Call Graph Construction in Object-Oriented Languages

David Grove, Greg DeFouw,
Jeffrey Dean, and Craig Chambers

Department of Computer Science and Engineering
University of Washington

Motivation

Goal: **flexible**, **expressive** object-oriented languages that are **efficient**

Approach: Implementation should optimize message sends

- Statically bind and inline where possible/profitable
- For sends/calls that remain: **interprocedural analysis**
  - **Caller**: better assumptions about impact of callee(s) on caller state
  - **Callee**: better assumptions about properties of formal parameters

First step: construct **program call graph**

- Representation of program's calling structure
  - Nodes are procedures/methods
  - Edge from X to Y implies X calls Y

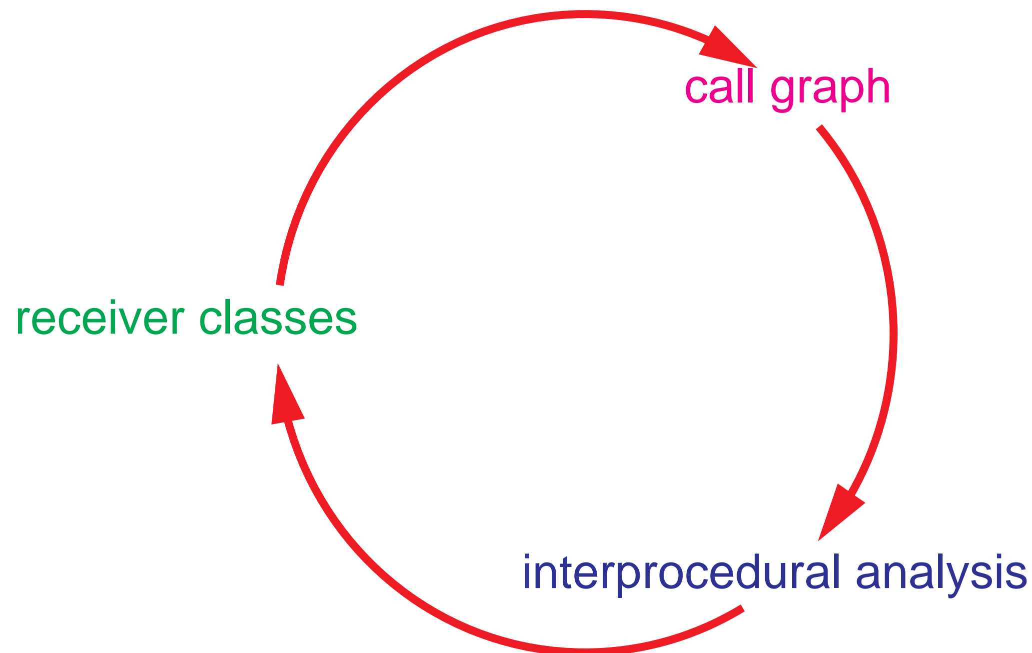
Second step: perform interprocedural analysis

Constructing the Call Graph

How to construct the program call graph in OO (or functional) languages?

