Lecture 2. Pointer Analysis

Wei Le

2016.1
Agenda

- Terms and concepts
- Algorithms: Andersen-Style and Steensgaard-Style
- Advanced topics
What is Pointer Analysis? [Hind2001]

**Pointer analysis** statically determines:

- the possible runtime values of a pointer
- what storage locations a pointer can point to
- there are certain models can represent the storage locations:
  - In Microsoft Phoenix, memory locations are labeled with tags (numbers), each tag represents an object
  - an entire heap is an object
What is Alias Analysis? [Hind2001]

**Alias analysis or points-to analysis**

- statically determines when two expressions (e.g., two pointer variables, or memory references) refer to the same memory location (global, stack storage, heap)
- produces alias (points-to) relations

**Example:** int x; p = &x; q = p;

**Representation:**

- alias pairs: *p and *q, *p and x, *q and x [Shapiro&Horwitz97]
- equivalence set: {*p, x, *q}
- points to pairs: p → x, q → x [Emami94]

*Note: pointer analysis, alias analysis, points-to analysis often are used interchangeably*
May and Must Aliasing

- May aliasing: aliasing that may occur during execution (e.g., if (c) p = &i)
- Must aliasing: aliasing that must occur during execution (e.g., p = &i)

Easiest alias analysis: nothing must alias, everything may alias
Client Problems

- Compiler optimization (conservative)
- Bug finding (don’t have to be conservative)

Example

1. \( X = 5 \)
2. \( *p = \ldots \) // p may or may not point to X
3. \( \ldots = X \)

**Constant propagation:** assume \( p \) does point to \( X \) (i.e., in statement 3, \( X \) cannot be replaced by 5).

**Dead Code Elimination:** assume \( p \) does not point to \( X \) (i.e., statement 1 cannot be deleted).
How Hard Is This Problem?

- Undecidable [Landi1992] [Ramalingan1994]
- Approximation algorithms, worst-case complexity, range from almost linear to doubly exponential [Hind2001]
- Research started in late 1970s.
Andersen-style and Steensgaard-style Analyses

**Flow-insensitive, Context-insensitive** pointer analysis:

- Control flow information is not used, the order of statements is not considered
- Compute solutions (points-to information) for the entire function rather than at a program point
  - what are the alias pairs for the function? v.s.
  - what are the alias pairs at this program point?
- Calling context is not considered – it does not matter where the function is invoked in the program, the alias information computed for the function is the same
Andersen-Style Pointer Analysis [Andersen1994]

- Flow-insensitive, context-insensitive analysis, first for C programs (1994) later for Java
- View pointer assignments as *subset constraints*: base – pointer initialization; simple – variable names, complex – pointer dereference

<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>
Andersen-Style Pointer Analysis

Example: Flow-Sensitive v.s Flow-Insensitive Solutions
Andersen-Style Pointer Analysis

Basic idea:

- map to subset constraints
- construct the *constraint graphs*
- compute transitive closure to propagate points-to relations along the edges of the constraint graphs

Constraint graph:

- one node for each variable representing its points-to set, e.g., \( \text{pts}(p) \), \( \text{pts}(a) \)
- one directed edge for certain constraint
Andersen-Style Pointer Analysis: Constructing Constraint Graphs

<table>
<thead>
<tr>
<th>Assgnt.</th>
<th>Constraint</th>
<th>Meaning</th>
<th>Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = &amp;b</td>
<td>a ⊇ {b}</td>
<td>b ∈ pts(a)</td>
<td>no edge</td>
</tr>
<tr>
<td>a = b</td>
<td>a ⊇ b</td>
<td>pts(a) ⊇ pts(b)</td>
<td>b → a</td>
</tr>
<tr>
<td>a = *b</td>
<td>a ⊇ *b</td>
<td>∀v ∈ pts(b). pts(a) ⊇ pts(v)</td>
<td>no edge</td>
</tr>
<tr>
<td>*a = b</td>
<td>*a ⊇ b</td>
<td>∀v ∈ pts(a). pts(v) ⊇ pts(b)</td>
<td>no edge</td>
</tr>
</tbody>
</table>
Andersen-Style Pointer Analysis: Constructing Constraint Graphs

