Interprocedural Analysis Motivation

a = 1;
b = 2;
c = a + b;

Does \texttt{foo()} modify \texttt{a} or \texttt{b}? 

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

c = \text{add}( 1, 2 );

Are \texttt{a} and \texttt{b} constant?
Does `foo()` modify `a` or `b`?
Does `foo()` modify `a` or `b`?

```c
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
Option 1: Worst-case conservative assumptions

Option 2: Inline, then analyze

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

```c
  c = 1 + 2;
```
Option 3: Method summary
Example: For each method $m$, compute $\text{MOD}(m)$, the set of all globals possibly modified in the method.
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.

Call string approach
Given dataflow lattice $L$, instead use lattice $C^* \rightarrow L$, where $C^*$ is set of strings of call sites.

Functional approach
Given dataflow lattice $L$, compute for each procedure an element of $L \rightarrow L$ summarizing its effect on each lattice element.
**Theorem**

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

**Theorem**

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

**Proof**

$$f(X) = f(\{\}) \sqcup \bigsqcup_{x \in X} f(\{x\})$$
Interprocedural Finite Distributive Subset

**Theorem**

Suppose $L = \mathcal{P}(D)$, where $D$ is a finite set, and function $f : L \rightarrow L$ 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 $L$. 
Interprocedural Finite Distributive Subset

Theorem

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

Proof

\[
f(X) = f(\emptyset) \sqcup \bigsqcup_{x \in X} f(\{x\})
\]

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 Graph

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

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

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

Call Graph

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

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

Which method(s) are invoked on the last line?
Assume $m$ could be any subtype of $A$. 

```java
A m;
m.foo();
```
Algorithm RTA():
1: reachableMethods := \{main\}
2: instantiatedTypes := \{\}
3: repeat
4: for all methods m in reachableMethods do
5: for all allocation sites x = new C() in m do
6: instantiatedTypes := instantiatedTypes \cup \{C\}
7: for all call sites n.foo() in m do
8: resolve the call assuming n can have any type in instantiatedTypes
9: add resolved method to reachableMethods
10: until no changes
Rapid Type Analysis Example

```java
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

reachableMethods

instantiatedTypes
Points-to-based Call Graph Construction

**Algorithm** PTA-CG():

1. add main method to call graph
2. **repeat**
3. generate points-to constraints for call graph
4. solve points-to constraints
5. resolve call sites using points-to information, adding edges to call graph
6. **until** no changes
PTA Cyclic Dependence

call graph

points-to constraints

points-to sets
Simplest case: single inheritance

```c
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
call m

m() {
    if(this.class!=mClass)
        goto slowLookup
    ...
}
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)
<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Many functions contain rarely or never-executed code (e.g. exception handlers)</td>
</tr>
<tr>
<td>- pollutes instruction cache</td>
</tr>
<tr>
<td>- confuses dataflow analysis</td>
</tr>
<tr>
<td>- inhibits inlining of hot code</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>move cold code to separate method, and call it if necessary</td>
</tr>
<tr>
<td>remove cold code entirely, and use recompilation and on-stack replacement if necessary</td>
</tr>
</tbody>
</table>
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