Pointer Analysis 2

Wei Le
Outline

- What it is?
- Why it is important?
- How hard is the problem?
- Classical algorithms: Andersen-Style and Steensgaard-Style
- Pointer analysis in Java
What is pointer analysis and what is alias analysis?

- **Pointer analysis** aims to determine what memory location a pointer points to?
  
  *Example:* In Microsoft Phoenix, memory locations are labeled with tags, and each pointer is associated with a tag.

- **Alias analysis** aims to determine when the two variables refer to the same memory/storage location.
  
  *Example:* `int x; p = &x; q = p; alias pairs: *p and *q, *p and x, *q and x`
Aliasing can arise due to:

- Pointers:
  e.g., int *p, i; p = &i;

- Call-by-reference:
  e.g, void m(Object a, Object b) m(x,x); // a and b alias in body of m

- Array indexing:
  e.g, int i,j,a[100]; i = j; // a[i] and a[j] alias
Alias Analysis

- Must-alias and maybe-alias (may-alias)
  - Aliasing that must occur during execution
  - Aliasing that may occur during execution

- Representation:
  - Complete alias pairs: store all alias pairs explicitly, such as in [Landi et al.’92].
  - Compact alias pairs: store only basic alias pairs, and derive new alias pairs by dereference, transitivity and commutativity, such as in [Choi et al.’93].
  - Points-to relations: points-to pairs to indicate one variable is pointing to another, such as in [Emmai’93]
  - Equivalence sets: sets that are aliases

- An example: ”q=&p; p=&i; r=&i;”
  - Complete: \(\langle *q, p \rangle \langle *p, i \rangle \langle *r, i \rangle \langle * * q, *p \rangle \langle * * q, i \rangle \langle *p, *r \rangle \langle * * q, *r \rangle\)
  - Compact: \(\langle *q, p \rangle \langle *p, i \rangle \langle *r, i \rangle\)
  - Points-to: \((q, p)(p, i)(r, i)\)
  - Equivalence set: \{*p, i, *r\}
Why Alias Analysis Is Important?

Compiler optimization:

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).

Bug detection: we need to precisely track the relations, values and ranges of variables

......
How Hard Is This Problem? [3]

Undecidable
- Landi 1992
- Ramalingan 1994

All solutions are conservative approximations

Is this problem solved?
- Why haven’t we solved this problem? [Hind 2001]
- Still a number of open issues
  - large programs
  - partial programs
  - modeling the heap (shape analysis)
  - ...

How Hard Is This Problem?

- Research started in late 1970s.
- Moderate alias analysis implemented in commercial compilers, like gcc and DEC.
- Aggressive alias analysis still in research stage.
- Classifications of existing alias analysis:
  - Intra-procedural: isolated functions only.
  - Inter-procedural: consider function calls and their interactions;
  - Type-based techniques: use type information to decide alias.
  - Flow-based techniques: consider alias generated by flow of statement.
How Hard Is This Problem?

- Function calls: will formal parameters and global variables be changed?
- Function pointers: how to resolve function pointers?
- Aggregate data structure: how to handle struct and union?
- Heap objects: how to include heap object?
- Recursive data structure: will it lead to infinite number of alias?
- Pointer arithmetic: will points-to relations change due to pointer arithmetic?
- Type casting: what to do if a variable is type-casted?
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
## Handling `struct`

Two approaches:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Field-independent</th>
<th>Field-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.x=&amp;z;</td>
<td>assign to A</td>
<td>assign to x</td>
</tr>
<tr>
<td>p=A.x;</td>
<td>p gets &amp;z</td>
<td>p gets &amp;z</td>
</tr>
<tr>
<td>q=A.y;</td>
<td>q gets &amp;z</td>
<td>---</td>
</tr>
<tr>
<td>r=B.x;</td>
<td>---</td>
<td>r gets &amp;z</td>
</tr>
<tr>
<td>s=B.y;</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
Andersen-Style Pointer Analysis [1]

- Flow-insensitive, context-insensitive analysis, first for C programs (1994) later for Java
- View pointer assignments as *subset constraints*, also called inclusion based algorithms
- 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>a = &amp;b</td>
<td>a $\supseteq$ {b}</td>
<td>loc(b) $\in$ pts(a)</td>
</tr>
<tr>
<td>Simple</td>
<td>a = b</td>
<td>a $\supseteq$ b</td>
<td>pts(a) $\supseteq$ pts(b)</td>
</tr>
<tr>
<td>Complex</td>
<td>a = *b</td>
<td>a $\supseteq$ *b</td>
<td>$\forall v \in$ pts(b). pts(a) $\supseteq$ pts(v)</td>
</tr>
<tr>
<td>Complex</td>
<td>*a = b</td>
<td>*a $\supseteq$ b</td>
<td>$\forall v \in$ pts(a). pts(v) $\supseteq$ pts(b)</td>
</tr>
</tbody>
</table>
Andersen-Style Pointer Analysis: An Example