Two Examples
Andersen-Style Pointer Analysis

- 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
Andersen-Style Pointer Analysis

Example
Andersen-style analysis: Algorithm Analysis

- Can be reduced to computing the transitive closure of a dynamic graph
  - *dynamic graph*: the graph changes over the analysis of the program
  - *the transitive closure* of a directed acyclic graph (DAG) is the reachability relation of the DAG. (graph: a set of nodes, and binary relations among the nodes)

- A well-studied problem for which the best known complexity is $O(n^3)$ (n is the number of node)
Andersen-Style Pointer Analysis: 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
Andersen-Style Pointer Analysis: Cycle Elimination
Steensgaard-Style Pointer Analysis
[Steensgaard1996POPL]

Points-to Analysis in almost linear time

- Uses equality constraints instead of subset constraints
- Unification based approach: assignment unifies the graph nodes, e.g., $x = y$ (unified $x$ and $y$ in the same node), also called union-find algorithm, exclusion-based approaches, nearly linear complexity $O(n \cdot \alpha(n))$, where $\alpha(n)$ is the inverse Ackermann's function, $\alpha(2^{132}) < 4$
- Less precise than Andersen-style, thus more scalable
Steensgaard-Style Pointer Analysis

Key idea: maintain a set of disjoint sets and supports two operations:

- **FIND(x)**: return the set containing x
- **UNION(x, y)**: union the two sets containing x and y
merge(x, y)
{
    x = FIND(x); y = FIND(y);
    if (x == y) then return;
    UNION(x, y);
    merge(points-to(x), points-to(y));
}

for each constraint LHS = RHS
    merge(LHS, RHS)
Examples
Andersen vs. Steensgaard (Points-To Graph)

That is, in Andersen's Algorithm we might have

\[ \text{p} \rightarrow \text{a} \]

\[ \text{b} \]

In Steensgaard's Algorithm we would instead have

\[ \text{p} \rightarrow \text{a} \]

\[ \text{b} \]

In effect any two locations that might be pointed to by the same pointer are placed in a single equivalence class.
Andersen vs. Steensgaard

Steensgaard's Algorithm is sometimes less accurate than Andersen's Algorithm. For example, the following points-to graph, created by Andersen's Algorithm, shows that p may point to a or b whereas q may only point to a:

![Diagram showing points-to relationships between p, a, b, and q]

In Steensgaard's Algorithm we get

![Diagram showing points-to relationships between p, a, b, and q]

incorrectly showing that if p may point to a or b then so may q.
Andersen vs. Steensgaard

- Horwitz and Shapiro examined 61 C programs, ranging in size from 300 to 24,300 lines.

- As expected, Steensgaard is less precise: On average points-to sets are 4 times bigger; at worst 15 times bigger.

- As expected, Andersen is slower. On average 1.5 times slower: at worst 31 times slower.

- Both are much better than the naive “address taken” approach.

- Bottom line: Use Andersen for small programs, use Steensgaard (or something else) for large programs.
## Andersen vs. Steensgaard

<table>
<thead>
<tr>
<th>Name</th>
<th>Size (LoC)</th>
<th>Andersen (sec)</th>
<th>Steensgaard (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>triangle</td>
<td>1986</td>
<td>2.9</td>
<td>0.8</td>
</tr>
<tr>
<td>gzip</td>
<td>4584</td>
<td>1.7</td>
<td>1.1</td>
</tr>
<tr>
<td>li</td>
<td>6054</td>
<td>738.5</td>
<td>4.7</td>
</tr>
<tr>
<td>bc</td>
<td>6745</td>
<td>5.5</td>
<td>1.6</td>
</tr>
<tr>
<td>less</td>
<td>12152</td>
<td>1.9</td>
<td>1.5</td>
</tr>
<tr>
<td>make</td>
<td>15564</td>
<td>260.8</td>
<td>6.1</td>
</tr>
<tr>
<td>tar</td>
<td>18585</td>
<td>23.2</td>
<td>3.6</td>
</tr>
<tr>
<td>espresso</td>
<td>22050</td>
<td>1373.6</td>
<td>10.2</td>
</tr>
<tr>
<td>screen</td>
<td>24300</td>
<td>514.5</td>
<td>10.1</td>
</tr>
</tbody>
</table>

