Interprocedural Data Flow Analysis

Uday P. Khedker

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

October 2009

Part 1

About These Slides

Copyright

These slides constitute the lecture notes for CS618 Program Analysis course at IIT Bombay and have been made available as teaching material accompanying the book:

- Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press (Taylor and Francis Group). 2009.

Apart from the above book, some slides are based on the material from the following books

- S. S. Muchnick and N. D. Jones. *Program Flow Analysis*. Prentice Hall Inc. 1981.

Outline

- Issues in interprocedural analysis
- Functional approach
- The classical call strings approach
- Modified call strings approach

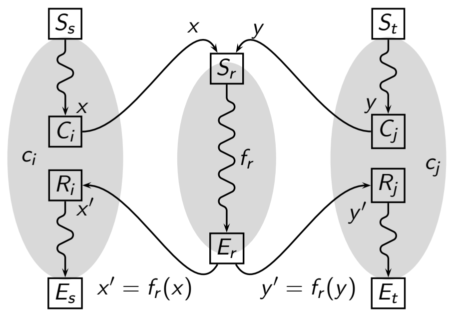
Part 3

Issues in Interprocedural Analysis

Interprocedural Analysis: Overview

- Extends the scope of data flow analysis across procedure boundaries
  Incorporates the effects of
  - procedure calls in the caller procedures, and
  - calling contexts in the callee procedures.
- Approaches :
  - Generic : Call strings approach, functional approach.
  - Problem specific : Alias analysis, Points-to analysis, Partial redundancy elimination, Constant propagation

Inherited and Synthesized Data Flow Information



Data Flow Information	
x	Inherited by procedure r from call site c_i in procedure s
y	Inherited by procedure r from call site c_j in procedure t
x'	Synthesized by procedure r in s at call site procedure c_i
y'	Synthesized by procedure r in t at call site procedure c_j

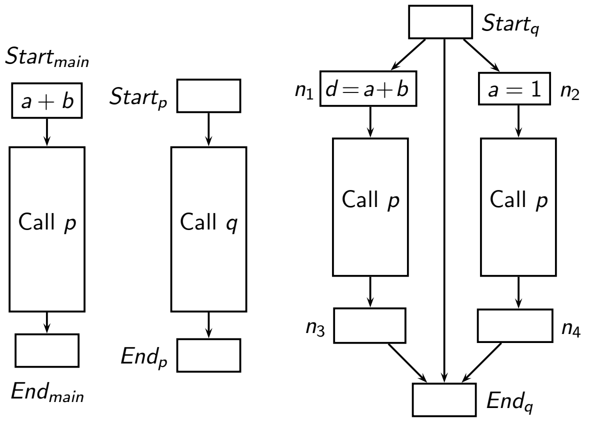
Inherited and Synthesized Data Flow Information

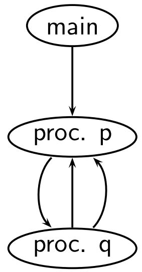
- Example of uses of inherited data flow information

  Answering questions about formal parameters and global variables:
  - Which variables are constant?
  - Which variables aliased with each other?
  - Which locations can a pointer variable point to?
- Examples of uses of synthesized data flow information

  Answering questions about side effects of a procedure call:
  - Which variables are defined or used by a called procedure? (Could be local/global/formal variables)
- Most of the above questions may have a *May* or *Must* qualifier.

Program Rrepresentation for Interprocedural Data Flow Analysis: Call Multi-Graph

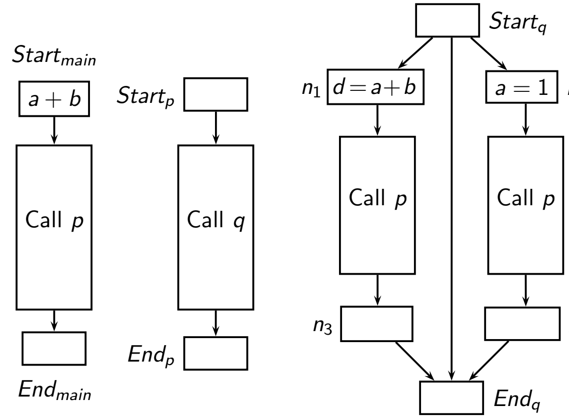




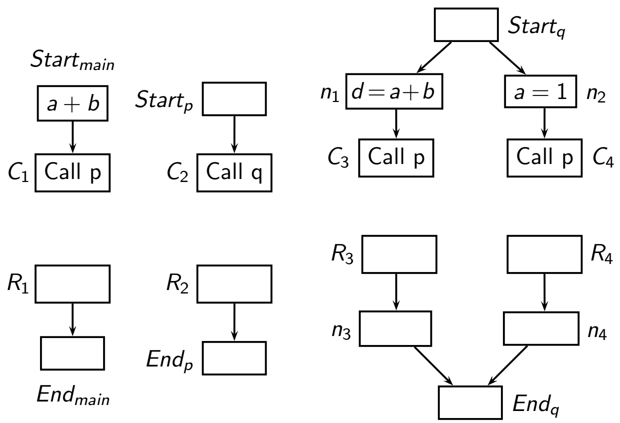
Supergraphs of procedures

Call multi-graph

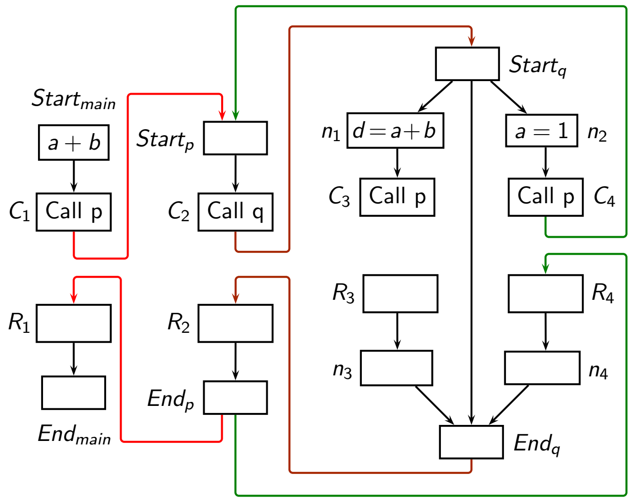
Program Rrepresentation for Interprocedural Data Flow Analysis: Supergraph



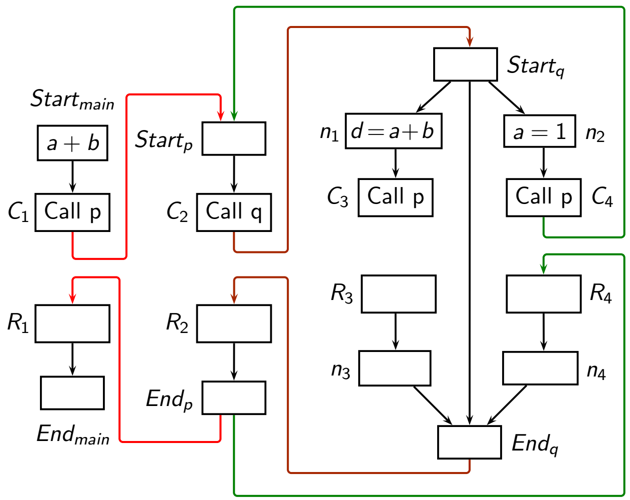
Program Rrepresentation for Interprocedural Data Flow Analysis: Supergraph



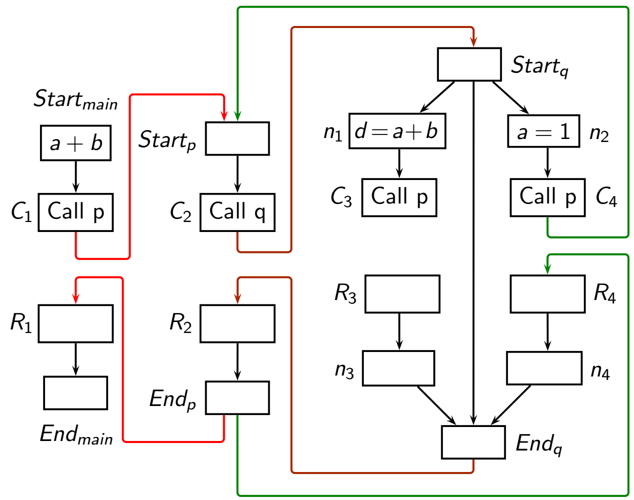
Program Rrepresentation for Interprocedural Data Flow Analysis: Supergraph



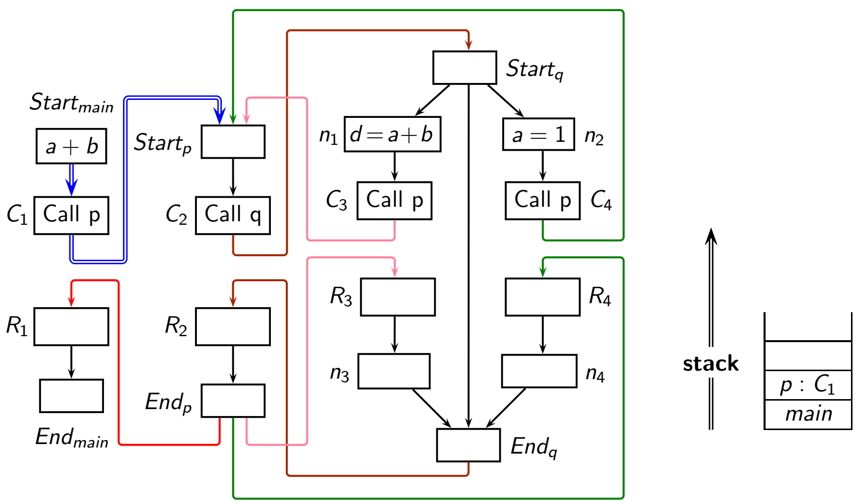
Program Rrepresentation for Interprocedural Data Flow Analysis: Supergraph



