CSE 5317

Lecture 23: Interprocedural analysis and optimization
13 April 2010

Nate Nystrom
UTA
Today

• Interprocedural analysis
  • Example: alias analysis

• Interprocedural optimization
  • Examples: inlining, cloning
Introduction

- Interprocedural analysis
  - Gathering information about the whole program instead of a single procedure
- Interprocedural optimization
  - Program transformation that involves more than one procedure in the program
  - (uses interprocedural analysis)
Aliasing

- Two expressions are **aliases** if they denote the same memory location
  - `p = new Object`
  - `q = p`  \(\Rightarrow \) \(*p\) and \(*q\) alias

- Aliases arise with:
  - pointers
  - call by reference
  - array indexing
  - C union
Aliasing examples

- Pointers
  - `int *p, i;`
  - `p = &i; // *p and i alias`

- Call by reference
  - `void proc(int& a, int &b) { ... }`
  - `proc(x, x); // a and b alias in proc`
  - `proc(x, glob); // b and glob alias in proc`

- `int i, j, a[N];`
  - `i = j; // a[i] and a[j] alias`
Alias analysis

- Goal: statically identify aliases
  - can memory reference m and n access the same state at program point p?
  - what program state can reference m access?

- Why important?
  - \( *p = a + b \);
  - \( y = a + b; \) // if \( *p \) aliases a or b, 2nd a+b
    // is NOT redundant

- If not done, other analyses must be very conservative
Trivial alias analysis

• Easy
  • Assume nothing **must** alias
  • Assume everything **may** alias

• Better (C):
  • Assume nothing **must** alias
  • Assume all pointer derefs **may** alias
  • Assume variables whose address is taken (and globals) **may** alias all pointer derefs

• \( p = \& a; \)  
  • \( *q = 5; \)  
  • \( a = 3; \) \( b = 4; \)  
  • \( *q \) and \( a \) may alias so \( a \) may be 3 or 5
  • \( *q \) and \( b \) do not alias, so \( b \) is 4
Type-based alias analysis

• If strongly typed, use type information too

• Two variables with incompatible types cannot alias
  • String vs. List – cannot alias
  • String vs. Object – may alias

• Two fields with different names cannot alias
Points-to analysis

• Represent alias analysis results as **points-to relation**
  • \(p \rightarrow \{x,y\}\)
  • \(p\) contains the address of \(x\) and \(y\)
  • “\(p\) points to \(x\) or \(y\)”
• How does control flow affect the analysis?

    if (c)
        p = &x;
    else
        p = &y;
Flow sensitivity

- An analysis may (or may not) consider control flow

- **Flow sensitive**
  - Control flow information is used

- **Flow insensitive**
  - Control flow information is not used
  - Do not consider order of execution
Calls

• What happens to points-to sets at function calls?

  • p1 = &x;
  • p2 = &p1;  // p2 -> p1 -> x
  • foo();  // ???

• Be conservative
  • assume any reachable pointer may be changed
• Lose lots of information

• Can do better!
Interprocedural control flow graph

- Compose the CFG for all procedures into a single graph
  - connect return nodes of callee back to caller

  **supergraph**

- Simple, relatively effective
- Flow sensitive

- Inaccurate
  - if proc (statically) called n times, n edges into proc, n edge from return out
  - “unrealizable paths”

- Slow
- No separate compilation

Tuesday, May 4, 2010
Context sensitivity

- Can make analysis more precise by considering calling context

- **Context sensitive**
  - Calling context is used

- **Context insensitive**
  - Calling context is not used
Classification

- Flow/context sensitive
  - more precise
  - slower
- Flow/context insensitive
  - less precise
  - faster
Brute force

- Reanalyze callee for all distinct calling paths

- Precise
- Exponentially expensive
- Recursion tricky
Middle ground

• Rather than building interprocedural CFG or reanalyzing functions:

  • Use **call graph** and compute **summaries**
Call graph

1  procedure f()
2    begin
3      call g()
4      call g()
5      call h()
6    end
7  procedure g()
8    begin
9      call h()
10     call i()
11    end
12  procedure h()
13    begin
14    end
15  procedure i()
16    procedure j()
17    begin
18    end
19  begin
20     call g()
21     call j()
22  end

- Nodes = procedures
- Edge (p1,p2) if p1 calls p2
- Edges labeled with call-sites (statement that does the call)
Summaries

- Compute summary information for each procedure
  - Summarize effect/result of called procedure for callers
  - Summarize effect/input of callers for called procedure

- Store summaries in database for later use

- Concise
- Fast to compute, use
- Separate compilation practical (save summaries to disk)

- Imprecise if only summarize per procedure
What to summarize?