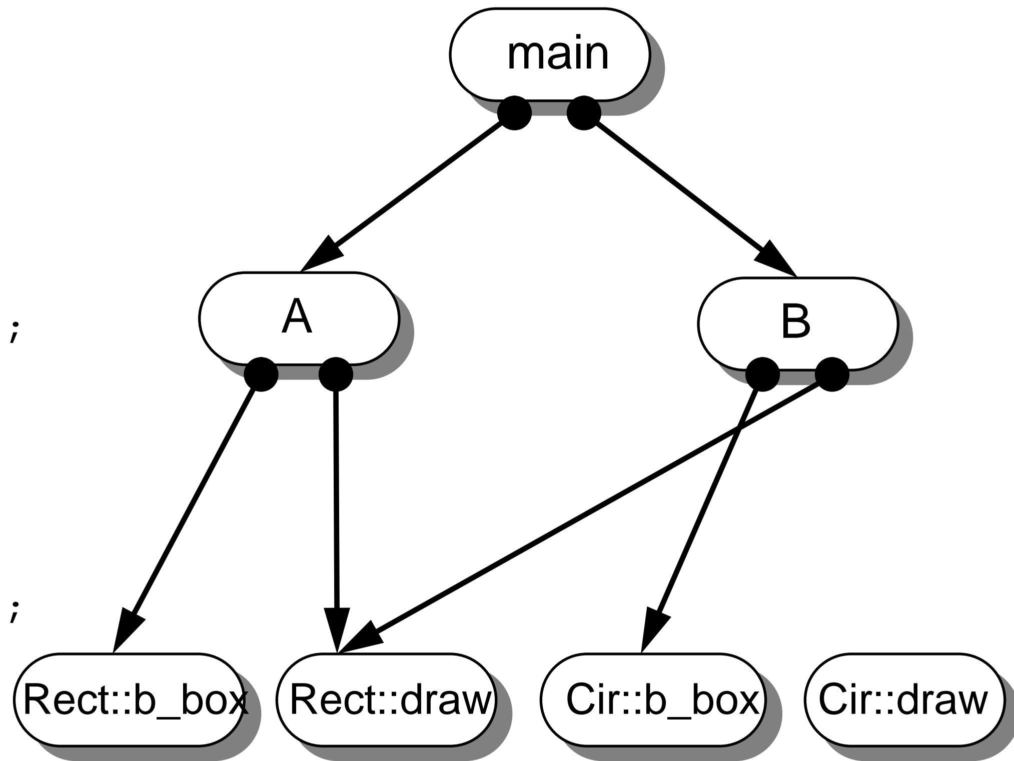
- **Dynamic dispatches**: callee(s) depend on class of receiver(s)
- **First-class functions**: callee(s) depend on value(s) of applied function

Analyzing possible classes & functions is an interprocedural analysis problem....

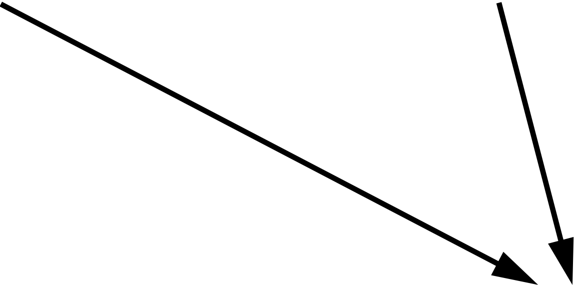


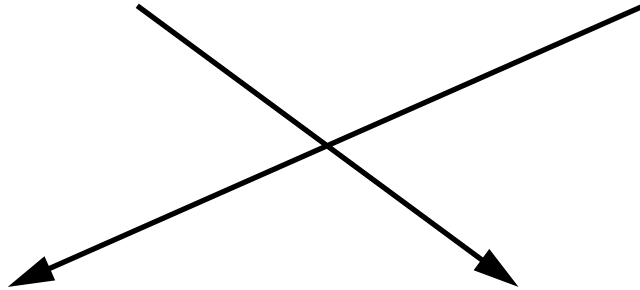
Call Graph Basics

```
procedure main() {
 A();
 B();
}

procedure A() {
 r := new Rectangle(...);
 r->bounding_box()->draw();
}

procedure B() {
 c := new Circle(...);
 c->bounding_box()->draw();
}
```



Want minimal safe call graph (fewest nodes/edges)





Contributions

A model of call graph construction algorithms,
including a formal lattice-theoretic model of call graph domain

- Unifies more algorithms better than previous work
- Suggests new algorithms
- Basis for a flexible implementation framework

Empirical assessment of cost and benefit of algorithms on **sizeable programs**

Call Graph Construction Algorithm

Add needed nodes & edges

unsound?

Initial Call Graph Construction

a call graph

sound?

solution call graph

additional precision desired?