Program Rrepresentation for Interprocedural Data Flow Analysis: Supergraph

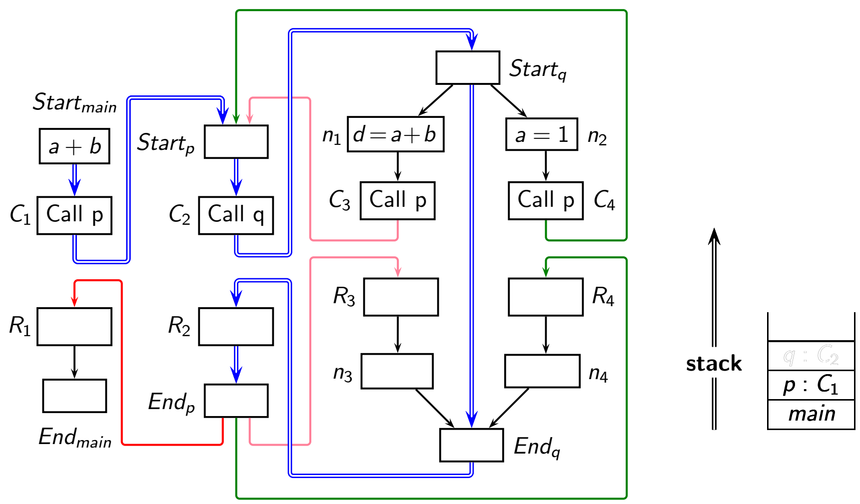


Validity of Interprocedural Control Flow Paths



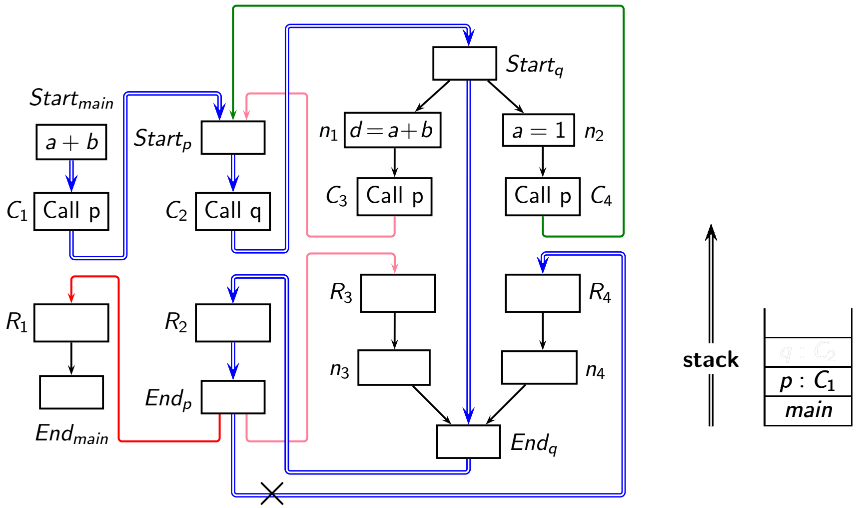
Interprocedurally valid control flow path

Validity of Interprocedural Control Flow Paths



Interprocedurally valid control flow path

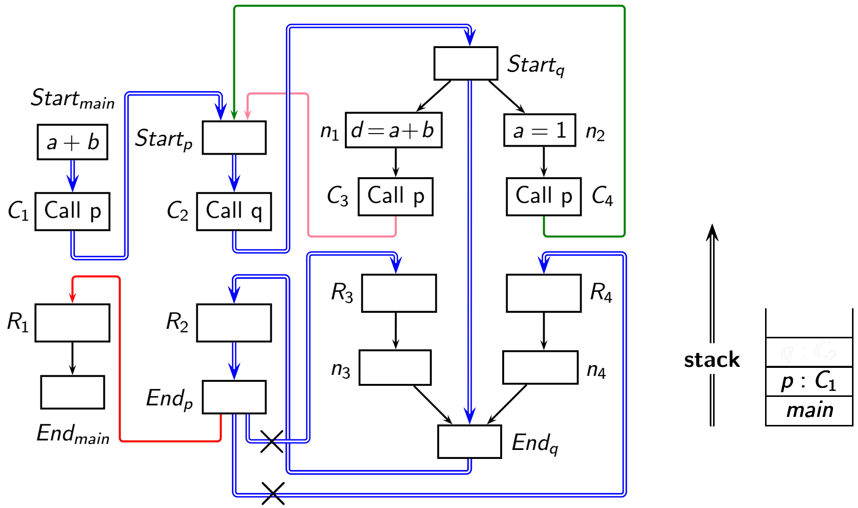
Validity of Interprocedural Control Flow Paths



Interprocedurally invalid control flow path

CS 618 Interprocedural DFA: Issues in Interprocedural Analysis 8/86

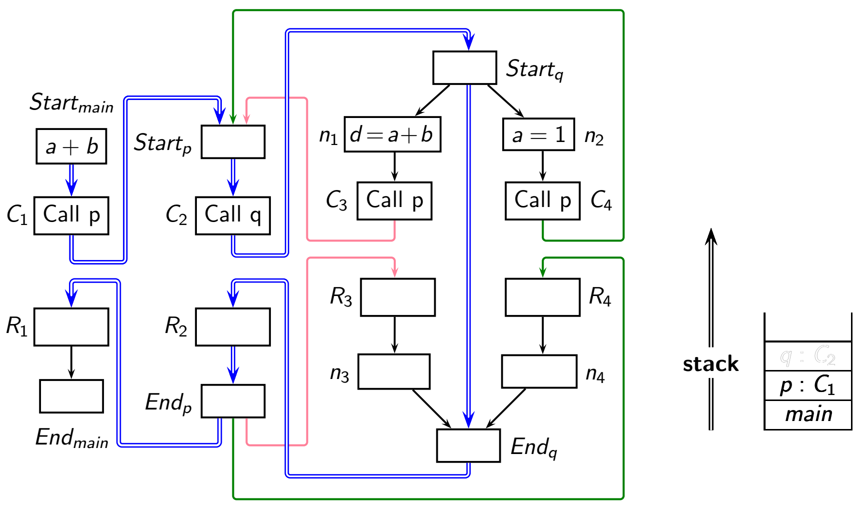
Validity of Interprocedural Control Flow Paths



Interprocedurally invalid control flow path

CS 618 Interprocedural DFA: Issues in Interprocedural Analysis 8/86

Validity of Interprocedural Control Flow Paths



Interprocedurally valid control flow path

Safety, Precision, and Efficiency of Data Flow Analysis

A path which represents legal control flow

- Data flow analysis uses static representation of programs to compute summary information along paths
- *Ensuring Safety*. All valid paths must be covered
- *Ensuring Precision*. Only valid paths should be covered.
- *Ensuring Efficiency*. Only relevant valid paths should be covered.

Subject to merging data flow values at shared program points without creating invalid paths

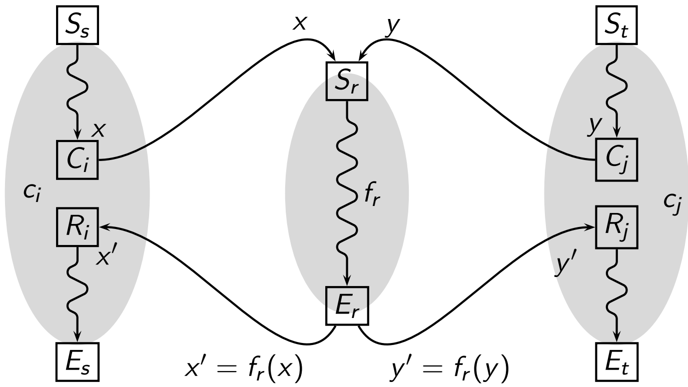
A path which yields information that affects the summary information.

Flow and Context Sensitivity

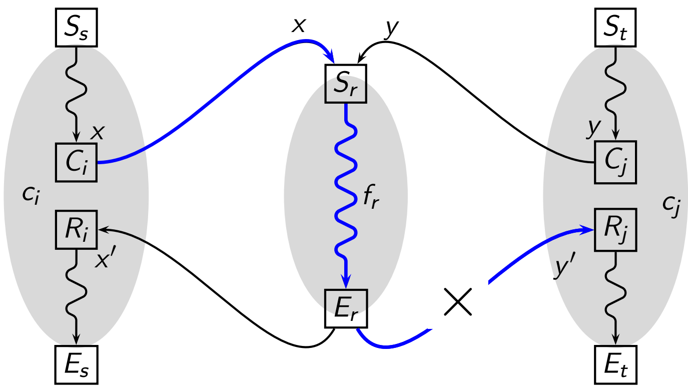
- Flow sensitive analysis:
  Considers intraprocedurally valid paths
- Context sensitive analysis:
  Considers interprocedurally valid paths
- For maximum statically attainable precision, analysis must be both flow and context sensitive.

