Interprocedural Analysis Motivation

```
a = 1;
b = 2;

c = a + b;
```

```
a = 1;
b = 2;

c = 3;
```

Interprocedural Analysis Motivation

```
a = 1;
b = 2;
foo();
c = a + b;
```

```
a = 1;
b = 2;
foo()
c = ???;
```

Does `foo()` modify a or b?

Interprocedural Analysis Motivation

```
a = 1;
b = 2;
foo();
c = a + b;
```

```
a = 1;
b = 2;
foo()
c = ???;
```

Does `foo()` modify a or b?

```
int add(int a, int b) {
  return a + b;
}
c = add( 1, 2 );
```

Are a and b constant?

Interprocedural Analysis Direction

- Bottom-Up: Information about called methods

```
a = 1;
b = 2;
foo();
c = a + b;
```

- Top-Down: Information about calling context

```
int add(int a, int b) {
  return a + b;
}
```

Many problems require both (e.g. alias analysis)

Interprocedural Analysis Options

- Option 1: Worst-case conservative assumptions

Interprocedural Analysis Options

- Option 1: Worst-case conservative assumptions
- Option 2: Inline, then analyze

```
int add(int a, int b) {
  return a + b;
}
c = add( 1, 2 );
```

```
c = 1 + 2;
```

Interprocedural Analysis Options

- Option 3: Method summary
  Example: For each method m, compute MOD(m), the set of all globals possibly modified in the method.

Interprocedural Analysis Options

- Option 3: Method summary
  Example: For each method m, compute MOD(m), the set of all globals possibly modified in the method.
- Option 4: Control flow supergraph
  Link control flow graphs of all methods together, then do some variation of dataflow analysis.

Context Sensitivity

Context-Insensitive Analysis

Analysis result for a procedure is sound for all invocations of the procedure.

Context-Sensitive Analysis

Distinguish analysis results for different invocations of the procedure.

“Two Approaches” to Context Sensitivity

M. Sharir, A. Pnueli. “Two approaches to interprocedural data flow analysis.” 1981

Call string approach

Given dataflow lattice L, instead use lattice $C* \to L$, where $C*$ is set of strings of call sites.

Functional approach

Given dataflow lattice L, compute for each procedure an element of $L \to L$ summarizing its effect on each lattice element.

IFDS: Efficient Representation of $L \to L$

Interprocdural Finite Distributive Subset

Theorem

Suppose $L = \mathcal{P}(D)$, where D is a finite set, and function $f : L \to L$ is distributive. Then f is uniquely determined by the effect of f on the empty set and on singleton sets.

IFDS: Efficient Representation of $L \to L$

Interprocdural Finite Distributive Subset

Theorem

Suppose $L = \mathcal{P}(D)$, where D is a finite set, and function $f : L \to L$ is distributive. Then f is uniquely determined by the effect of f on the empty set and on singleton sets.

Proof

$f(X) = f(\{\}) \sqcup \bigsqcup_{x \in X} f(\{x\})$

IFDS: Efficient Representation of $L \to L$

Interprocdural Finite Distributive Subset

Theorem

Suppose $L = \mathcal{P}(D)$, where D is a finite set, and function $f : L \to L$ is distributive. Then f is uniquely determined by the effect of f on the empty set and on singleton sets.

Proof

$f(X) = f(\{\}) \sqcup \bigsqcup_{x \in X} f(\{x\})$

We can represent any such f by a graph of edges from $L \cup \{0\}$ to L.

IFDS: Efficient Representation of $L \to L$

Interprocdural Finite Distributive Subset

Theorem

Suppose $L = \mathcal{P}(D)$, where D is a finite set, and function $f : L \to L$ is distributive. Then f is uniquely determined by the effect of f on the empty set and on singleton sets.

Proof

$f(X) = f(\{\}) \sqcup \bigsqcup_{x \in X} f(\{x\})$

We can represent any such f by a graph of edges from $L \cup \{0\}$ to L.

Function composition: combine copies of graphs.

Join on functions: combine edges from graphs.

Call Graphs

Call Graphs

Call Graph

Edges $C \to M$, where
C is a call site,
M is a target method.