Remove spurious nodes & edges

A ***sound*** call graph conservatively approximates the program’s runtime behavior

- Every procedure called during some program execution is included
- Every call arc traversed during some program execution is included

Initial Call Graphs

Option 1: construct a conservative call graph (quickly)

+ No further work required
– May be very imprecise (especially with first-class functions)
- Some (near-)linear-time algorithms:
  - **flow-insensitive**:
    Bacon & Sweeney's Rapid Type Analysis (RTA) algorithm
    Steensgaard's near-linear-time points-to analysis
  - **limited flow-sensitive**:
    DeFouw, Grove & Chambers's k-limited family of algorithms [POPL '98]

Option 2: construct an optimistic call graph (e.g., the empty call graph)

– Nodes/edges must be added
+ Potential for more precise final call graph

Adding Missing Nodes and Edges

Perform class analysis within each procedure

- Compute mapping of variables to sets of classes
- Start with sets of classes for formal parameters
- Use result set of classes of callees

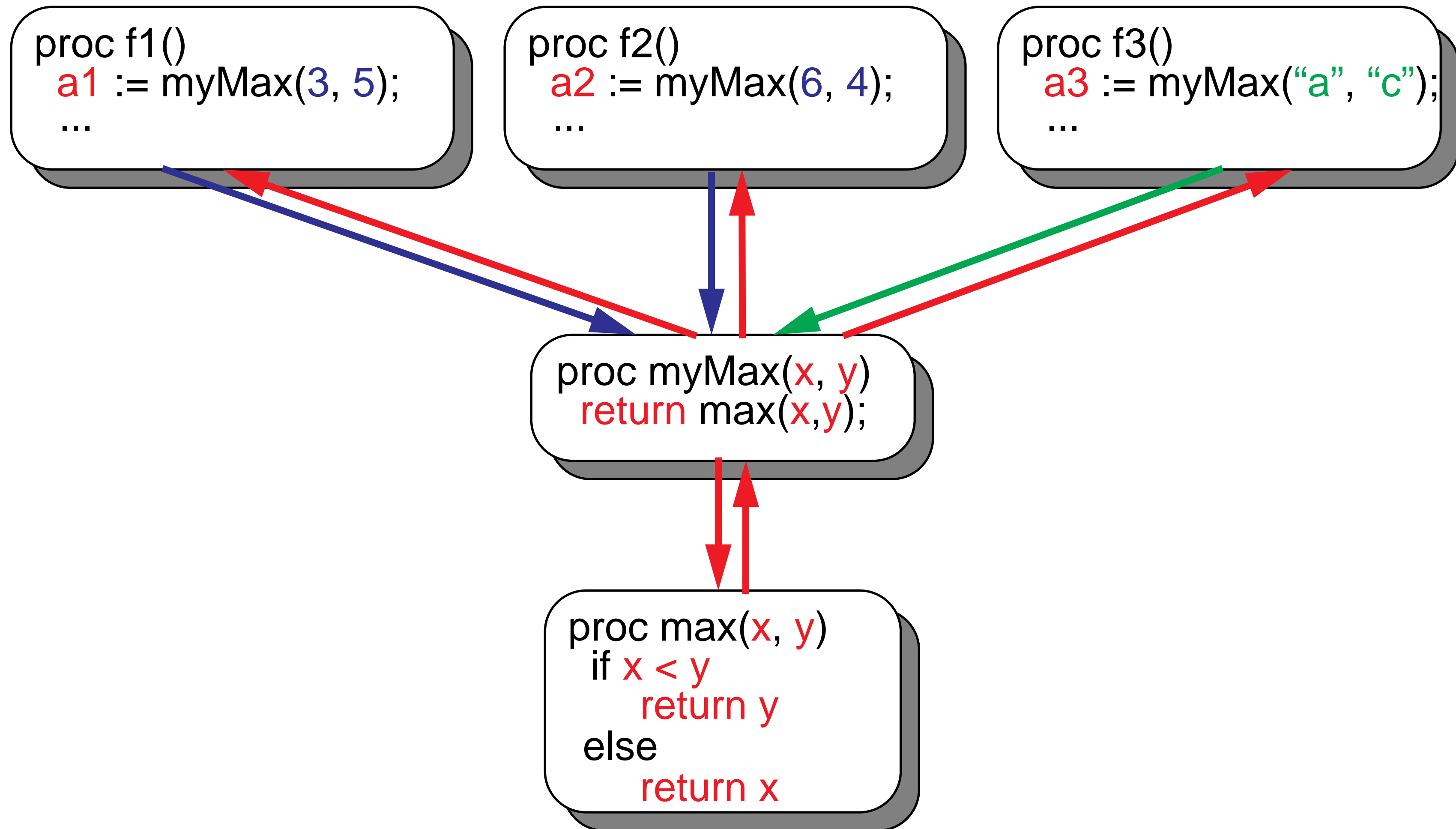
```
procedure B() {
  c := new Circle(...);            c→{Circle}
  t1 := c->bounding_box();         c→{Circle}, t1→{Rectangle}
  t1->draw();
}
```

