Ultra-fast Aliasing Analysis using CLA
A Million Lines of C Code in a Second

N. Heintze    O. Tardieu

Neel Krishnaswami / 15-745 Optimizing Compilers Paper Presentation
The Problem

- Large (1+ MLoc) code base in C
- Programmer changes variable or struct type
- What else should be updated for “type-consistency”?
Partial Solution (1) – Typechecking

- Typechecking doesn’t work
- Casts make C’s type system unsound
- True dependencies get lost
void foo(int tag, void *data) {
    switch(tag) {
        case CHAR: bar((char*) data); break;
        case NUM: baz((short*) data); break;
    }
}

int main(...) {
    short* data = f();
    foo(NUM, (void*) data);
}
void foo(int tag, void *data) {
    switch(tag) {
    case CHAR: bar((char*) data); break;
    case NUM:  baz((short*) data); break;
    }
}

int main(...) {
    long* data = f();
    foo(NUM, (void*) data);
}
Variable \( x \) \textit{depends on} \( y \) if a change in \( y \)'s value could change \( x \)'s value.

\[
x := y + 5 \quad // \quad x \text{ depends on } y
\]
\[
u := x \times v \quad // \quad u \text{ depends on } x, v
\]

Can we compute a dependency graph of the program variables?
Problems with Data Dependency Graphs

The C code:

\[ *x = y + 5; \]

has the IR:

\[ H[x] := y + 5 \]

1. Heap H is (conceptually) a variable
2. Every pointer update modifies the heap
3. Analysis uselessly conservative
Flow-insensitive, context-insensitive points-to analysis:

1. Model the heap $H$ as a set of *abstract locations* (usually program expressions).
2. Model the program $P$ as a set of assignment statements.
3. Compute the transitive closure for each assignment

\[
\begin{align*}
  x &\to \& y \quad (\text{if } \star x = e \text{ in } P) \\
  y &\to e \\
  e_1 &\to e_2 \quad (\text{if } e_1 = e_2 \text{ in } P) \\
  x &\to \& y \\
  y &\to e \\
  e_2 &\to e_3 \\
  e_1 &\to e_3
\end{align*}
\]
Problems with Points-to Analysis

- Program has $O(n)$ abstract locations, and $O(n)$ variables.
- With full sets, reachability graph has $O(n^2)$ memory usage.
- If $n = 10^6$, then $O(n) \approx 10^{12}$!
Heintze and Tardieu’s Solution

Two parts:

1. Algorithmic Improvements
2. Architectural Improvements
Basic problem: transitive closure of reachability graph has $O(n^2)$ edges.
Heintze and Tardieu’s solution: Store the graph in pre-transitive form.
Algorithmic Improvements

The points-to analysis:

\[
\frac{x \rightarrow & y}{y \rightarrow e} \quad \text{(if } \star x = e \text{ in } P) \quad \frac{x \rightarrow & y}{e \rightarrow y} \quad \text{(if } e = \star x \text{ in } P) \\
\frac{e_1 \rightarrow e_2}{e_3} \quad \text{(if } e_1 = e_2 \text{ in } P) \quad \frac{e_1 \rightarrow e_2}{e_2 \rightarrow e_3} \quad \frac{e_1 \rightarrow e_2}{e_3}
\]

The pre-transitive points-to analysis:

\[
\frac{x \rightarrow & y}{y \rightarrow e} \quad \text{(if } \star x = e \text{ in } P) \quad \frac{x \rightarrow & y}{e \rightarrow y} \quad \text{(if } e = \star x \text{ in } P) \\
\frac{e_1 \rightarrow e_2}{e_3} \quad \text{(if } e_1 = e_2 \text{ in } P)
\]

Now, to find reachable locations, we must traverse the graph manually – familiar time/space tradeoff. (Recall epsilon transition elimination from automata theory.)
Relational presentation hides traversals. Two optimizations of traversal:

1. Merge nodes in cycles, whenever graph reachability detects them
2. Memoize reachability calls (with the expected algorithmic changes)
Heintze and Tardieu claim that standard tools:

- Parse entire source base
- Build in-memory data structures
- Analyze these data structures

For large systems, this is slow and resource-hungry.
Break the analyzer into three parts:

- “Compiler”, which takes source code and produces algorithm-neutral summaries as “object files”.
- “Linker”, which merges the needed object files for an analysis
- “Analyzer”, which does the analysis
<table>
<thead>
<tr>
<th>program</th>
<th>LOC</th>
<th>“Object” file</th>
<th>variables</th>
<th>pointers</th>
<th>run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>nethack</td>
<td>-</td>
<td>0.7MB</td>
<td>3856</td>
<td>1018</td>
<td>0.03s</td>
</tr>
<tr>
<td>burlap</td>
<td>-</td>
<td>1.4MB</td>
<td>6859</td>
<td>3332</td>
<td>0.08s</td>
</tr>
<tr>
<td>vortex</td>
<td>-</td>
<td>2.6MB</td>
<td>11395</td>
<td>4359</td>
<td>0.15s</td>
</tr>
<tr>
<td>emacs</td>
<td>-</td>
<td>2.6MB</td>
<td>12587</td>
<td>8246</td>
<td>0.54s</td>
</tr>
<tr>
<td>povray</td>
<td>-</td>
<td>3.1MB</td>
<td>12570</td>
<td>6126</td>
<td>0.11s</td>
</tr>
<tr>
<td>gcc</td>
<td>-</td>
<td>4.4MB</td>
<td>18749</td>
<td>11289</td>
<td>0.20s</td>
</tr>
<tr>
<td>gimp</td>
<td>440K</td>
<td>27.2MB</td>
<td>131552</td>
<td>45091</td>
<td>1.05s</td>
</tr>
<tr>
<td>lucent</td>
<td>1.3M</td>
<td>20.1MB</td>
<td>96509</td>
<td>22360</td>
<td>0.46s</td>
</tr>
</tbody>
</table>