MFP computation restricted to valid paths only

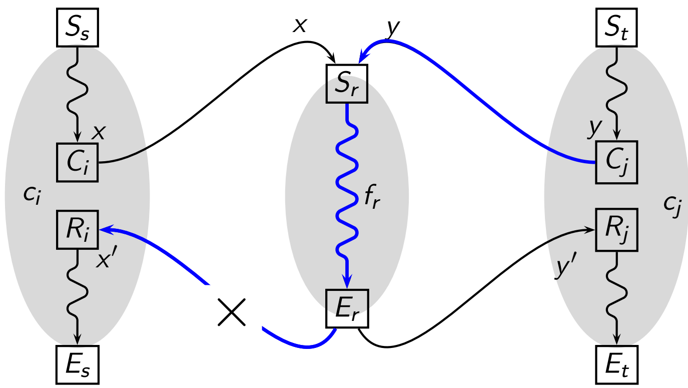
Context Sensitivity in Interprocedural Analysis



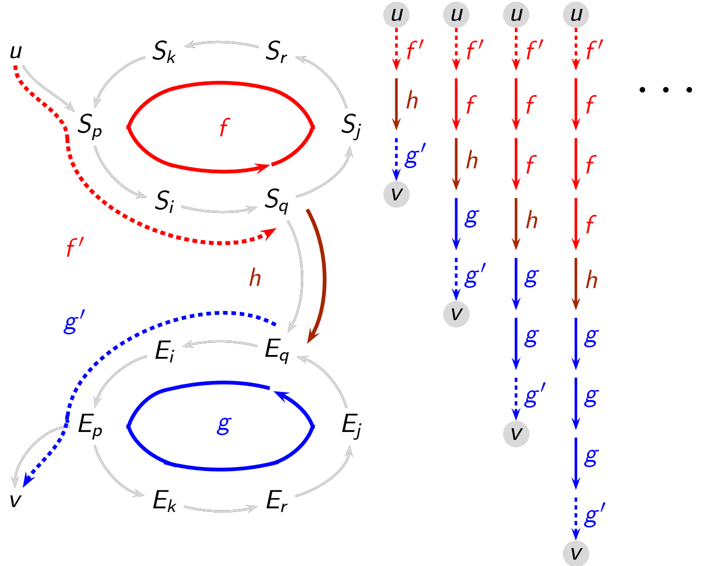
Context Sensitivity in Interprocedural Analysis



Context Sensitivity in Interprocedural Analysis

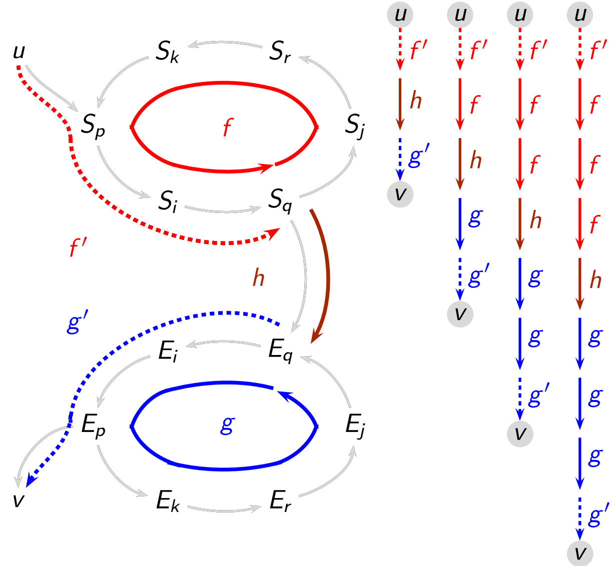


Context Sensitivity in Presence of Recursion



CS 618 Interprocedural DFA: Issues in Interprocedural Analysis 12/86

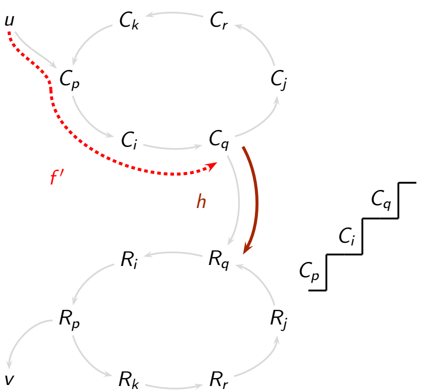
Context Sensitivity in Presence of Recursion



- For a path from u to v, g must be applied exactly the same number of times as f.
- For a prefix of the above path, g can be applied only at most as many times as f.

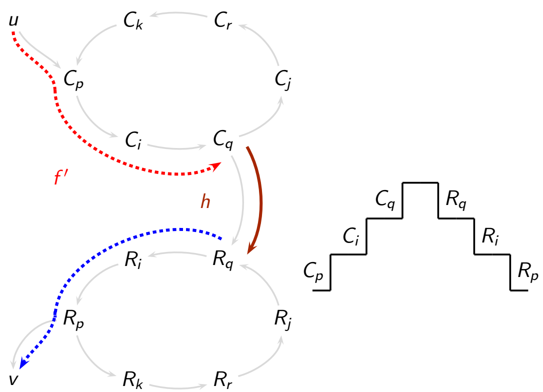
Oct 2009 IIT Bombay

Staircase Diagrams of Interprocedurally Valid Paths



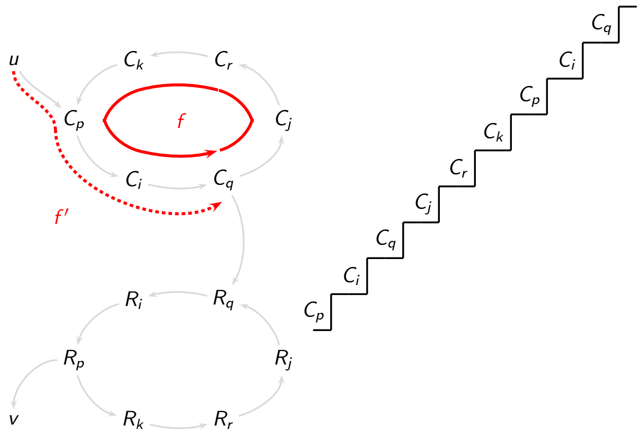
CS 618 Interprocedural DFA: Issues in Interprocedural Analysis 13/86

Staircase Diagrams of Interprocedurally Valid Paths



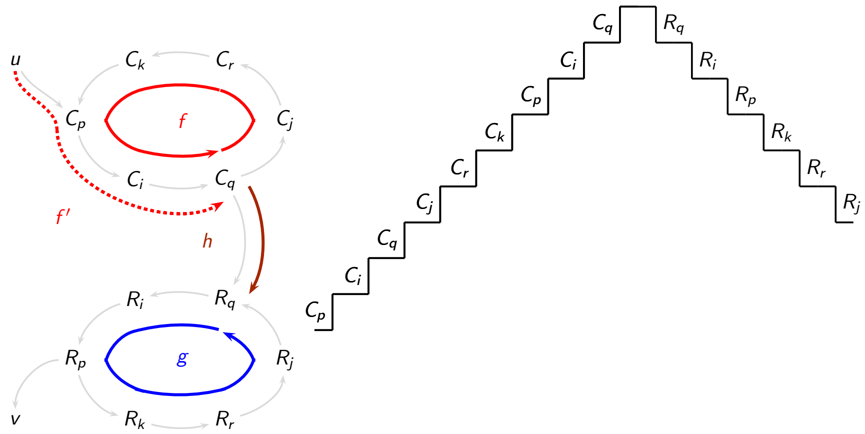
Oct 2009 IIT Bombay

Staircase Diagrams of Interprocedurally Valid Paths

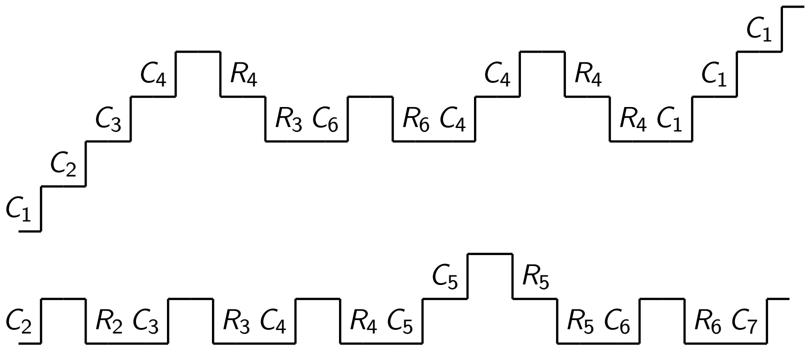


CS 618 Interprocedural DFA: Issues in Interprocedural Analysis 13/86

Staircase Diagrams of Interprocedurally Valid Paths



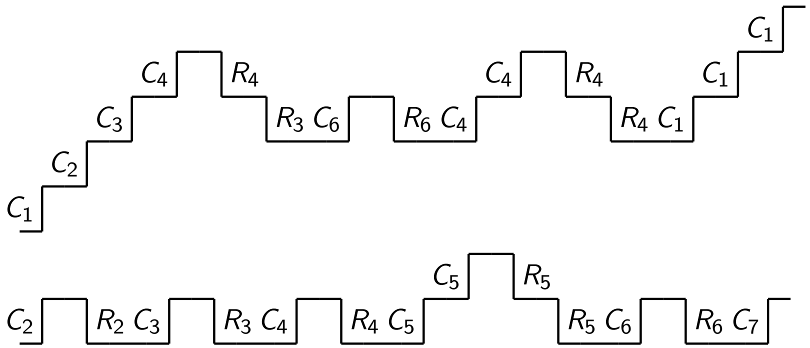
Staircase Diagrams of Interprocedurally Valid Paths



- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step.