At each message send:

- determine callee(s)
- add missing node/edges
- propagate actuals to formals
- propagate result

Contour Selection Function

Key decision: How to analyze shared callees?



General framework is parameterized, allowing algorithm-specific behavior

Context-Insensitive vs. Context-Sensitive Analysis

Context-insensitive analysis:

analyze each procedure once in shared context of all its callers

+ simple
– pass back same **smeared** result info to all callers

Context-sensitive analysis:

analyze callee procedure multiple times,
each time for a separate, more precise calling **context**

+ each version (**contour**) gets more precise analysis
+ pass back more precise result info to callers of that contour
– may be more expensive, harder to scale to big programs

How to capture calling context?

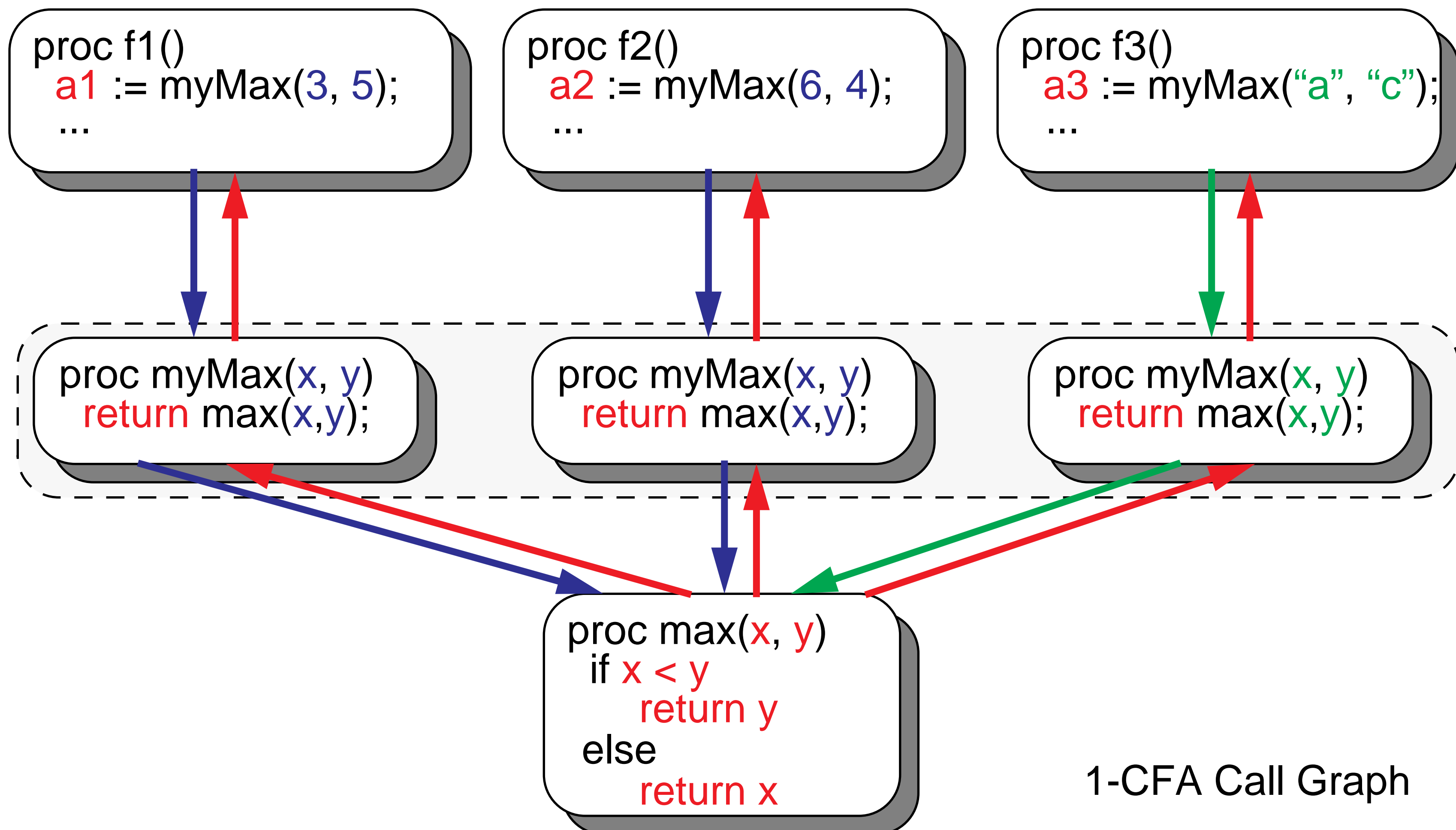
- calling context $\leftrightarrow$ callee contour