• Information that flows into a procedure
  • *propagation problems*
  • what formals are constant?
  • what formals are aliased to globals?

• Information that flows out of a procedure
  • *side effect problems*
  • what globals are defined/used?
  • what locals are defined/used?
  • what actual parameters are defined?
Examples

• Propagation summaries

  • May alias
    • set of formals that may be aliased to globals or each other

  • Must alias
    • set of formals that are definitely aliased to globals or each other

• Side effect summaries:

  • MOD: set of variables possibly modified by the call
  • REF: set of variables possibly read
Context sensitivity of summaries

- k levels of the call path
  - compute separate results for each different context

- k = 0: smears all callers to a procedure together
- k = 1: consider only the call-site

- Complexity grows exponentially with k
Using summaries

• When analyzing a call, get summary of proc(s) called
• Substitute actual arguments for formal parameters to compute effect of call
Flow insensitive and context insensitive

- Ignore the control-flow graph
- Assume statements can execute in any order
- Produce a single solution valid for the whole program

- Two common alias analyses
  - Andersen
  - Steensgaard

- Most others are hybrids of these two
Andersen 94

- Uses subset constraints
- O(n^3) in program size

```c
int **a, *b, c, *d, e;
a = &b;  // 1
b = &c;  // 2
d = &e;  // 3
a = &d;  // 4
```

Diagram:

```
due to stmt 4
```

Source: Barbara Ryder's Reference Analysis slides
Steensgaard 96

- Uses unification constraints
- Almost O(n) in program size
- Uses fast union-find algorithm
- Imprecise merging of sets

```c
int **a, *b, c, *d, e;
a = &b; // 1
b = &c; // 2
d = &e; // 3
a = &d; // 4
```

![Diagram of pointer relationships]
Issues

• Large programs
• Partial programs
• Modeling the heap (shape analysis)
Effect on optimization

• Interprocedural analysis more precise than making conservative assumptions about calls

• Does it actually help?
  • Not much!
  • < 2% performance gain

• So why do it?
For optimization

• 2% is still > 0

• very space efficient vs. other interprocedural optimizations
Bug finding

- Analyze code to look for bugs
  - memory leaks
  - data races
  - bad use of locking
  - security bugs

- More precise analysis => fewer false positives, less work for programmer
Interprocedural optimization

- Can optimize across procedures with very little analysis
  - just need a call graph
- Inlining
- Procedure cloning
Inlining

- Replace call with procedure body
- Reduces call overhead
- Exposes calling context to procedure body
- Exposes side effects of procedure to caller
- Simple (don’t have to do interprocedure analysis!)

- Code bloat
- Can’t always statically determine callee (OO)
- Can’t always inline (recursion)
Policies

• How to decide what calls to inline?
• Possible heuristics:
  • small functions
  • programmer-supplied inline directive
  • code-expansion budget
  • profiling to identify hot paths (JITs do this)
  • inlining trials
    • keep database of functions, types, and benefit of inlining
    • effective with incremental compilation
Benefit?

• Interprocedural analysis
  • < 2% benefit

• Link-time inlining
  • 10% benefit
Virtual calls and closures

• Virtual calls
  • class C { void m() { ... } }
  • class D extends C { void m() { ... } }
  • C c = random() ? new C() : new D();
  • c.m(); // what methods does this call?

• Closures
  • val f = if (random()) (x:Int) => x+1 else (x:Int) => x
  • f(1) // what function does this call?
Control flow analysis

• Identify functions a given call-site might invoke
  • i.e., build the call graph

• Can be context sensitive also

• 0-cfa – context insensitive
• k-cfg – considers k levels of calls
Inlining a virtual call

• class C { void m() { ... } }
• class D extends C { void m() { ... } }
• C c = random() ? new C() : new D();
• c.m(); // what methods does this call?

• =>

• if (c.getClass() == C.class) /* body of C.m */
• else if (c.getClass() == D.class) /* body of D.m */
• else c.m();
Cloning

• Create a customized version of procedure for each call site

• Less code bloat than inlining
• Recursion not an issue
• Better caller/callee optimization potential (vs. interproc analysis)

• Still some code bloat
• May have to do interprocedural analysis anyway to guide cloning
Trends

- Cost of procedures is growing
  - More of them and smaller (OO)
  - More memory op aliasing
- Cost of inlining is growing
  - Code bloat bad
- Programs becoming larger
- Cost of interprocedural analyses shrinking
  - faster machines, better methods