Interprocedural Analysis

Last time
- Interprocedural analysis

Today
- Interprocedural alias analysis
- Interprocedural optimization

Jump Functions and Return Jump Functions for ICP
\[
J^{\text{formal}}_{\text{callsite}} = f(\text{actualls, globals, constants})
\]
\[
R^{\text{global OR refparam}}_{\text{function}} = f(\text{formals, globals, constants})
\]

\[
\begin{align*}
\text{int } & a, b, c, d; \\
\text{void } & \text{foo}(e) \\
& \text{a} = b + c; \\
& d = e + 2;
\end{align*}
\]
\[
\text{foo}(3);
\]
\[
J^e_{\text{foo}(3)} = 3 \\
R^d_{\text{foo}} = e + 2 \\
R^a_{\text{foo}} = b + c
\]

Partial Transfer Functions for Interprocedural Alias Analysis
- funcOutput = PTF(funcInput)
- use memoization
- PTF lazily computed for each input pattern that occurs

Improving the Efficiency of the Iterative Algorithm
Partial Transfer Function [Wilson et. al. 95]

Example [http://www.cs.princeton.edu/~jqwu/Memory/survey.html]

```c
main() {
    int *a, *b, c, d;
    a = &c;
    b = &d;
    S0 foo(&a, &b);
    for (i = 0; i<2; i++) {
        S1 bar(&a, &a);
        S2 bar(&b, &b);
        S3 bar(&a, &b);
        S4 bar(&b, &a);
    }
}

void bar(int **i, int **j) { foo(i, j); }
void foo(int **x, int **y) {
    int *temp = *x;
    *x = *y;
    *y = temp;
}
```

Characterizing Interprocedural Analysis

**Definiteness**
- May (possible) versus must (definite)

**Flow sensitivity**
- Sensitive (consider control flow)
  - Requires iterative data-flow analysis or similar technique
  - More accurate than flow-insensitive
- Insensitive (ignore control flow)
  - Can compute in linear time
  - May information only

**Context sensitivity**
- Sensitive (polyvariant analysis)
  - Re-analyze callee for each caller
  - Variations based on how much of the call path is maintained
- Insensitive (monovariant analysis)
  - Perform one analysis independent of callers
**Emami 1994**

**Overview**
- Compute L and R locations to implement flow-sensitive data-flow analysis
- Uses invocation graph for full context-sensitivity
- Can be exponential in program size
- Handles function pointers

**Characterization of Emami**
- Whole program
- Flow-sensitive
- Context-sensitive
- May and must analysis
- Alias representation: points-to
- Heap modeling: one heap variable
- Aggregate modeling: fields and first array element

**Choi 1993**

**Overview**
- Iterates over call graph with callsite labeled edges
- Iterates over Sparse Evaluation Graph for each procedure

**Characterization of Choi**
- Whole program
- Flow-sensitive
- Context-sensitive (one-level)
- May analysis
- Alias representation: compact pairs (similar to points-to)
- Heap modeling: k names for each malloc stmt
- Aggregate modeling: fields?
**Burke 1995**

**Overview**
- Iterates over call graph with callsite labeled edges
- Iterates over Sparse Evaluation Graph for each procedure
- Use kill information before propagating on call graph
- Handles function pointers

**Characterization of Burke**
- Whole program
- Flow-insensitive (kill info is propagated along call edges)
- Context-sensitive
- May analysis
- Alias representation: compact pairs (similar to points-to)
- Heap modeling: k names for each malloc stmt?
- Aggregate modeling: fields?

---

**Alias/Pointer Analysis Summary**
Interprocedural Analysis vs. Interprocedural Optimization

Interprocedural analysis
- Gather information across multiple procedures (typically across the entire program)
- Can use this information to improve intraprocedural analyses and optimization (e.g., CSE)

Interprocedural optimizations
- Optimizations that involve multiple procedures e.g., Inlining, procedure cloning, interprocedural register allocation
- Optimizations that use interprocedural analysis

Alternative to Interprocedural Analysis: Inlining

Idea
- Replace call with procedure body

Pros
- Reduces call overhead
- Exposes calling context to procedure body
- Exposes side effects of procedure to caller
- Simple!