Contour Selection by Call Chain

Idea: analyze each procedure separately for each **vector of *k* enclosing callers**

Examples:

- Shivers's ***k*-CFA** family of context-sensitive analysis: 0-CFA, 1-CFA, etc.
- **unbounded-CFA** [e.g., Emami, Ghiya, & Hendren]

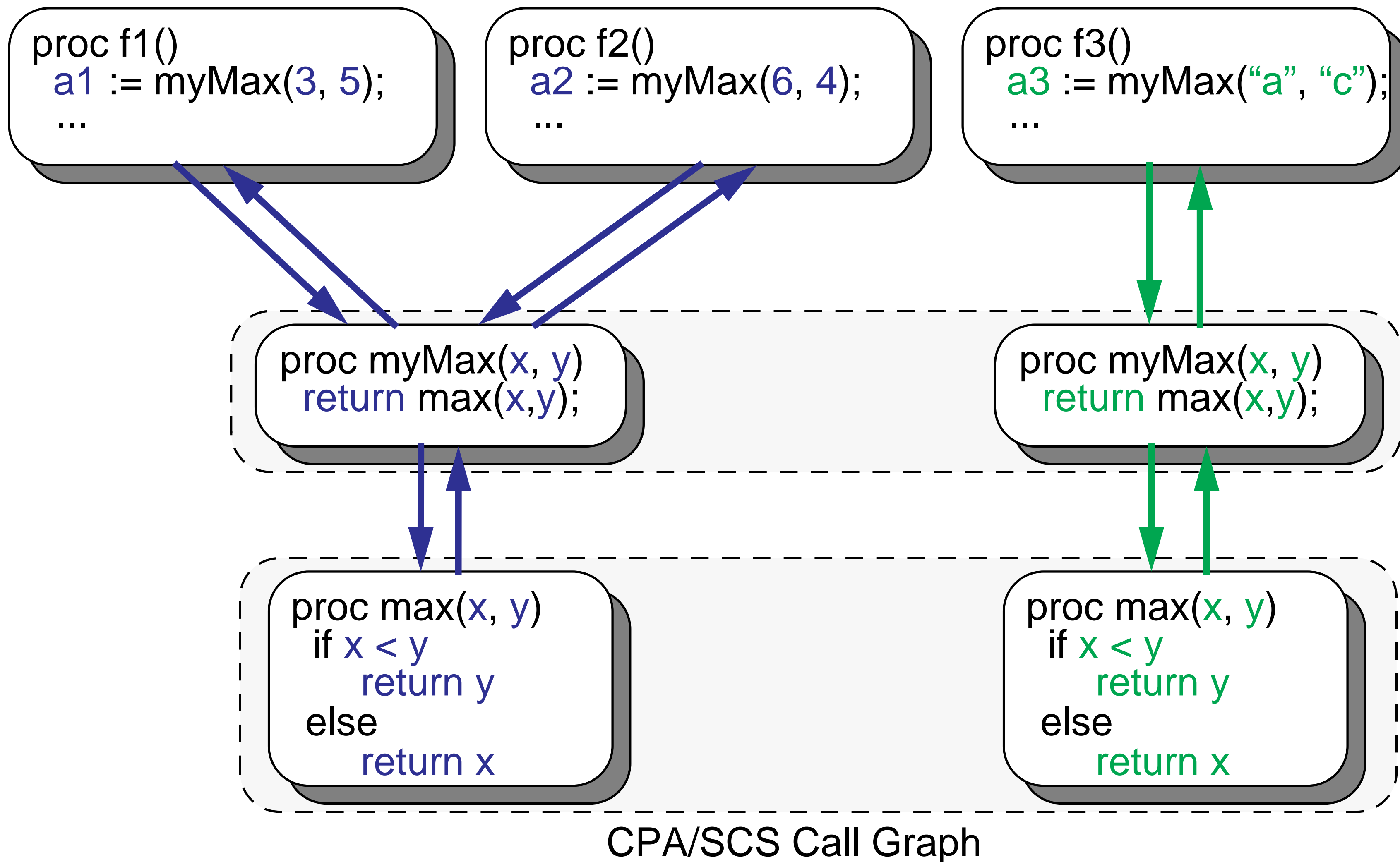


Contour Selection by Actual Parameters

Idea: reanalyze based on **actual class info** passed at that call site

Examples:

- Agesen's **Cartesian Product Algorithm (CPA)**
- Our **Simple Class Set Algorithm (SCS)**



CPA/SCS Call Graph

Instantiating Call Graph Construction Framework

To turn framework into specific algorithm:

- Choose an initial call graph construction method
- Choose a contour selection function (e.g., 0-CFA, 1-CFA, CPA, SCS…)
- Choose a spurious node/edge removal method (optional)

Framework also supports context-sensitive analysis of classes and instance variables in a similar fashion as context-sensitive analysis of procedures

Framework has been implemented in Vortex optimizing compiler

- 4,000 lines of shared code
- 100-300 additional lines per algorithm
- removing spurious node/edge component not implemented

Experimental Assessment

Goal: Evaluate costs and benefits on sizeable applications

What are the costs of different call graph construction algorithms?

- Analysis time
- Analysis space

What are the benefits of the resulting call graphs?

- Call graph precision
- Speed-up, resulting from interprocedural optimizations
- Compiled code space, from removing unreachable methods

How practical is interprocedural analysis?