CS 618 Interprocedural DFA: Issues in Interprocedural Analysis 14/86

Staircase Diagrams of Interprocedurally Valid Paths



- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step.

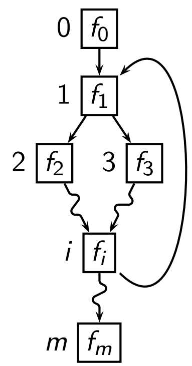
Oct 2009 IIT Bombay

Flow Insensitivity in Data Flow Analysis

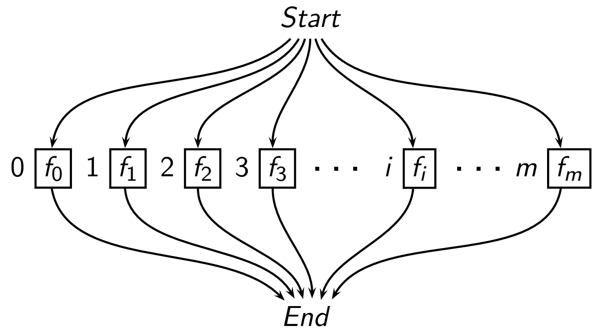
- Assumption: Statements can be executed in any order.
- Instead of computing point-specific data flow information, summary data flow information is computed.
  The summary information is required to be a safe approximation of point-specific information for each point.
- $\text{Kill}_n(\text{x})$ component is ignored.
  If statement n kills data flow information, there is an alternate path that excludes n.

Flow Insensitivity in Data Flow Analysis

Assuming that $DepGen_n(\text{x}) = \emptyset$, and $\text{Kill}_n(X)$ is ignored for all n



Control flow graph



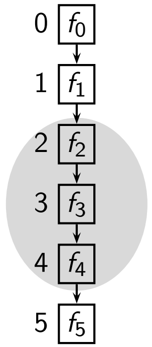
Flow insensitive analysis

Function composition is replaced by function confluence

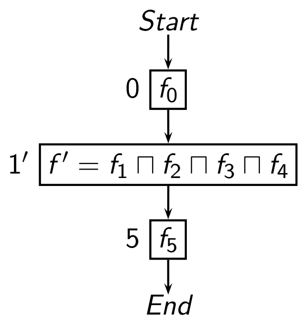
Flow Insensitivity in Data Flow Analysis

If $DepGen_n(\text{x}) \neq \emptyset$ for some basic block

$DepGen_0(\text{x}) \neq \emptyset$
$DepGen_1(\text{x}) \neq \emptyset$
$DepGen_2(\text{x}) = \emptyset$
$DepGen_3(\text{x}) = \emptyset$
$DepGen_4(\text{x}) = \emptyset$
$DepGen_5(\text{x}) \neq \emptyset$



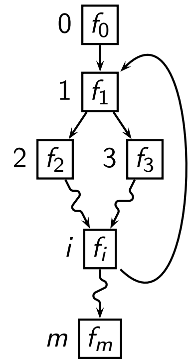
Control flow graph

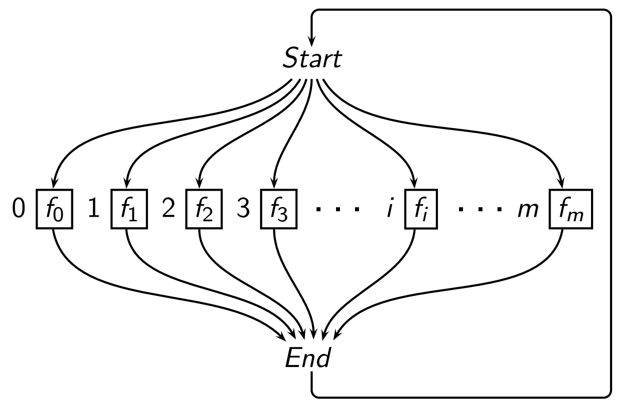


Flow insensitive analysis

Flow Insensitivity in Data Flow Analysis

An alternative model if $DepGen_n(\text{x}) \neq \emptyset$

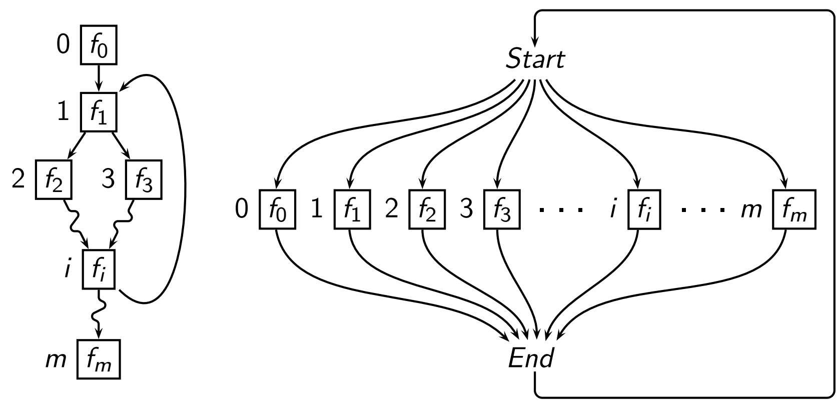




Allows arbitrary compositions of flow functions in any order ⇒ Flow insensitivity

Flow Insensitivity in Data Flow Analysis

An alternative model if $DepGen_n(x) \neq \emptyset$

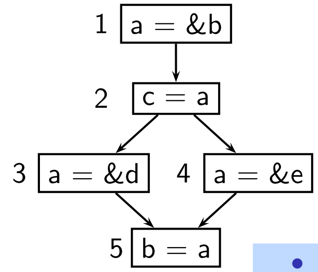


In practice, dependent constraints are collected in a global repository in one pass and then are solved independently

Example of Flow Insensitive Analysis

Flow insensitive points-to analysis
⇒ Same points-to information at each program point

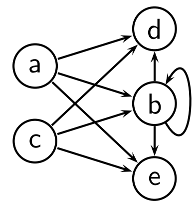
Program



Constraints

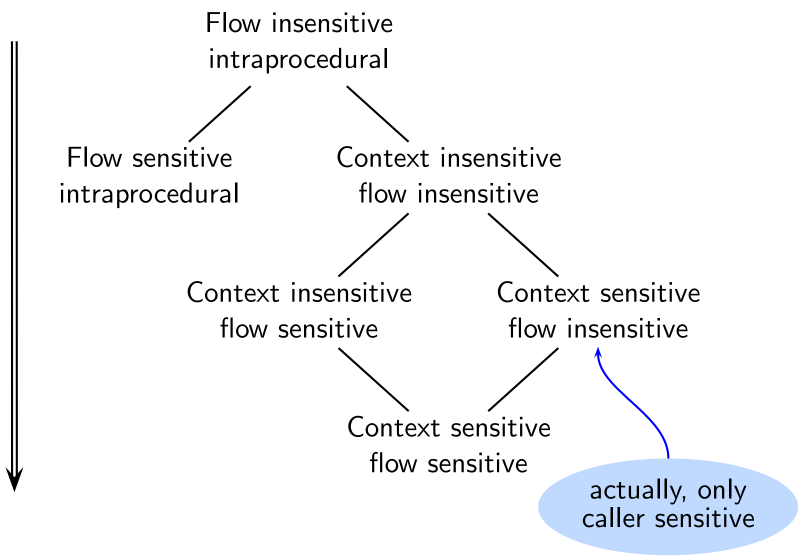
Node	Constraint
1	$P_a \supseteq \{b\}$
2	$P_c \supseteq P_a$
3	$P_a \supseteq \{d\}$
4	$P_a \supseteq \{e\}$
5	$P_b \supseteq P_a$

Points-to Graph



- c does not point to any location in block 1
- c does not point b in block 5
- b does not point to itself at any time

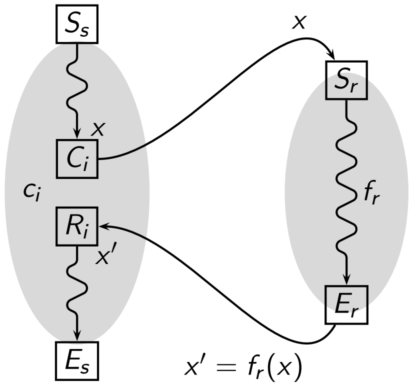
Increasing Precision in Data Flow Analysis



