CS711 Advanced Programming Languages

Inter-Procedural Analysis

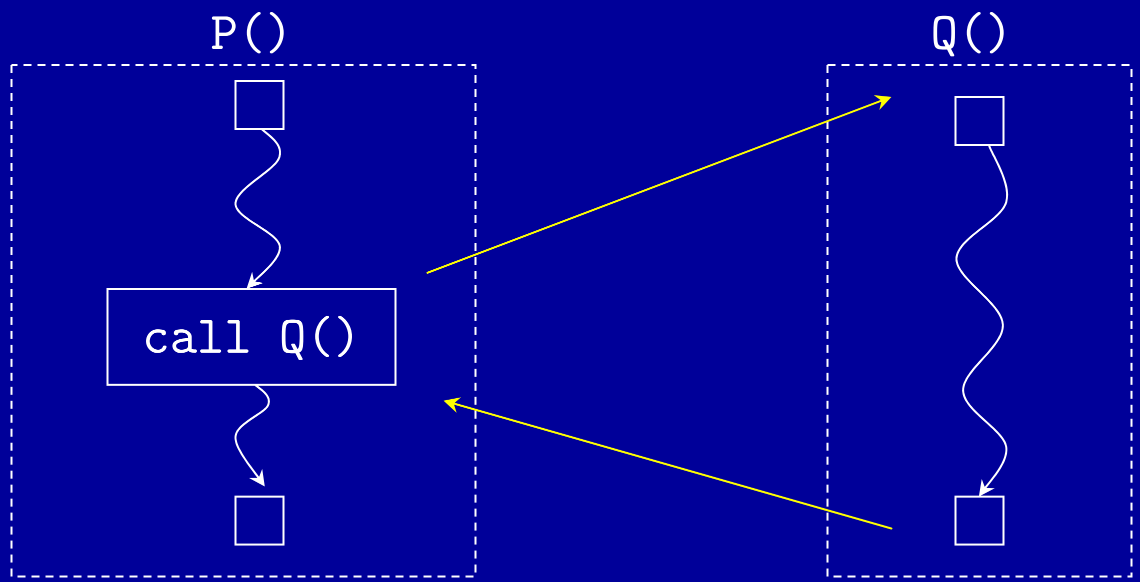
Radu Rugina

1 Sep 2005

Inter-Procedural Analysis

- Standard intra-procedural dataflow analysis:
  - Build flow graph, propagate dataflow facts
  - Assumes no procedure calls
  - Or uses worst-case assumptions about procedure calls
- Inter-procedural analysis
  - Analyze procedure interactions more precisely
  - Difficult to do it efficiently and precisely

The problem



- Transfer function of call = analysis of callee's body
- Two quick 'solutions" to this problem

Quick Solution 1: Inlining

- Inline callees into callers
  - End up with one big procedure
  - CFGs of individual procedures = duplicated many times
- Good: it is precise
  - distinguishes between different calls to the same function
- Bad: exponential blow-up, not efficient