Our Experiments

For sizeable programs (5K to 50K lines) in two different languages (Cecil, Java)

- Cecil: pure, heavy use of first class functions
- Java: hybrid, no first class functions

Compare to Vortex’s best non-interprocedural configuration [OOPSLA ‘96]

- includes sophisticated optimizations for message sends
- Java: 4X faster than Sun JDK1.0.2 JIT,
  2X faster than Toba 1.0.5 Java to C compiler

Focus on bottom-line metrics:

- Analysis time
- Speed-up
- Compiled code space

Summary of Results

	Analysis Time	Speed-Up	Compiled Code Space
Cecil			
flow insensitive	◉	●	○
limited flow-sensitive	◓	◒	○
context-insensitive	◒	◉	◓
context-sensitive	●	◉	◓
Java			
flow insensitive	◉	●	○
limited flow-sensitive	◓	●	○
context-insensitive	◓	◒	○
context-sensitive	○	◒	○

Excellent ◉ ◓ ○ ◒ ● Poor

Summary of Results

Programming language/style impacts cost/benefit

- Cecil
  - fast algorithms ineffective
  - can achieve large speed-up, but at a high cost
- Java
  - analysis time reasonable, but speed-up small

Scalability is a major concern for context-sensitive algorithms

Even imprecise algorithms enable substantial code space reduction

- Doing analysis actually reduces total compile time

Conclusions

Contributions:

- Unified model of call graph construction problem
- Experimental assessment using sizeable programs

Current work:

- Limited flow-sensitive algorithms [POPL ‘98]
- Extend algorithmic framework to better include linear-time algorithms
- Integrate with Vortex’s incremental recompilation

http://www.cs.washington.edu/research/projects/cecil/