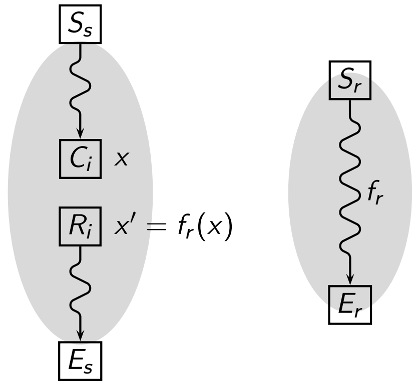
Part 4

Classical Functional Approach

Functional Approach



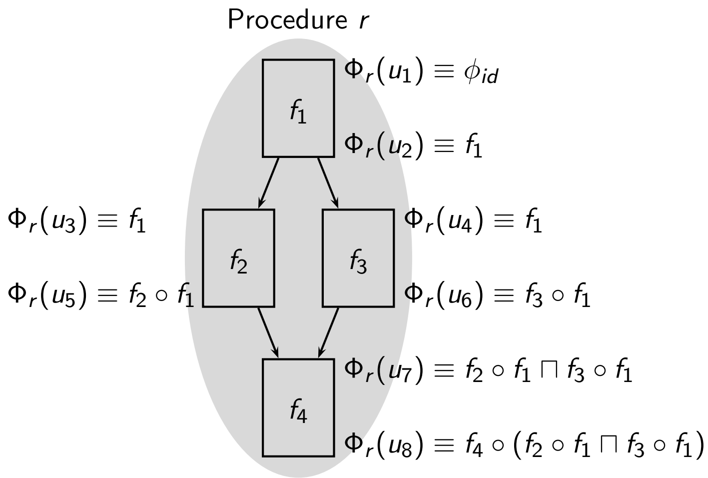
Functional Approach



- Compute summary flow functions for each procedure
- Use summary flow functions as the flow function for a call block

Notation for Summary Flow Function

For simplicity forward flow is assumed.



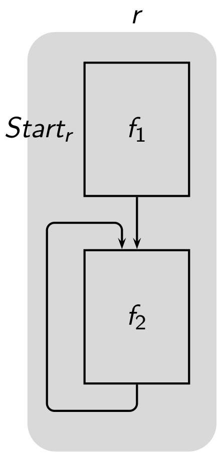
Constructing Summary Flow Function

For simplicity forward flow is assumed.

$$\Phi_r(Entry(n)) = \begin{cases} \phi_{id} & \text{if } n \text{ is } Start_r \\ \displaystyle\bigsqcap_{p \in pred(n)} \Big(\Phi_r(Exit(p))\Big) & \text{otherwise} \end{cases}$$

$$\Phi_r(Exit(n)) = \begin{cases} \Phi_s(u) \circ \Phi_r(Entry(n)) & \text{if } n \text{ calls procedure } s \text{ and } u \text{ is } Exit(End_s) \\ f_n \circ \Phi_r(Entry(n)) & \text{otherwise} \end{cases}$$

Constructing Summary Flow Functions



Iteration #1

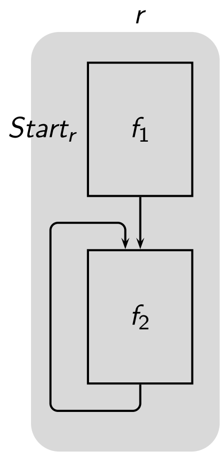
$\Phi_r(u_1) = \phi_{id}$

$\Phi_r(u_2) = f_1$

$\Phi_r(u_3) = f_1$

$\Phi_r(u_4) = f_2 \circ f_1$

Constructing Summary Flow Functions



Iteration #2

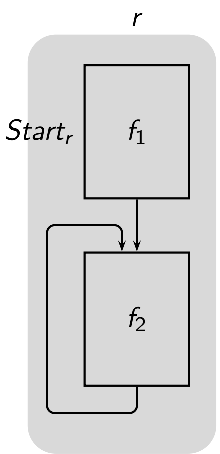
$\Phi_r(u_1) = \phi_{id}$

$\Phi_r(u_2) = f_1$

$\Phi_r(u_3) = f_1 \sqcap f_2 \circ f_1$

$\Phi_r(u_4) = f_2 \circ (f_1 \sqcap f_2 \circ f_1)$

Constructing Summary Flow Functions



Iteration #3

$\Phi_r(u_1) = \phi_{id}$

$\Phi_r(u_2) = f_1$

$\Phi_r(u_3) = f_1 \sqcap f_2 \circ f_1 \sqcap f_2 \circ (f_1 \sqcap f_2 \circ f_1)$

$\Phi_r(u_4) = f_2 \circ (f_1 \sqcap f_2 \circ f_1 \sqcap f_2 \circ (f_1 \sqcap f_2 \circ f_1))$

Termination is possible only if all function compositions and confluences can be reduced to a finite set of functions

Lattice of Flow Functions for Live Variables Analysis

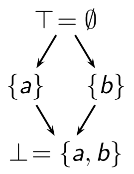
Component functions (i.e. for a single variable)

<table>
<tr><th>Lattice of data flow values</th><th>All possible flow functions</th><th>Lattice of flow functions</th></tr>
<tr>
<td>$\widehat{\top} = \emptyset$
$\downarrow$
$\widehat{\bot} = \{a\}$</td>
<td>
<table>
<tr><th>Gen_n</th><th>$Kill_n$</th><th>$\widehat{f}_n$</th></tr>
<tr><td>$\emptyset$</td><td>$\emptyset$</td><td>$\widehat{\phi}_{id}$</td></tr>
<tr><td>$\emptyset$</td><td>$\{a\}$</td><td>$\widehat{\phi}_{\top}$</td></tr>
<tr><td>$\{a\}$</td><td>$\emptyset$</td><td>$\widehat{\phi}_{\bot}$</td></tr>
</table>
</td>
<td>$\widehat{\phi}_{\top}$
$\downarrow$
$\widehat{\phi}_{id}$
$\downarrow$
$\widehat{\phi}_{\bot}$</td>
</tr>
</table>

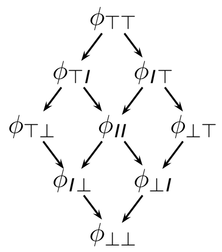
Lattice of Flow Functions for Live Variables Analysis

Flow functions for two variables

Lattice of data flow values	All possible flow functions	Lattice of flow functions



Gen_n	Kill_n	f_n	Gen_n	Kill_n	f_n
$\emptyset$	$\emptyset$	ϕ_{II}	$\{b\}$	$\emptyset$	$\phi_{I\bot}$
$\emptyset$	$\{a\}$	$\phi_{\top I}$	$\{b\}$	$\{a\}$	$\phi_{\top\bot}$
$\emptyset$	$\{b\}$	$\phi_{I\top}$	$\{b\}$	$\{b\}$	$\phi_{I\bot}$
$\emptyset$	$\{a,b\}$	$\phi_{\top\top}$	$\{b\}$	$\{a,b\}$	$\phi_{\top\bot}$
$\{a\}$	$\emptyset$	$\phi_{\bot I}$	$\{a,b\}$	$\emptyset$	$\phi_{\bot\bot}$
$\{a\}$	$\{a\}$	$\phi_{\bot I}$	$\{a,b\}$	$\{a\}$	$\phi_{\bot\bot}$
$\{a\}$	$\{b\}$	$\phi_{\bot\top}$	$\{a,b\}$	$\{b\}$	$\phi_{\bot\bot}$
$\{a\}$	$\{a,b\}$	$\phi_{\bot\top}$	$\{a,b\}$	$\{a,b\}$	$\phi_{\bot\bot}$



Reducing Function Compositions

Assumption: No dependent parts (as in bit vector frameworks).
Kill_n is $ConstKill_n$ and Gen_n is $ConstGen_n$.

$$
\begin{aligned}
f_3(x) &= f_2(f_1(x)) \\
&= f_2((x - \text{Kill}_1) \cup \text{Gen}_1) \\
&= \Big(((x - \text{Kill}_1) \cup \text{Gen}_1) - \text{Kill}_2 \Big) \cup \text{Gen}_2 \\
&= (x - (\text{Kill}_1 \cup \text{Kill}_2)) \cup (\text{Gen}_1 - \text{Kill}_2) \cup \text{Gen}_2
\end{aligned}
$$

Hence,

$$
\begin{aligned}
\text{Kill}_3 &= \text{Kill}_1 \cup \text{Kill}_2 \\
\text{Gen}_3 &= (\text{Gen}_1 - \text{Kill}_2) \cup \text{Gen}_2
\end{aligned}
$$

Reducing Function Confluences

Assumption: No dependent parts (as in bit vector frameworks).
Kill_n is $ConstKill_n$ and Gen_n is $ConstGen_n$.

- When $\sqcap$ is $\cup$,

$$
\begin{aligned}
f_3(x) &= f_2(x) \cup f_1(x) \\
&= ((x - \text{Kill}_2) \cup \text{Gen}_2) \cup ((x - \text{Kill}_1) \cup \text{Gen}_1) \\
&= (x - (\text{Kill}_1 \cap \text{Kill}_2)) \cup (\text{Gen}_1 \cup \text{Gen}_2)
\end{aligned}
$$

Hence,

$$
\begin{aligned}
\text{Kill}_3 &= \text{Kill}_1 \cap \text{Kill}_2 \\
\text{Gen}_3 &= \text{Gen}_1 \cup \text{Gen}_2
\end{aligned}
$$

Reducing Function Confluences

Assumption: No dependent parts (as in bit vector frameworks).
Kill_n is $ConstKill_n$ and Gen_n is $ConstGen_n$.

