Class 9

- Review; questions
- Discussion of Semester Project
- Arbitrary interprocedural control flow
- Assign (see Schedule for links)
  - Readings on pointer analysis
  - Problem Set 5: due 9/22/09
  - Project proposal
    - Initial: due by e-mail 9/22/09
    - Final: due (written, 2 pages) 9/29/09

Complicating Factors

A. Programs with more than one procedure
B. Recursion
C. Programs with arbitrary control flow
D. Programs with pointers
E. Programs with arrays

Review: Interprocedural Analysis

Approaches to performing analysis on programs with more than one procedure (data-flow, slicing, etc.) to preserve calling context

1. Compute summary information at the call sites
2. Keep the call stack information

Note: Analysis that preserve the calling context are *calling context-sensitive*; those that don't preserve the calling context are *calling context-insensitive*

Context-sensitive VS Context-insensitive

- Which is Weiser's slicing algorithm?
- Which is the SDG?

Complicating Factors

A. Programs with more than one procedure
B. Programs with recursion
C. Programs with arbitrary control flow
D. Programs with pointers
E. Programs with complex data structures

Recursion

Programs containing recursion result in additional complexity in the program-analysis algorithms. For example

- Recursion results in cycles in the interprocedural graph, making it difficult to order the nodes in the graph for processing
- Iterative algorithms may need significant processing when these cycles are present
- Etc.

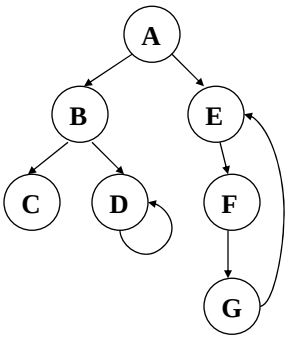
To accommodate analysis of programs with recursion, we can perform analysis on the interprocedural graph to identify cycles—similar to analysis to identify loops.

Recursion

A call graph can be used to represent the interactions among procedures (or modules) in a system.

In a call graph, nodes represent procedures and directed edges represent the interaction between procedures. An edge (M,N) in a call graph means that M calls N. If there is more than one call from M to N in M, the one edge between M and N represents all of them.

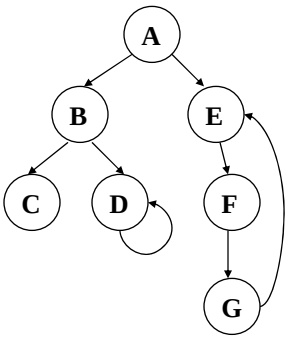
In the example, A calls B and E, B calls C and D, D calls itself, E calls F, F calls G, and G calls E.



Recursion

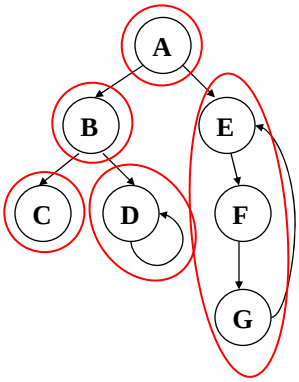
Additional information about call graphs

- The call graph defined here has one edge between two nodes even if there are multiple calls between the procedures represented by the nodes.
- A graph that contains an edge for every call between two nodes, and that then can contain information about parameters passed on each edge, is called a call multi-graph.



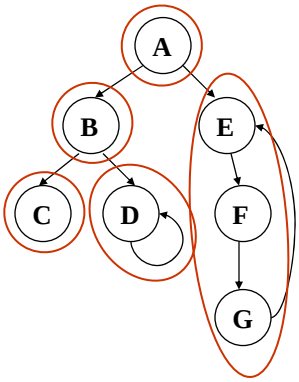
Recursion

- A common way to analyze an interprocedural graph is to first perform analysis to identify the strongly-connected components. A strongly-connected component is a set of nodes in the graph such that any node in the set is reachable from any other node. You can easily develop or find an algorithm to find strongly-connected components.
- In the call graph at right, the strongly-connected components are shown in red.