Cons
- Code bloat (decrease efficacy of caches, branch predictor, etc)
- Can’t always statically determine callee (e.g., in OO languages)
- Library source is usually unavailable
- Can’t always inline (recursion)
**Inlining Policies**

**The hard question**
- How do we decide which calls to inline?

**Many possible heuristics**
- Only inline small functions
- Let the programmer decide using an `inline` directive
- Use a code expansion budget [Ayers, et al ’97]
- Use profiling or instrumentation to identify hot paths—inline along the hot paths [Chang, et al ’92]
- JIT compilers do this
- Use inlining trials for object oriented languages [Dean & Chambers ’94]
- Keep a database of functions, their parameter types, and the benefit of inlining
- Keeps track of indirect benefit of inlining
- Effective in an incrementally compiled language

**Inlining versus Interprocedural Analysis**

**How effective is inlining?**
- Richardson & Ganapathi [1989] compared it to interprocedural analysis
- Context
  - Pascal on RISC processors
  - Used interprocedural USE, MOD, ALIASES information

**Results**
- Interprocedural analysis resulted in small benefit (<2%)
- Simple link-time inlining provided more benefit (10%)
**Alternative to Interprocedural Analysis: Cloning**

**Procedure Cloning/Specialization**
- Create a customized version of procedure for particular call sites
- *Compromise* between inlining and interprocedural optimization

**Pros**
- Less code bloat than inlining
- Recursion is not an issue (as compared to inlining)
- Better caller/callee optimization potential (versus interprocedural analysis)

**Cons**
- Still some code bloat (versus interprocedural analysis)
- May have to do interprocedural analysis anyway
  - *e.g.* Interprocedural constant propagation can guide cloning

**Procedure Cloning**

**Abstract implementation**
- Given a set of call sites to procedure p
  - *e.g.*, \(\{c_1, c_2, c_3, c_4, c_5, c_6\}\)
- Partition them into equivalence classes of “similar” call sites
  - *e.g.*, \(\{\{c_1, c_4\}, \{c_2, c_3\}, \{c_6\}\}\)
- Meaning of “similar” depends on the intended benefit
  - *e.g.*, For constant propagation, partition according to constant valued actual parameters

**Important question**
- How do we partition the call sites?
**Evaluation**

**Why don’t many compilers use interprocedural analysis?**
- Benefits on optimization have not been well explored
- Common view: not beneficial for most scalar optimizations
- It’s expensive and complex
- Separate compilation + interprocedural analysis requires **recompilation analysis** [Burke and Torczon’93]
- Can’t analyze library code

**When is it useful?**
- Pointer analysis
- Parallelization
- Constant propagation
- Object oriented class analysis
- Error checking

---

**Trends**

**Questions**
- Is interprocedural analysis really useful?
- Is it worth doing anything beyond inlining?

**Trends**
- Cost of procedures is growing
  - More of them and they’re smaller (OO languages)
  - Modern machines demand precise information (memory op aliasing)
- Cost of inlining is growing
  - Code bloat degrades efficacy of many modern structures
  - Procedures are being used more extensively
- Programs are becoming larger
- Cost of interprocedural analysis is shrinking
  - Faster machines
  - Better methods
**Trends (cont)**

**Trends**
- Call graph construction is complicated by modern languages
  - Dynamic binding of methods
  - Dynamically loaded code

**Summary**
- Interprocedural analysis (and cloning) are becoming more important

---

**Historical Note: Interprocedural Alias Analysis**

**Until recently**
- Interprocedural alias analysis was mostly concerned with detecting aliasing formal parameters and globals (in call-by-var context)
- Perhaps the general (*i.e.*, C) problem was viewed as hopeless

**Recently (c. 2003-2004)**
- Pointer analysis using Binary Decision Diagrams (BDDs)
- Approach to handle context-sensitivity in an efficient manner
Concepts

Partial transfer functions for context-sensitive alias analysis

Different kinds of context-sensitivity

Comparison of alias analysis algorithms in terms of context and flow sensitivity

Alternatives to interprocedural analysis
  – Inlining
  – Procedure cloning

Next Time

Reading
  – Ch 16 in Muchnick, focus on 16.3.11

Next lecture
  – Register allocation