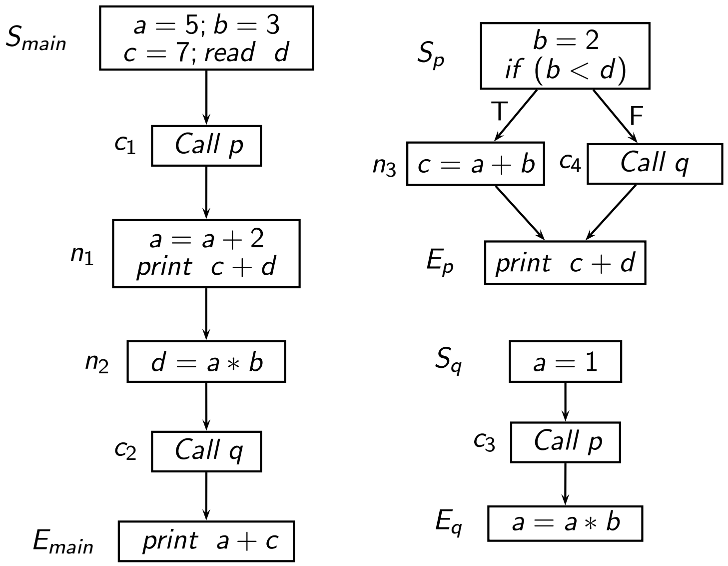
- When $\sqcap$ is $\cap$,

$$
\begin{aligned}
f_3(x) &= f_2(x) \cap f_1(x) \\
&= ((x - \text{Kill}_2) \cup \text{Gen}_2) \cap ((x - \text{Kill}_1) \cup \text{Gen}_1) \\
&= (x - (\text{Kill}_1 \cup \text{Kill}_2)) \cup (\text{Gen}_1 \cap \text{Gen}_2)
\end{aligned}
$$

Hence

$$
\begin{aligned}
\text{Kill}_3 &= \text{Kill}_1 \cup \text{Kill}_2 \\
\text{Gen}_3 &= \text{Gen}_1 \cap \text{Gen}_2
\end{aligned}
$$

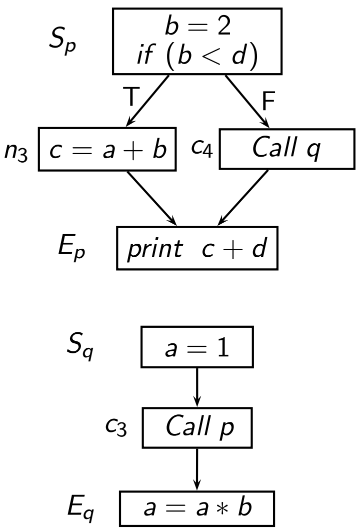
An Example of Interprocedural Liveness Analysis



Summary Flow Functions for Interprocedural Liveness Analysis

Proc.	Flow Function	Defining Expression	Iteration #1		Changes in iteration #2	
			Gen	Kill	Gen	Kill
p	$\Phi_p(E_p)$	f_{E_p}	$\{c, d\}$	$\emptyset$		
	$\Phi_p(n_3)$	$f_{n_3} \circ \Phi_p(E_p)$	$\{a, b, d\}$	$\{c\}$		
	$\Phi_p(c_4)$	$f_q \circ \Phi_p(E_p) = \phi_\top$	$\emptyset$	$\{a, b, c, d\}$	$\{d\}$	$\{a, b, c\}$
	$\Phi_p(S_p)$	$f_{S_p} \circ (\Phi_p(n_3) \sqcap \Phi_p(c_4))$	$\{a, d\}$	$\{b, c\}$		
	f_p	$\Phi_p(S_p)$	$\{a, d\}$	$\{b, c\}$		
q	$\Phi_q(E_q)$	f_{E_q}	$\{a, b\}$	$\{a\}$		
	$\Phi_q(c_3)$	$f_p \circ \Phi_q(E_q)$	$\{a, d\}$	$\{a, b, c\}$		
	$\Phi_q(S_q)$	$f_{S_q} \circ \Phi_q(c_3)$	$\{d\}$	$\{a, b, c\}$		
	f_q	$\Phi_q(S_q)$	$\{d\}$	$\{a, b, c\}$		

Computed Summary Flow Function



Summary Flow Function	
$\Phi_p(E_p)$	$BI_p \cup \{c, d\}$
$\Phi_p(n_3)$	$(BI_p - \{c\}) \cup \{a, b, d\}$
$\Phi_p(c_4)$	$(BI_p - \{a, b, c\}) \cup \{d\}$
$\Phi_p(S_p)$	$(BI_p - \{b, c\}) \cup \{a, d\}$
$\Phi_q(E_q)$	$(BI_q - \{a\}) \cup \{a, b\}$
$\Phi_q(c_3)$	$(BI_q - \{a, b, c\}) \cup \{a, d\}$
$\Phi_q(S_q)$	$(BI_q - \{a, b, c\}) \cup \{d\}$

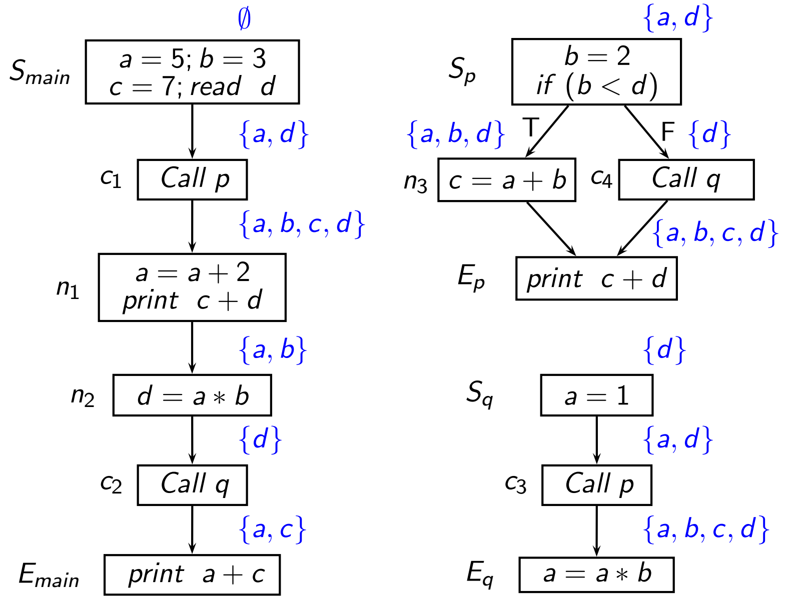
Result of Interprocedural Liveness Analysis

Data flow variable	Summary flow function		Data flow value
	Name	Definition	
Procedure *main*, $BI = \emptyset$			
In_{E_m}	$\Phi_m(E_m)$	$BI_m \cup \{a, c\}$	$\{a, c\}$
In_{c_2}	$\Phi_m(c_2)$	$(BI_m - \{a, b, c\}) \cup \{d\}$	$\{d\}$
In_{n_2}	$\Phi_m(n_2)$	$(BI_m - \{a, b, c, d\}) \cup \{a, b\}$	$\{a, b\}$
In_{n_1}	$\Phi_m(n_1)$	$(BI_m - \{a, b, c, d\}) \cup \{a, b, c, d\}$	$\{a, b, c, d\}$
In_{c_1}	$\Phi_m(c_1)$	$(BI_m - \{a, b, c, d\}) \cup \{a, d\}$	$\{a, d\}$
In_{S_m}	$\Phi_m(S_m)$	$BI_m - \{a, b, c, d\}$	$\emptyset$

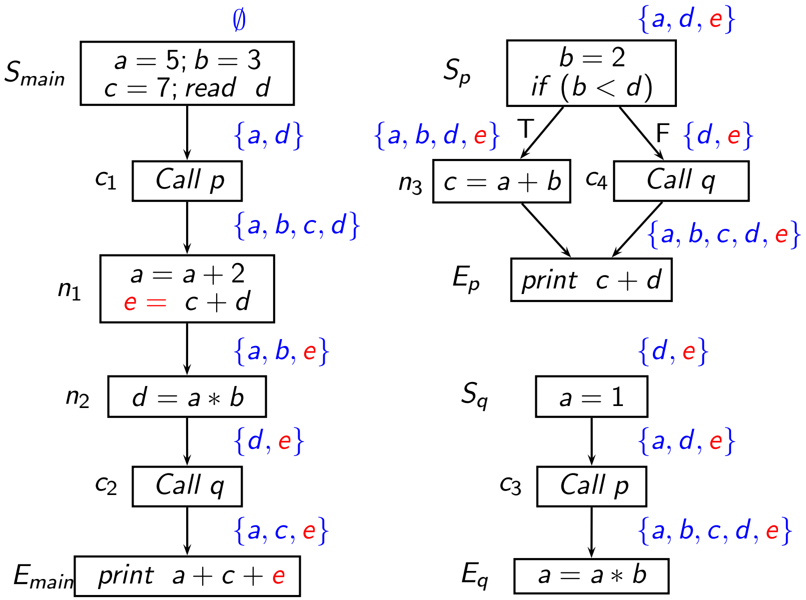
Result of Interprocedural Liveness Analysis