Recursion

- Given the interprocedural component with strongly-connected components identified, the analysis can proceed by
  - Ordering the traversal using the strongly-connected components
  - Completing the analysis within a strongly-connected component until it stabilizes; then moving onto the next node



Complicating Factors

A. Programs with more than one procedure
B. Programs with recursion
C. Programs with arbitrary control flow
D. Programs with pointers
E. Programs with complex data structures

Semantic Dependence

- Intuitively, n is *semantic dependent* on m if the semantics of m may affect the execution behavior of n
  - Important because semantic dependence is a necessary condition for certain semantic relationships between statements
  - However, no definitions of syntactic dependence are a sufficient condition for semantic dependence

→ Justification for approximated algorithms based on syntactic dependence

Control Dependence Revisited

- Two definitions of control dependence
  - Strong—termination of loops, number of times executed not considered (Ferrante, Ottenstein, and Warren)
  - Weak—doesn't assume termination of loops, etc. (Podgurski and Clarke)

Arbitrary Interprocedural CF

- Three ways in which intra-procedural control dependences can be inaccurate
  - Entry-dependence effect
  - Multiple-context effect
  - Return-dependence effect

Arbitrary Interprocedural Control Flow

What are intraprocedural control dependences?

```
procedure M
1.  begin M
2.    read i, j
3.    sum := 0
4.    while i < 10 do
5.      call B
      endwhile
6.    no-op
7.    print sum
8.  end M
```

```
procedure B
9.  begin B
10.   call C
11.   if j >= 0 then
12.     sum := sum + j
13.     read j
      endif
14.   i := i + 1
15. end B
```

```
procedure C
16. begin C
17.   if sum > 100 then
18.     print("error")
      endif
19. end C
```

Intraprocedural CD	
Statements	**CD on**
2, 3, 4, 6, 7	
10, 11, 14	
17	
4, 5	
12, 13	
18	

Arbitrary Interprocedural Control Flow

Entry-dependence effect

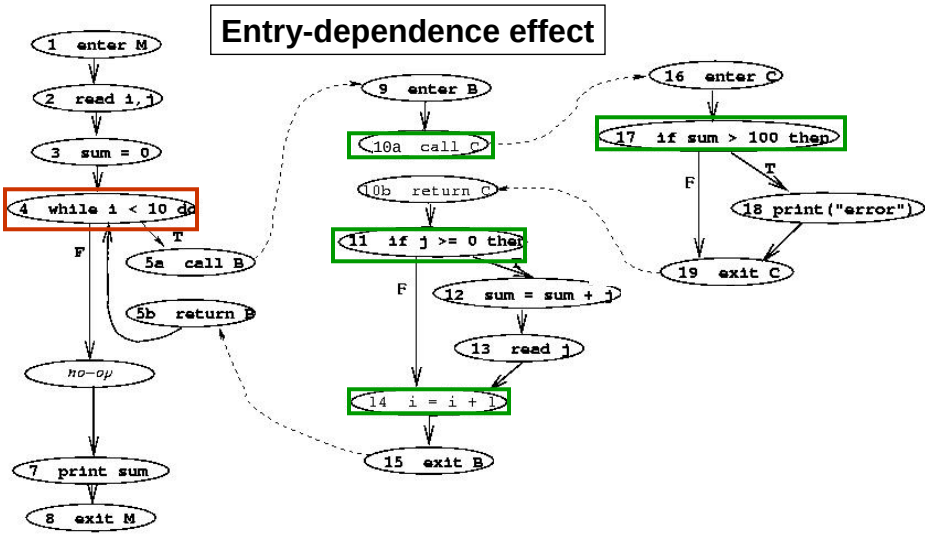
```
procedure M
1.  begin M
2.    read i, j
3.    sum := 0
4.    while i < 10 do
5.      call B
      endwhile
6.    no-op
7.    print sum
8.  end M
```

```
procedure B
9.  begin B
10.   call C
11.   if j >= 0 then
12.     sum := sum + j
13.     read j
      endif
14.   i := i + 1
15. end B
```

```
procedure C
16. begin C
17.   if sum > 100 then
18.     print("error")
      endif
19. end C
```

