06
  - Object-sensitive pointer analysis more precise than call-stack contexts for Java
  - Likely to scale better
Closing remarks

• Pointer analysis: important, challenging, active area
  • Many clients, including call-graph construction, live-variable analysis, constant propagation, …
  • Inclusion-based analyses (aka Andersen-style)
  • Equality-based analyses (aka Steensgaard-style)

• Requires a tradeoff between precision and efficiency
  • Ultimately an empirical question. Which clients, which code bases?

• Recent results promising
  • Scalable flow-sensitivity (Hardekopf and Lin, POPL 09)
  • Context-sensitive Andersen-style analyses seem scalable (See Thurs)

• Other issues/questions (see Hind, PASTE’01)
  • How to measure/compare pointer analyses? Different clients have different needs
  • Demand-driven analyses? May be more precise/scalable…