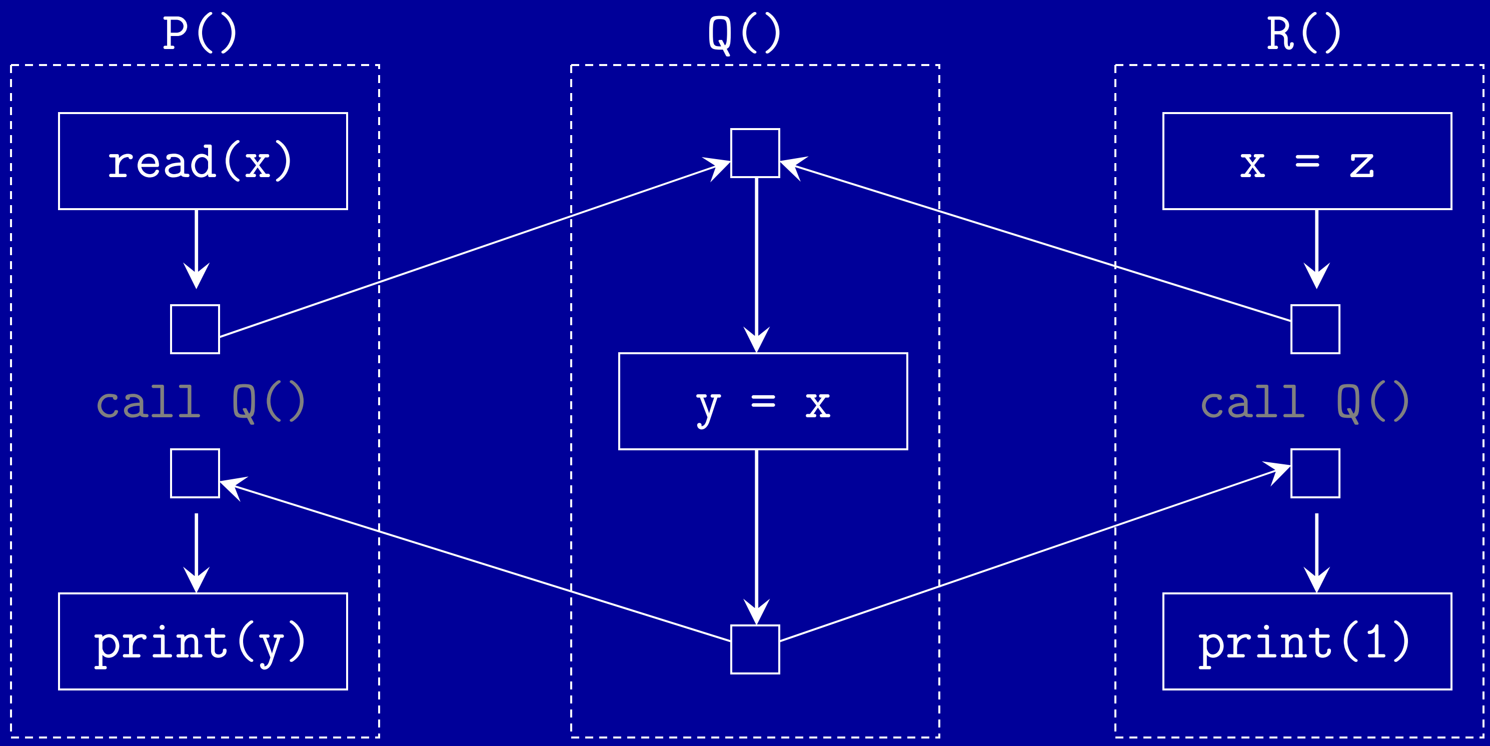
```
main() { f(); f(); }
f() { g(); g(); }
g() { h(); h(); }
h() { ... }
```