Arbitrary Interprocedural Control Flow

Entry-dependence effect



Arbitrary Interprocedural Control Flow

Multiple-context effect

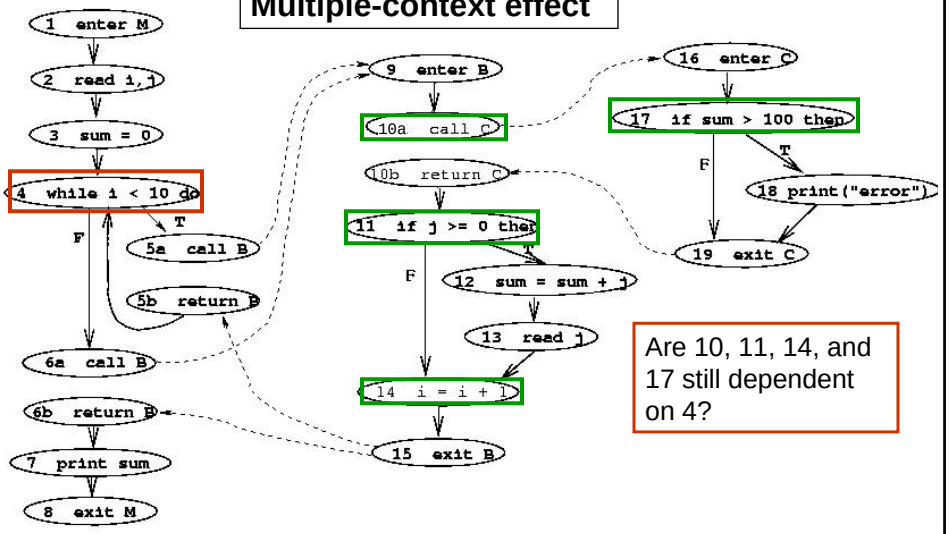
```
procedure M
1.  begin M
2.     read i, j
3.     sum := 0
4.     while i < 10 do
5.        call B
       endwhile
6.     call B
7.     print sum
8.  end M
```

```
procedure B
9.  begin B
10.    call C
11.    if j >= 0 then
12.       sum := sum + j
13.       read j
       endif
14.    i := i + 1
15. end B
```

```
procedure C
16. begin C
17.    if sum > 100 then
18.       print("error")
       endif
19. end C
```

Arbitrary Interprocedural Control Flow

Multiple-context effect



Are 10, 11, 14, and 17 still dependent on 4?

Arbitrary Interprocedural Control Flow

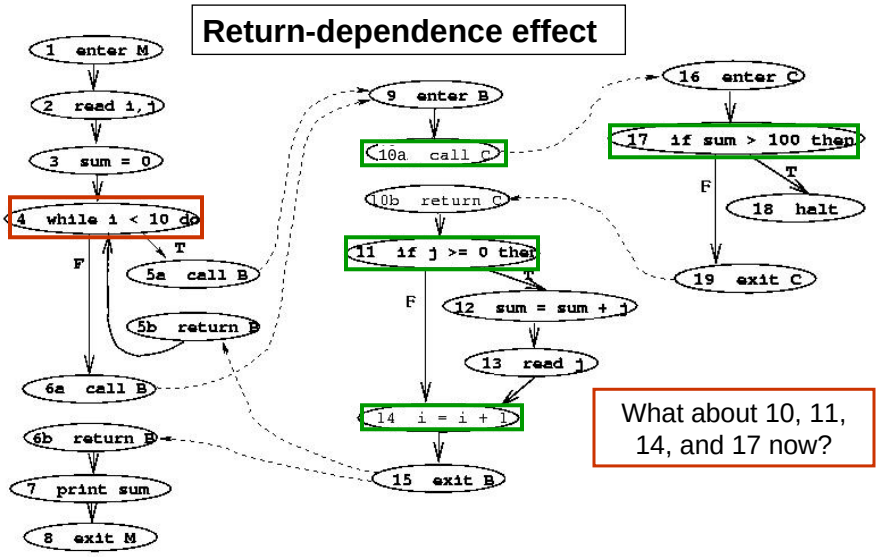
Return-dependence effect

```
procedure M
1.  begin M
2.    read i, j
3.    sum := 0
4.    while i < 10 do
5.      call B
      endwhile
6.    call B
7.    print sum
8.  end M
```

```
procedure B
9.  begin B
10.   call C
11.   if j >= 0 then
12.     sum := sum + j
13.     read j
      endif
14.   i := i + 1
15. end B
```

```
procedure C
16. begin C
17.   if sum > 100 then
18.     halt
      endif
19. end C
```

Arbitrary Interprocedural Control Flow



Arbitrary Interprocedural Control Flow

procedure M
1. begin M
2. read i, j
3. sum := 0
4. while i < 10 do
5. call B
endwhile
6. *no-op*
7. print
8. end M

procedure B
9. begin B
10. call C
11. if j >= 0 then
12. sum := sum + j
13. read j
endif

procedure C
16. begin C
17. if sum > 100 then
18. halt
endif
19. end C

Intraprocedural CD	
Statements	**CD on**
2, 3, 4, 6, 7	Entry M
10, 11, 14	Entry B
17	Entry C
4, 5	4
12, 13	11
18	17

Interprocedural CD	
Statements	**CD on**
2, 3, 4	Entry M
4, 7, 11, 14, 18	17
5, 6, 10, 17	4
12, 13	11

Instances of Arbitrary Interprocedural Control Flow

- Exception-handling constructs
  - throw-catch construct in Java
  - raise-exception when construct in Ada
- Halt statements
  - exit() call in C, C++
  - System.exit() call in Java
- Interprocedural jump statements
  - setjmp()-longjmp() calls in C and C++

