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

Partial Solution (1) – Typechecking

```
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);
}
```

Partial Solution (1) – Typechecking

```
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);
}
```

Partial Solution (2) – Data Dependency Graphs

Variable x *depends on* y if a change in y's value could change x's value.

```
x := y + 5  // x depends on y
u := x * v  // u 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

Partial Solution (3) – Points-to Analysis

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

$$\frac{x \rightarrow \&y}{y \rightarrow e} \text{ (if } \star x = e \text{ in P)} \qquad \frac{x \rightarrow \&y}{e \rightarrow y} \text{ (if } e = \star x \text{ in P)}$$

$$\frac{}{e_1 \rightarrow e_2} \text{ (if } e_1 = e_2 \text{ in P)} \qquad \frac{e_1 \rightarrow e_2 \quad e_2 \rightarrow e_3}{e_1 \rightarrow e_3}$$

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

Algorithmic 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 Improvments

The points-to analysis:

$$\frac{x \to \&y}{y \to e} \text{ (if } \star x = e \text{ in P)} \qquad \frac{x \to \&y}{e \to y} \text{ (if } e = \star x \text{ in P)}$$

$$\frac{}{e_1 \to e_2} \text{ (if } e_1 = e_2 \text{ in P)} \qquad \frac{e_1 \to e_2 \quad e_2 \to e_3}{e_1 \to e_3}$$

The pre-transitive points-to analysis:

$$\frac{x \to \&y}{y \to e} \text{ (if } \star x = e \text{ in P)} \qquad \frac{x \to \&y}{e \to y} \text{ (if } e = \star x \text{ in P)}$$

$$\frac{}{e_1 \to e_2} \text{ (if } e_1 = e_2 \text{ in P)}$$

Now, to find reachable locations, we must traverse the graph manually – familiar time/space tradeoff. (Recall epsilon transition elimination from automata theory.)

Traversal Optimizations

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)

Architectural Improvements

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.

Compile-Link-Analyze Architecture

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

Results

program	LOC	“Object” file	variables	pointers	run time
nethack	-	0.7MB	3856	1018	0.03s
burlap	-	1.4MB	6859	3332	0.08s
vortex	-	2.6MB	11395	4359	0.15s
emacs	-	2.6MB	12587	8246	0.54s
povray	-	3.1MB	12570	6126	0.11s
gcc	-	4.4MB	18749	11289	0.20s
gimp	440K	27.2MB	131552	45091	1.05s
lucent	1.3M	20.1MB	96509	22360	0.46s