- Bad: doesn't work with recursion

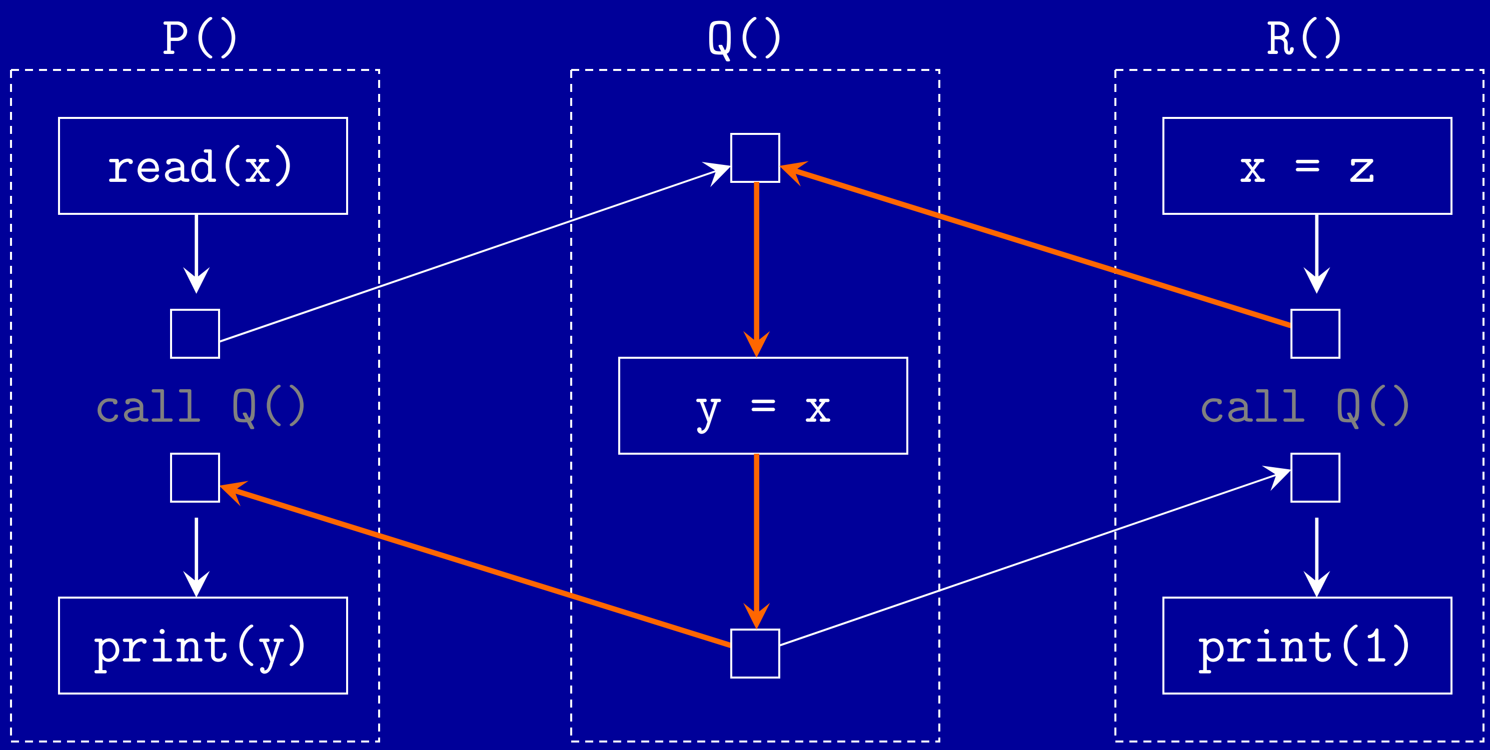
Quick Solution 2: Extend CFG

- Build a "supergraph" = inter-procedural CFG
- Replace each call from P to Q
  - An edge from point before the call (call point) to Q's entry point
  - An edge from Q's exit point to the point after the call (return pt)
  - If necessary, add assignments of actuals to formals, and assignment of return value
- Good: efficient
  - Graph of each function included exactly once in the supergraph
  - Works for recursive functions (although local variables need additional treatment)
- Bad: imprecise, "context-insensitive"
  - The "unrealizable paths problem": dataflow facts can propagate along infeasible control paths

Unrealizable Paths



Unrealizable Paths



DFA Review

- CFG with nodes $n \in N$
- Dataflow facts: $d \in L$ (lattice)
- Transfer function: $[\![n]\!] : L \rightarrow L$
- MFP (maximal fixed point) solution = greatest solution of:

  $X(n) = d_0$, if n = entry

  $X(n) = \sqcap \{ [\![m]\!] \, X(m) \mid m \in preds(n) \}$
- MOP (meet-over-paths) solution:

  $MOP(n) = \sqcap \{ ([\![p_k]\!] \circ \ldots \circ [\![p_1]\!] \circ [\![p_0]\!]) \, (d_0) \mid p_0 \, p_1 \ldots p_k$ is a path to $n\}$
- Safe: $MOP \sqsubseteq MFP$
- Precise if transfer functions are distributive: $MOP = MFP$