Arbitrary Interprocedural Control Flow

Sinha, Harrold, Rothermel paper

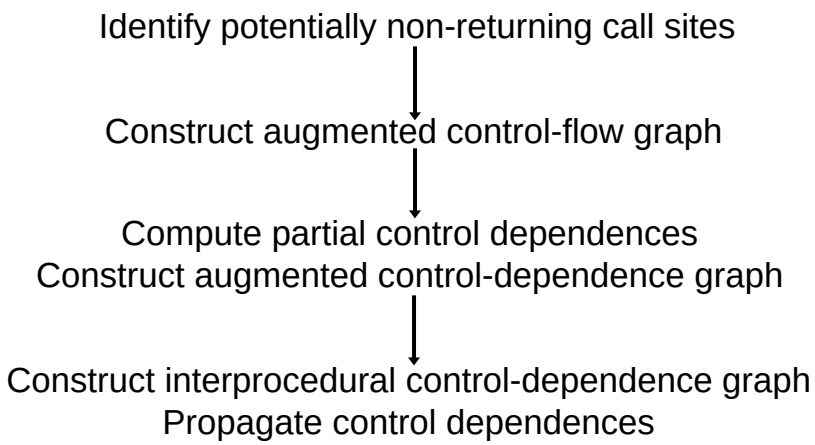
- Contributions
  - Ways that intraprocedural CD inaccurately model CDs in whole programs
  - Precise definition of interprocedural CD
  - Approaches for computing interprocedural CD
  - Empirical results suggesting effectiveness and efficiency
- Later work
  - Extensions to handle other types of interprocedural CD such as longjumps, exception-handling constructs

First Approach: Interprocedural Inlined Flow Graph (IIFG)

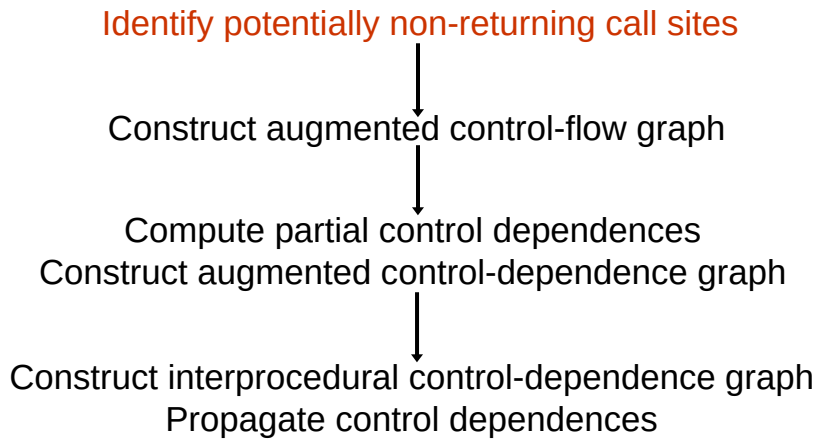
- Each procedure inlined at each call site
- Precise computation of dependences by adapting approaches defined for the intra-procedural case, but
  - Possibly infinite
  - Exponential in size in the worst case

→ Second approach (less precise)

Computation of Interprocedural CD



Computation of Interprocedural CD



PNRC Analysis

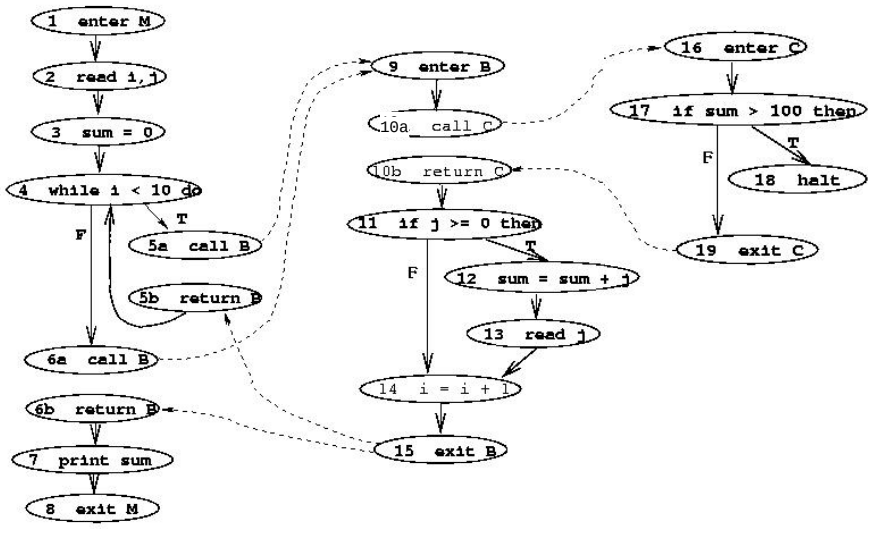
- Step 1: Identifies three sets
  - DNRPList: Definitely non-returning procedures
  - UnreachList: Statically unreachable nodes
  - HNList: Halt statements reachable from entry
- Method
  - Build ICFG
  - Depth first traversal along realizable paths marking visited nodes
    - Unmarked nodes are unreachable
    - Unmarked exit nodes indicate DNRPs
    - Marked halt nodes indicate reachable halts