75MHz SuperSPARC, 256MB RAM

[Shapiro-Horwitz POPL’97]
Points-to Analyses Work in Real DataFlow Problems?

In "Which Pointer Analysis Should I Use," Hind and Pioli survey the effectiveness of a number of points-to analyses in actual data flow analyses (mod/ref, liveness, reaching defs, interprocedural constant propagation).

Their conclusions are essentially the same across all these analyses:

- Steensgaard's analysis is significantly more precise than address-taken analysis and not significantly slower.

- Andersen's analysis produces modest, but consistent, improvements over Steensgaard's analysis.

- Both context-sensitive points-to analysis and flow-sensitive points-to analysis give little improvement over Andersen's analysis.
Andersen vs. Steensgaard: Summary

- Both are flow-insensitive and context-insensitive
- Differ in points-to set construction
  - Andersen-style: many out edges, one variable per node
  - Steensgaard-style: one out edge, many variables per node

- Andersen-style: inclusion-based, subset-based – the slowest but most precise flow-insensitive algorithm
- Steensgaard-style: equality-based, unification-based – the fastest but least precise
Modification of Anderson-Style and Steensgaard

The Horwitz-Shapiro Approach: 1997 POPL – Fast and Accurate Flow-Insensitive Points-To Analysis
Horwitz and Shapiro suggest each node in the points-to graph be limited to out degree k, where \(1 \leq k \leq n\).

If \(k = 1\) then they have Steensgaard's approach.

If \(k = n\) (n is number of nodes in points-to graph), then they have Andersen's approach.

Their worst case run-time is \(O(k^2 \cdot n)\), which is not much worse than Steensgaard if \(k\) is kept reasonably small.
Modification of Anderson-Style and Steensgaard

To use their approach assign each variable that may be pointed to to one of \( k \) categories.

Now if \( p \) may point to \( x \) and \( p \) may also point to \( y \), we merge \( x \) and \( y \) **only if** they both are in the same category.

If \( x \) and \( y \) are in different categories, they **aren't** merged, leading to more accurate points-to estimates.
Modification of Anderson-Style and Steensgaard

Example

\[ p_1 = &a; \]
\[ p_1 = &b; \]
\[ p_1 = &c; \]
\[ p_2 = &c; \]

Say we have \( k = 2 \) and place \( a \) and \( b \) in category 1 and \( c \) in category 2.

We then build:

This points-to graph is just as accurate as that built by Andersen's approach.
Modification of Anderson-Style and Steensgaard

But...

What if we chose to place $a$ in category 1 and $b$ and $c$ in category 2.
We now have:

This graph is inexact, since it tells us $p_2$ may point to $b$, which is false.
(Steensgaard would have been worse still, incorrectly telling us $p_2$ may point to $a$ as well as $b$ and $c$).
Modification of Anderson-Style and Steensgaard

What if we ran Shapiro and Horwitz's points-to analysis twice, each with different category assignments?

Each run may produce a different points-to graph. One may say p2 points to b whereas the other says it can't.

Which do we believe?

Neither analysis misses a genuine points-to relation. Rather, merging of nodes sometimes creates false points-to information.

So we will believe p2 may point to b only if all runs say so.

This means multiple runs may "filter out" many of the false points-to relations caused by merging.
How Many Runs are Needed?

How are Categories to be Set?

We want to assign categories so that during at least one run, any pair of pointed-to variables are in different categories.