Inter-Procedural DFA

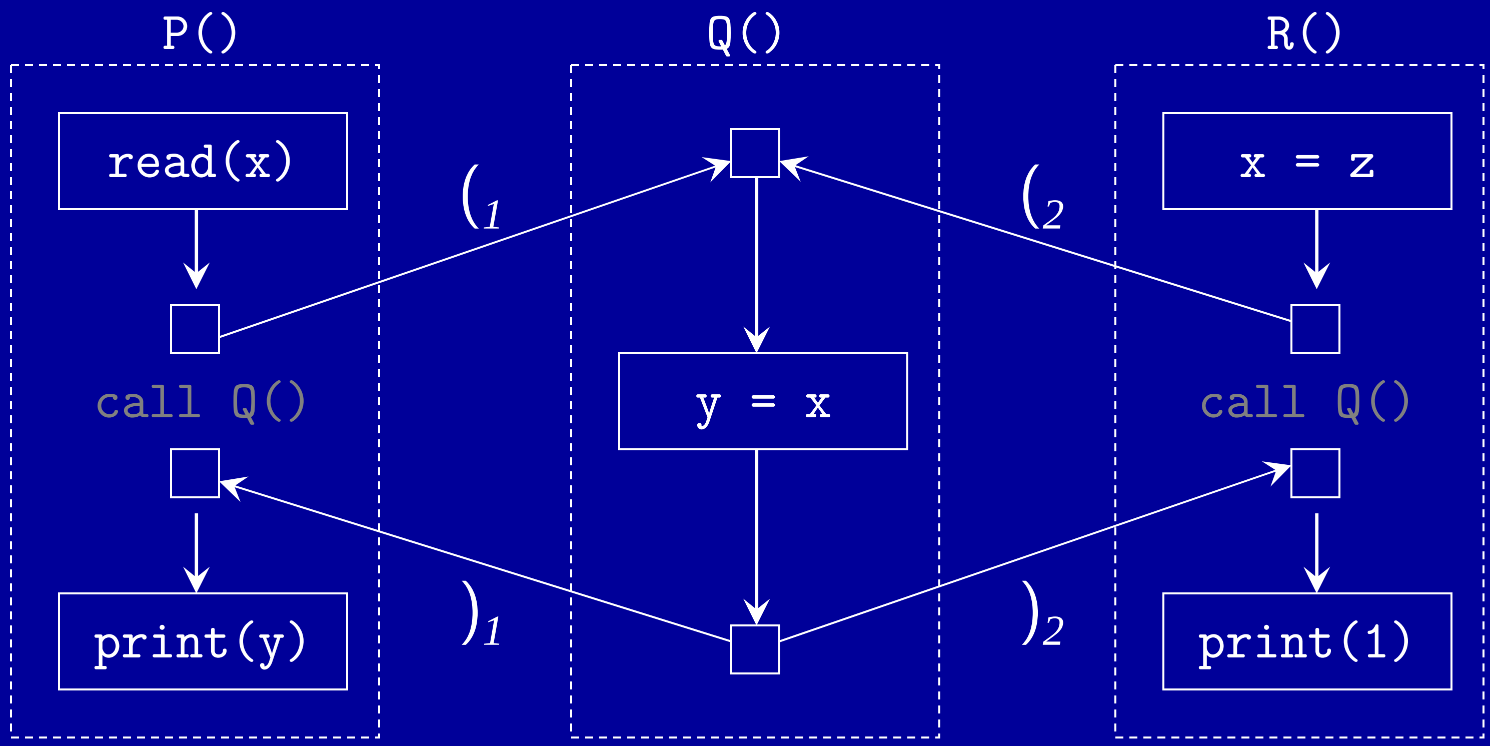
- Consider the supergraph
- Additionally, for each call i :
  - label call → entry edge with $(_i$
  - label exit → return edge with $)_i$
- Consider only valid paths through the supergraph:

$$\begin{aligned} \text{matched} &::= \text{matched } (_i \text{ matched })_i \quad | \quad \varepsilon \\ \text{valid} &::= \text{valid } (_i \text{ matched} \quad | \quad \text{matched} \end{aligned}$$