Data flow variable	Summary flow function: Name	Summary flow function: Definition	Data flow value
Procedure p, $BI = \{a,b,c,d\}$			
In_{E_p}	$\Phi_p(E_p)$	$BI_p \cup \{c,d\}$	$\{a,b,c,d\}$
In_{n_3}	$\Phi_p(n_3)$	$(BI_p - \{c\}) \cup \{a,b,d\}$	$\{a,b,d\}$
In_{c_4}	$\Phi_p(c_4)$	$(BI_p - \{a,b,c\}) \cup \{d\}$	$\{d\}$
In_{S_p}	$\Phi_p(S_p)$	$(BI_p - \{b,c\}) \cup \{a,d\}$	$\{a,d\}$
Procedure q, $BI = \{a,b,c,d\}$			
In_{E_q}	$\Phi_q(E_q)$	$(BI_q - \{a\}) \cup \{a,b\}$	$\{a,b,c,d\}$
In_{c_3}	$\Phi_q(c_3)$	$(BI_q - \{a,b,c\}) \cup \{a,d\}$	$\{a,d\}$
In_{S_q}	$\Phi_q(S_q)$	$(BI_q - \{a,b,c\}) \cup \{d\}$	$\{d\}$

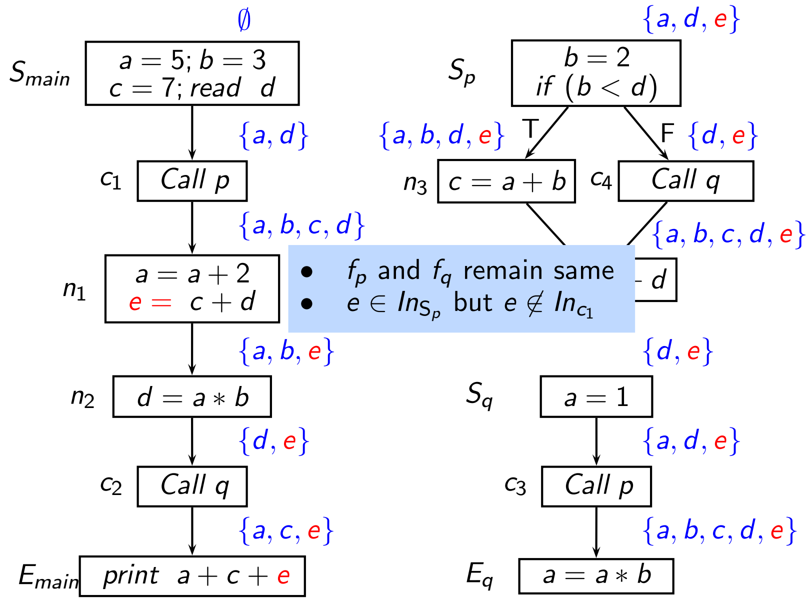
Result of Interprocedural Liveness Analysis



Context Sensitivity of Interprocedural Liveness Analysis



Context Sensitivity of Interprocedural Liveness Analysis



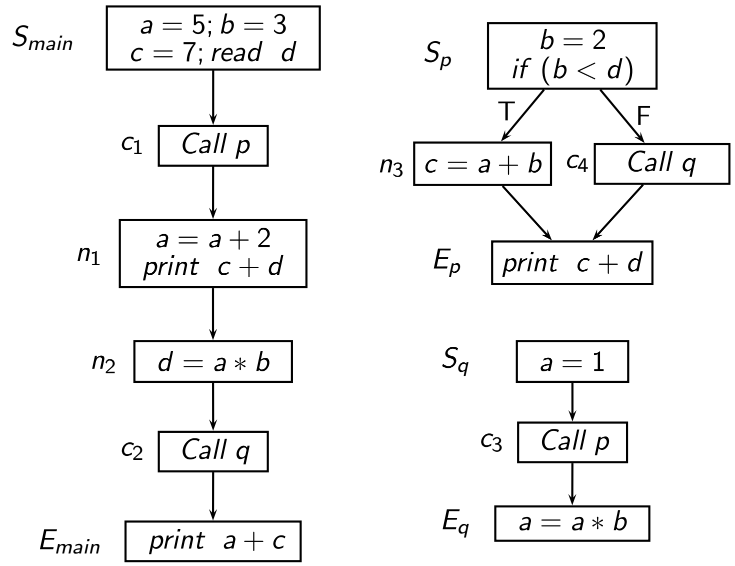
- f_p and f_q remain same
- $e \in In_{S_p}$ but $e \notin In_{c_1}$

Limitations of Functional Approach to Interprocedural Data Flow Analysis

- Problems with constructing summary flow functions
  - Reducing expressions defining flow functions may not be possible when $DepGen_n \neq \emptyset$
  - May work for some instances of some problems but not for all
- Enumeration based approach
  - Instead of constructing flow functions, remember the mapping $x \mapsto y$ as input output values
  - Reuse output value of a flow function when the same input value is encountered again

  Requires the number of values to be finite

Functional Approach for Constant Propagation Example



Summary Flow Functions for Interprocedural Constant Propagation

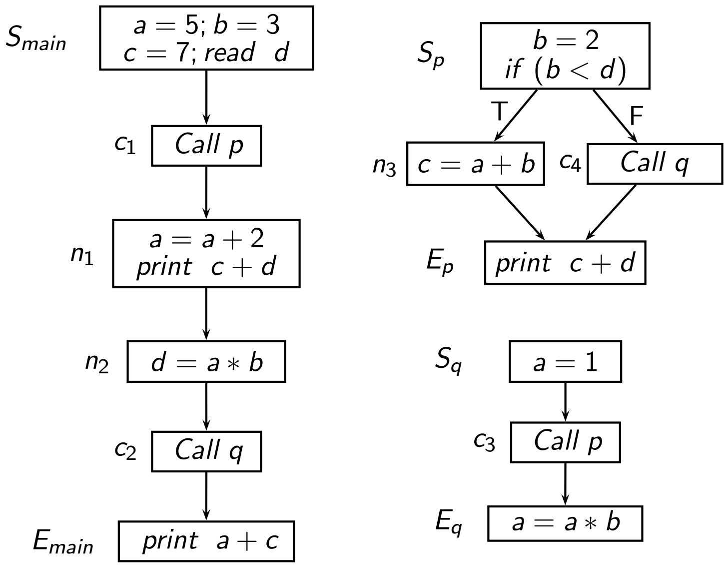
Flow Function	Iteration #1	Changes in iteration #2	Changes in iteration #3	Changes in iteration #4
$\Phi_p(E_p)$	$\langle v_a, 2, v_c, v_d \rangle$			
$\Phi_p(n_3)$	$\langle v_a, 2, v_a + 2, v_d \rangle$			
$\Phi_p(c_4)$	$\langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle$	$\langle 2, 2, 3, v_d \rangle$	$\langle \widehat{\bot}, 2, 3, v_d \rangle$	$\langle \widehat{\bot}, 2, \widehat{\bot}, v_d \rangle$
$\Phi_p(S_p)$	$\langle v_a, 2, v_a + 2, v_d \rangle$	$\langle v_a \sqcap 2, 2, (v_a + 2) \sqcap 3, v_d \rangle$	$\langle \widehat{\bot}, 2, \widehat{\bot}, v_d \rangle$	
$\Phi_q(E_q)$	$\langle 1, v_b, v_c, v_d \rangle$			
$\Phi_q(c_3)$	$\langle 1, 2, 3, v_d \rangle$	$\langle \widehat{\bot}, 2, 3, v_d \rangle$	$\langle \widehat{\bot}, 2, \widehat{\bot}, v_d \rangle$	
$\Phi_q(S_q)$	$\langle 2, 2, 3, v_d \rangle$	$\langle \widehat{\bot}, 2, 3, v_d \rangle$	$\langle \widehat{\bot}, 2, \widehat{\bot}, v_d \rangle$	

Interprocedural Constant Propagation Using the Functional Approach

Block	Out_n
S_m	$\langle 5, 3, 7, \widehat{\bot} \rangle$
c_1	$\langle \widehat{\bot}, 2, \widehat{\bot}, \widehat{\bot} \rangle$
n_1	$\langle \widehat{\bot}, 2, \widehat{\bot}, \widehat{\bot} \rangle$
n_2	$\langle \widehat{\bot}, 2, \widehat{\bot}, \widehat{\bot} \rangle$
c_2	$\langle \widehat{\bot}, 2, \widehat{\bot}, \widehat{\bot} \rangle$
E_m	$\langle \widehat{\bot}, 2, \widehat{\bot}, \widehat{\bot} \rangle$