This guarantees that if all the runs tell us \( p \) may point to \( a \) and \( b \), it is not just because \( a \) and \( b \) always happened to be assigned the same category.

To force different category assignments for each pair of variables, we assign each pointed-to variable an index and write that index in base \( k \) (the number of categories chosen).
Modification of Anderson-Style and Steensgaard

For example, if we had variables a, b, c and d, and chose k = 2, we’d use the following binary indices:

a 00
b 01
c 10
d 11

Note that the number of base k digits needed to represent indices from 0 to n–1 is just ceiling(\(\log_k n\)).

This number is just the number of runs we need!
Modification of Anderson-Style and Steensgaard

Why?
In the first run, we'll use the right most digit in a variable's index as its category.
In the next run, we'll use the second digit from the right, then the third digit from the right, ...
Any two distinct variables have different index values, so they must differ in at least digit position.
Modification of Anderson-Style and Steensgaard

Returning to our example,
\[
\begin{align*}
  a & \ 00 \\
  b & \ 01 \\
  c & \ 10 \\
  d & \ 11 \\
\end{align*}
\]

On run #1 we give \( a \) and \( c \) category 0 and \( b \) and \( d \) category 1.

On run #2, \( a \) and \( b \) get category 0 and \( c \) and \( d \) get category 1.

So using just 2 runs in this simple case, we eliminate much of the inaccuracy Steensgaard's merging introduces.

Run time is now \( O(\log_k(n) \ k^2 \ n) \).
Modification of Anderson-Style and Steensgaard

How Well does this Approach Work?

On 25 tests, using 3 categories, Horwitz & Shapiro points-to-sets on average are 2.67 larger than those of Andersen (Steensgaard's are 4.75 larger).

This approach is slower than Steensgaard but on larger programs it is 7 to 25 times faster than Andersen.
Advanced Topics

Paper: Pointer Analysis: Haven’t We Solved This Problem Yet?
Goal: determine the set of objects pointed to by a reference variable or a reference object field.

Different languages use pointers differently:

- Most C programs have many more occurrences of the address-of (&) operator than dynamic allocation.
- 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)
- Larger libraries in Java, more entry points in Java
- ...
Object-Sensitive Pointer Analysis

Context-Sensitivity:

- **Context-sensitivity:** call site, program statement of method invocation
  
  \[ S: \text{this} \rightarrow \text{call\_method()} \]

- **Object sensitivity:** receiving object of method invocation (state of the object is not merged)
  
  \[ S: \text{this} \rightarrow \text{call\_method()} \]
Modeling Memory Locations

- For global variables, use a single node
- For local variables, use a single node per context, 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

- For each allocation statement, use one node per context
- One node for entire heap
- One node for each type
- Nodes based on the shape of the heap
Recursive Data Structure

- 1-level: combine all elements as one big cell [Emami’93, Wilson et al’95]
- k-limiting: distinguish first k-elements (k is an arbitrary number) [Landi et al’92, Cheng et al’00]
- beyond k-limiting: consider all elements, using some kinds symbolic access paths [Deutch’94]
- shape analysis: not only distinguish all elements, but also tell how they are connected (connection analysis) and what their shape is. [Ghiya et al’95]
Shape Analysis

Goal:
- Static code analysis that discovers and verifies properties of linked, dynamically allocated data structures in (usually imperative) computer programs, e.g., discriminating between cyclic and acyclic lists and proving that two data structures cannot access the same piece of memory.
- Identify may-alias relationships
- $x$ points to an acyclic list, cyclic list, tree, dag, ...
- **disjointedness** properties: $x$ and $y$ point to structures that do not share cells
- show that data-structure invariants hold

Applications:
- Verification: detecting memory leaks, proving the absence of mutual exclusive and deadlock
- Optimization: parallel programs

History:
- Originally formulated by Jones and Muchnick, 1981
- Mooly Sagiv, Tom Reps and Reinhard Wihelm, since 1995