Call Graphs

Call Graph

Edges $C \to M$, where
C is a call site,
M is a target method.

```
if() {
  f = &foo;
} else {
  f = &bar;
}
c = f(1, 2);
```

Which method(s) are invoked on the last line?

Call Graph Construction

Class Hierarchy Analysis

```
A m;
m.foo();
```

Assume `m` could be any subtype of `A`.

Call Graph Construction

Rapid Type Analysis

Algorithm RTA():

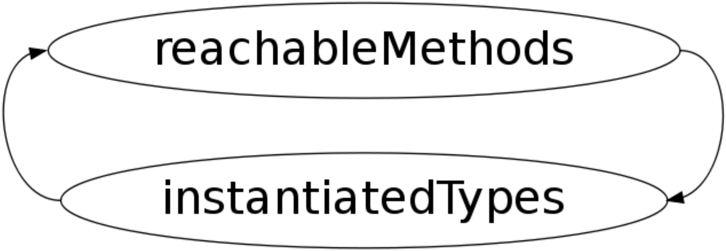
```
reachableMethods := {main}
instantiatedTypes := {}
repeat
    for all methods m in reachableMethods do
        for all allocation sites x = new C() in m do
            instantiatedTypes := instantiatedTypes ∪ {C}
        for all call sites n.foo() in m do
            resolve the call assuming n can have any type in
            instantiatedTypes
            add resolved method to reachableMethods
until no changes
```

Rapid Type Analysis Example

```
List l = new ArrayList();
l.add("string");
```

- CHA would assume `l` can be any subtype of `List` (such as `LinkedList`, `Vector`, `Stack`, ...).
- RTA uses the fact that only an `ArrayList` is ever instantiated.

RTA Cyclic Dependence



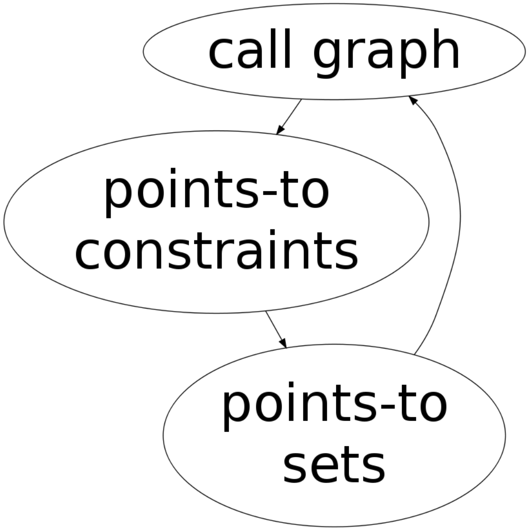
Call Graph Construction

Points-to-based Call Graph Construction

Algorithm PTA-CG():

```
add main method to call graph
repeat
    generate points-to constraints for call graph
    solve points-to constraints
    resolve call sites using points-to information, adding
    edges to call graph
until no changes
```

PTA Cyclic Dependence



Implementing Virtual Function Calls

Simplest case: single inheritance

```
o->f();
```

```
load *(o+$vtbl), t
load *(t+$fOffset), m
call m
```

Static devirtualization

- reduces overhead of call sequence
- enables other optimizations
- is difficult for realistic programs

Inline Cache

```
call m
```

```
m() {
 if(this.class!=mClass)
  goto slowLookup
 ...
}
```

Inlining

Devirtualization removes most, but not all function call overhead.

Inlining

- removes call overhead completely
- enables additional optimizations
- may increase code size (infinitely in case of recursion)
- may reduce instruction cache effectiveness
- requires devirtualized call (or can be speculative)

Outlining

Problem

Many functions contain rarely or never-executed code (e.g. exception handlers)

- pollutes instruction cache
- confuses dataflow analysis
- inhibits inlining of hot code

Solutions

- move cold code to separate method, and call it if necessary
- remove cold code entirely, and use recompilation and on-stack replacement if necessary

Trace-based Optimization

- Profile to find hot traces (paths through CFG)
- Straighten them and aggressively optimize
    - Convert branches to bail-out code, move as early as possible
    - Optimization can be cheap: trace $\simeq$ basic block