PNRC Analysis

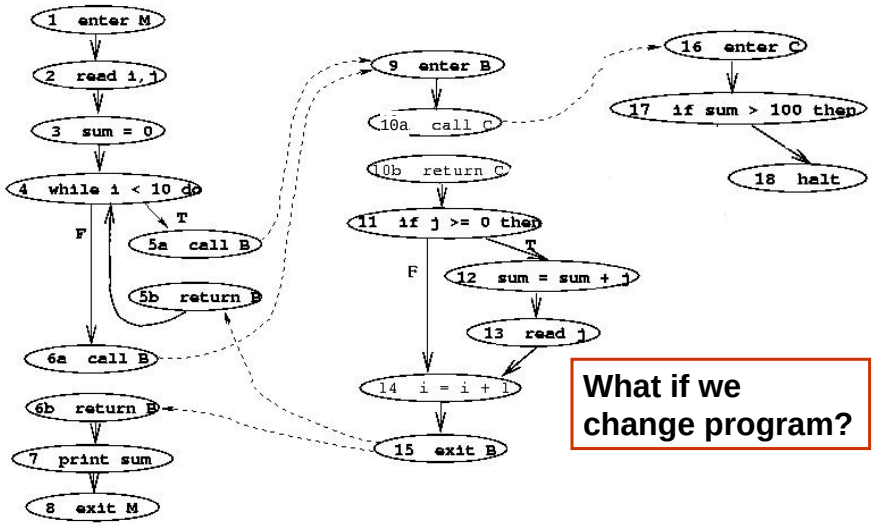
- Step 1: Identifies three sets
  - DNRPList: Definitely non
  - UnreachList: Statically ur
  - HNList: Halt statements reach rom entry
- Method
  - Build ICFG
  - Depth first traversal along realizable paths marking visited nodes
    - Unmarked nodes are unreachable
    - Unmarked exit nodes indicate DNRPs
    - Marked halt nodes indicate reachable halts

What's a *realizable path?*

PNRC Analysis



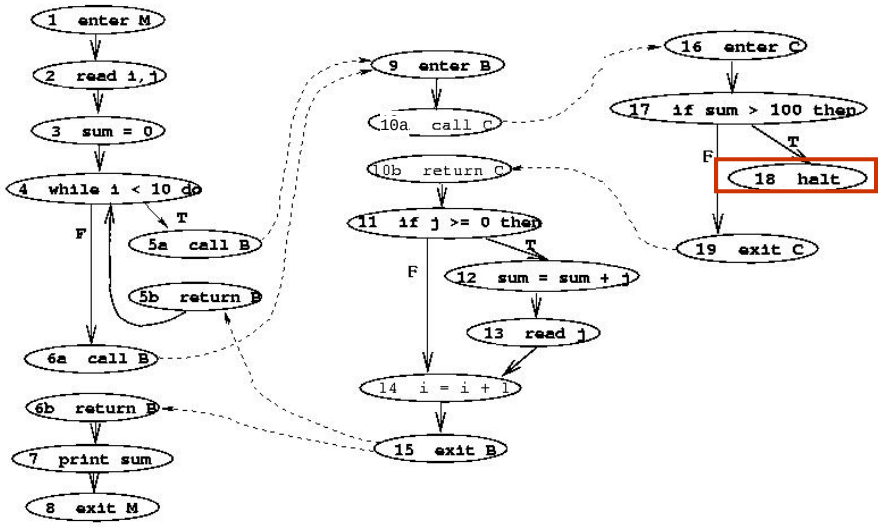
PNRC Analysis



PNRC Analysis

- Step 2: Compute partial CD
  - Identify PNRCList: Possibly non-returning call-sites
  - Build ACFGs
- Method
  - Backward traversal of ICFG starting from (1) halt nodes and (2) calls to DNRPs
    - Ascending into callers, but not descending into callees
  - Any call site reached is a PNRC

PNRC Analysis



PNRC Analysis