- MOVP = meet-over-valid-paths

$$\mathrm{MOVP}(n) = \sqcap \{ ([\![p_k]\!] \circ \ldots \circ [\![p_1]\!] \circ [\![p_0]\!]) (d_0) \mid p_0\, p_1 \ldots p_k \text{ is a valid path to } n \}$$

Valid Paths



IFDS Problems

- Finite subset, distributive problems:
  - Lattice: $L = 2^D$ for some finite set D
  - Partial order is $\subseteq$, meet is $\cup$
  - Transfer functions are distributive

- A precise, efficient solution to IPA for such dataflow problems

  1: an encoding of transfer functions

  2: a formulation of the problem using CFL reachability

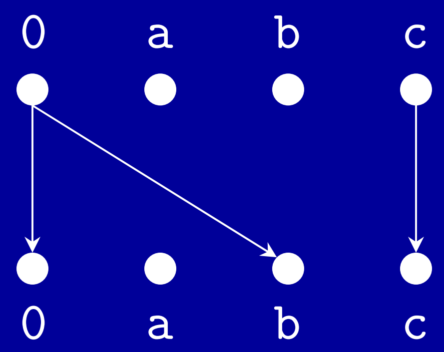
  3: an efficient CFL reachability algorithm for the matched parentheses grammar

Transfer Function Encoding

- Enumerate all input space and output space
- Represent functions as graphs with 2(D+1) nodes
- Use a special symbol "0" to describe empty sets