<table>
<thead>
<tr>
<th>Program</th>
<th>Constraints</th>
<th>Points-to Relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>a := b</td>
<td>a \supseteq { b, d }</td>
<td>a \rightarrow { b, d }</td>
</tr>
<tr>
<td>c := a</td>
<td>c \supseteq a</td>
<td>c \rightarrow { b, d }</td>
</tr>
<tr>
<td>a := d</td>
<td>e \supseteq a</td>
<td>e \rightarrow { b, d }</td>
</tr>
<tr>
<td>e := a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We’ve reached a fixed point
Andersen-Style Pointer Analysis

- Can be cast as a graph closure problem: one node for each $\text{pts}(p)$, $\text{pts}(a)$
- Each node has an associated points-to set
- Compute transitive closure of graph, and add edges according to complex constraints

<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>$a = *b$</td>
<td>$a \supseteq *b$</td>
<td>$\forall v \in \text{pts}(b). \text{pts}(a) \supseteq \text{pts}(v)$</td>
<td>no edge</td>
</tr>
<tr>
<td>$*a = b$</td>
<td>$*a \supseteq b$</td>
<td>$\forall v \in \text{pts}(a). \text{pts}(v) \supseteq \text{pts}(b)$</td>
<td>no edge</td>
</tr>
</tbody>
</table>
Andersen-Style Pointer Analysis: 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
Andersen-Style Pointer Analysis: Workqueue Algorithm

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

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

W: \ p \ q \ r \ s \ a
Andersen-Style Pointer Analysis: Workqueue Algorithm

\[
\begin{align*}
p &= \&a \\
q &= \&b \\
*p &= q; \\
r &= \&c; \\
s &= p; \\
t &= *p; \\
*s &= r; \\
p \supseteq \{a\} \\
q \supseteq \{b\} \\
*p \supseteq q \\
r \supseteq \{c\} \\
s \supseteq p \\
t \supseteq *p \\
*s \supseteq r
\end{align*}
\]
Andersen-Style Pointer Analysis: 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
Andersen-Style Pointer Analysis: Cycle Elimination

---

[Diagram of a directed graph showing nodes labeled 'e', 'a', 'b', 'c', 'd' and 'a,b,c,d']
Steensgaard-Style Pointer Analysis [5]

- Uses equality constraints instead of subset constraints (1996)
- 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
- 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>$a \supseteq {b}$</td>
<td>$\text{loc}(b) \in \text{pts}(a)$</td>
</tr>
<tr>
<td>Simple</td>
<td>$a = b$</td>
<td>$a = b$</td>
<td>$\text{pts}(a) = \text{pts}(b)$</td>
</tr>
<tr>
<td>Complex</td>
<td>$a = *b$</td>
<td>$a = *b$</td>
<td>$\forall v \in \text{pts}(b). , \text{pts}(a) = \text{pts}(v)$</td>
</tr>
<tr>
<td>Complex</td>
<td>$*a = b$</td>
<td>$*a = b$</td>
<td>$\forall v \in \text{pts}(a). , \text{pts}(v) = \text{pts}(b)$</td>
</tr>
</tbody>
</table>
Steensgaard-Style Pointer Analysis: An Example

Program

\[
\begin{align*}
& a := &b \\
& c := &a \\
& a := &d \\
& e := &a
\end{align*}
\]

Constraints

\[
\begin{align*}
& a = \{ b, d \} \\
& c = a \\
& e = a
\end{align*}
\]

Points-to Relations

\[
\begin{array}{c}
\text{a, c, e} \\
\text{b, d}
\end{array}
\]
Andersen vs. Steensgaard Pointer Analysis

- Both are flow-insensitive and context-insensitive
- Differ in points-to set construction
  - Andersen: many out edges, one variable per node
  - Steensgaard: one out edge, many variables per node
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 [2, 4]

Context-Sensitivity:

- Call site: by program statement of method invocation
  \[ S: \textit{this} \rightarrow \text{call\_method()} \]

- Object sensitivity: by receiving object of method invocation (state of the object is not merged)
  \[ S: \textit{this} \rightarrow \text{call\_method()} \]
Lars Ole Andersen.  

Laurie Hendren.  
Context-sensitive points-to analysis: Is it worth it.  

Michael Hind.  
Pointer analysis: Haven’t we solved this problem yet?  

Ana Milanova, Atanas Rountev, and Barbara G. Ryder.  
Parameterized object sensitivity for points-to analysis for java.  

Bjarne Steensgaard.  
Points-to analysis in almost linear time.  