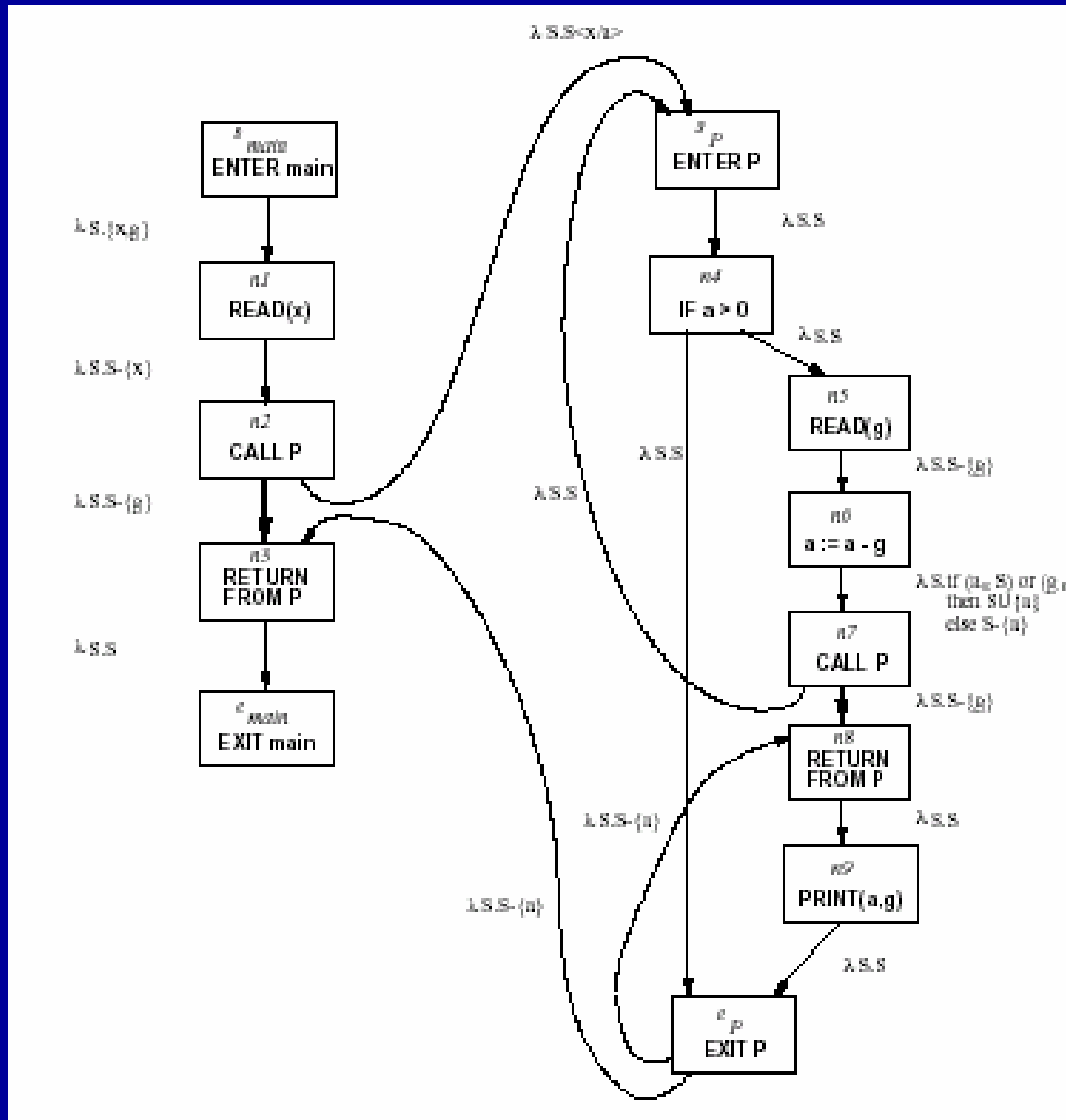
- Example: $D = \{ a, b, c \}$

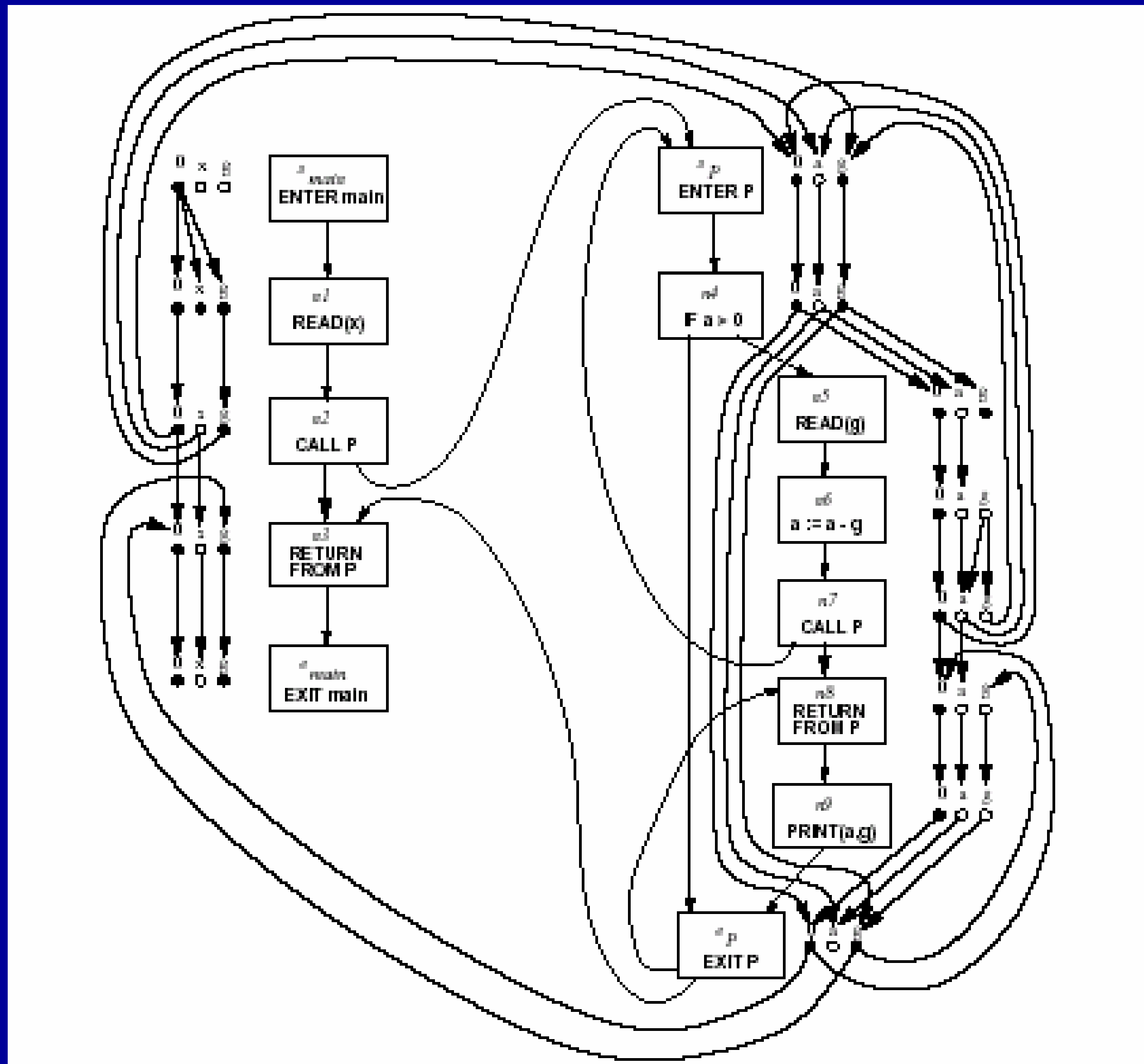
  $f(S) = (S - \{ a \}) \cup \{ b \}$



Exploded Supergraph

- Exploded supergraph:
  - Start with supergraph
  - Replace each node by its graph representation
  - Add edges between corresponding elements in D at consecutive program points
- CFL reachability:
  - Finding MOVP solution is equivalent to computing CFL reachability over the exploded supergraph using the valid parentheses grammar.


λ S.S<x/a>
ENTER main
λ S.{x,g}
n1
READ(x)
λ S.S-{x}
n2
CALL P
λ S.S-{g}
n3
RETURN FROM P
λ S.S
EXIT main
ENTER P
λ S.S
n4
IF a > 0
λ S.S
n5
READ(g)
λ S.S-{g}
n6
a := a - g
n7
CALL P
λ S.S-{g}
n8
RETURN FROM P
λ S.S
n9
PRINT(a,g)
λ S.S
EXIT P
λ S.S-{a}
λ S.S-{a}



ENTER main
n1
READ(x)
n2
CALL P
n3
RETURN FROM P
EXIT main
ENTER P
n4
IF a > 0
n5
READ(g)
n6
a := a - g
n7
CALL P
n8
RETURN FROM P
n9
PRINT(a,g)
EXIT P

The Tabulation Algorithm

- Worklist algorithm, start from entry of “main”
- Keep track of:
  - Path edges: matched paren paths from procedure entry
  - Summary edges: matched paren call-return paths
- At each instruction:
  - Propagate facts using transfer functions; extend path edges
- At each call:
  - Propagate to procedure entry, start with an empty path
  - If a summary for that entry exits, use it
- At each exit:
  - Store paths from corresponding call points as summary paths
  - When a new summary is added, propagate to the return node

Complexity

- Polynomial-time complexity
  - Recall that inlining is exponential

- Inter-procedural: $O(ED^3)$
  - $E =$ number of edges
  - $D =$ size of the dataflow set

- Locally-separable (bit-vector): $O(ED)$

Experiments

Example	Tabulation Algorithm (realizable paths)		Naive Algorithm (any path)	
	Time (sec.)	Reported uses of possibly uninitialized variables	Time (sec.)	Reported uses of possibly uninitialized variables
struct-beauty	4.83+0.75	543	1.58+0.04	583
C-parser	0.70+0.19	11	0.54+0.02	127
ratfor	3.15+0.58	894	1.46+0.04	998
twig	5.45+1.20	767	5.04+0.11	775