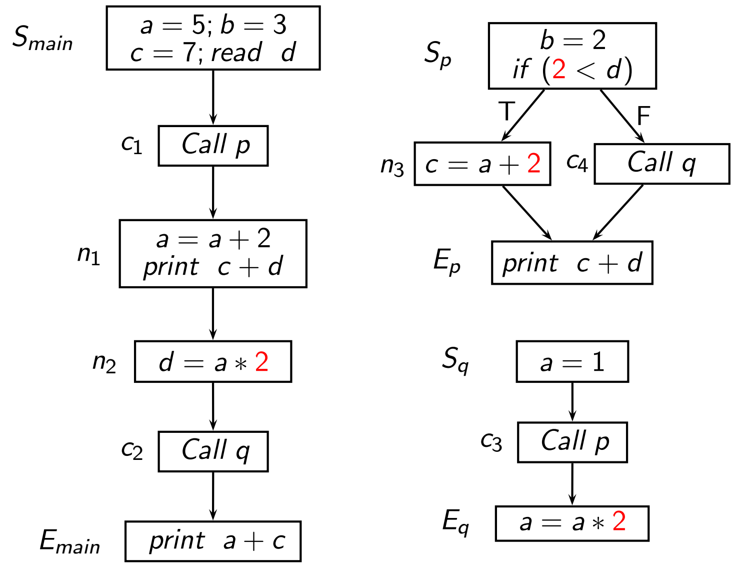
Block	Out_n
S_p	$\langle \widehat{\bot}, 2, \widehat{\bot}, \widehat{\bot} \rangle$
n_3	$\langle \widehat{\bot}, 2, \widehat{\bot}, \widehat{\bot} \rangle$
c_4	$\langle \widehat{\bot}, 2, \widehat{\bot}, \widehat{\bot} \rangle$
E_p	$\langle \widehat{\bot}, 2, \widehat{\bot}, \widehat{\bot} \rangle$
S_q	$\langle \widehat{\bot}, 2, \widehat{\bot}, \widehat{\bot} \rangle$
c_3	$\langle \widehat{\bot}, 2, \widehat{\bot}, \widehat{\bot} \rangle$
E_2	$\langle \widehat{\bot}, 2, \widehat{\bot}, \widehat{\bot} \rangle$

Constant Propagation Using Functional Approach



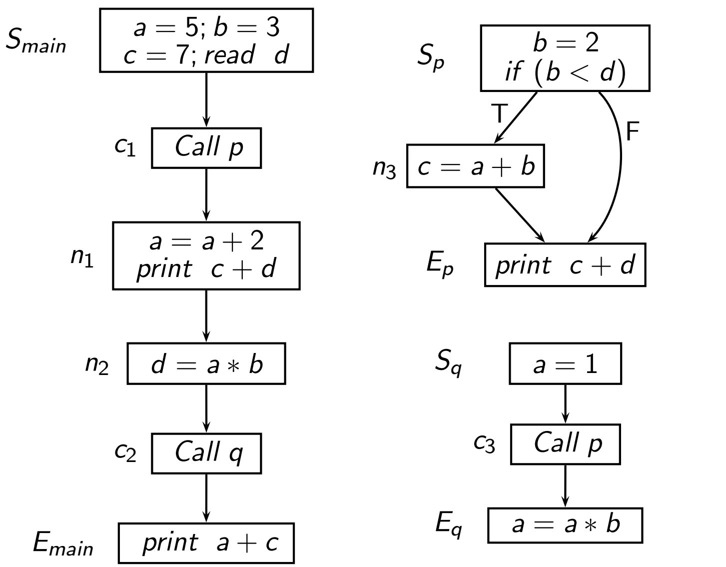
CS 618 Interprocedural DFA: Classical Functional Approach 41/86

Constant Propagation Using Functional Approach



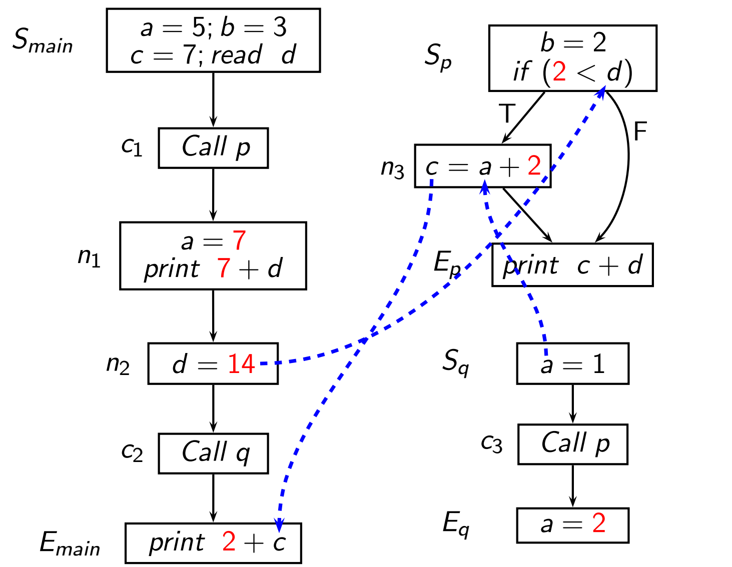
Oct 2009 IIT Bombay

Tutorial Problem for Functional Interprocedural Analysis



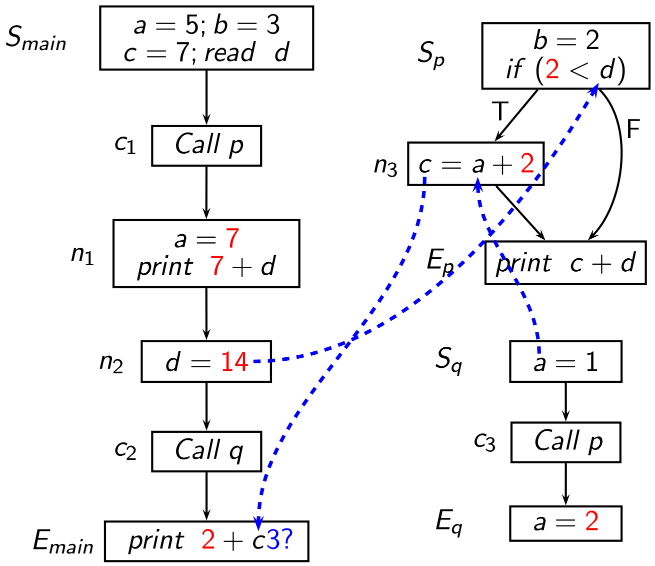
CS 618 Interprocedural DFA: Classical Functional Approach 42/86

Tutorial Problem for Functional Interprocedural Analysis



Oct 2009 IIT Bombay

Tutorial Problem for Functional Interprocedural Analysis



Part 5

Classical Call Strings Approach

Classical Full Call Strings Approach

Most general, flow and context sensitive method

- Remember call history
  Information should be propagated *back* to the correct point
- Call string at a program point:
  - Sequence of *unfinished calls* reaching that point
  - Starting from the S_{main}

  A snap-shot of call stack in terms of call sites

Interprocedural Data Flow Analysis Using Call Strings

- Tagged data flow information
  - IN_n and OUT_n are sets of the form $\{\langle\sigma, x\rangle \mid \sigma \text{ is a call string }, x \in L\}$
  - The final data flow information is

$$In_n = \bigsqcap_{\langle\sigma,x\rangle\in \text{IN}_n} x$$

$$Out_n = \bigsqcap_{\langle\sigma,x\rangle\in \text{OUT}_n} x$$

- Flow functions to manipulate tagged data flow information
  - Intraprocedural edges manipulate data flow value x
  - Interprocedural edges manipulate call string σ

Overall Data Flow Equations

$$\mathrm{IN}_n = \begin{cases} \langle \lambda, BI \rangle & n \text{ is a } S_{main} \\ \biguplus_{p \in pred(n)} \mathrm{OUT}_p & \text{otherwise} \end{cases}$$

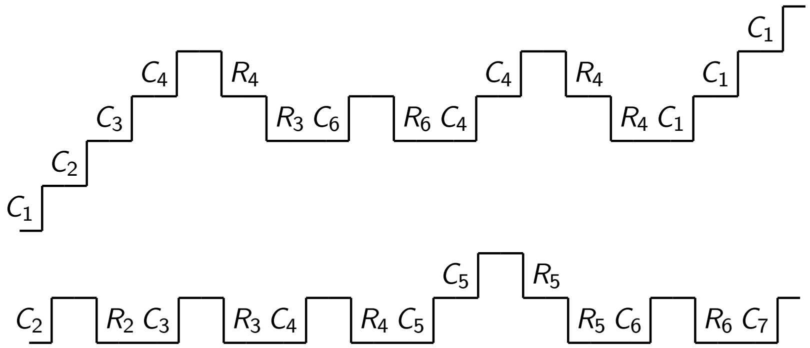
$$\mathrm{OUT}_n = DepGEN_n$$

Effectively, $ConstGEN_n = ConstKILL_n = \emptyset$ and $DepKILL_n(X) = X$.

$$\begin{aligned} X \uplus Y = & \left\{ \langle \sigma, \mathrm{x} \sqcap \mathrm{y} \rangle \mid \langle \sigma, \mathrm{x} \rangle \in X,\ \langle \sigma, \mathrm{y} \rangle \in Y \right\} \cup \\ & \left\{ \langle \sigma, \mathrm{x} \rangle \mid \langle \sigma, \mathrm{x} \rangle \in X,\ \forall \mathrm{z} \in L, \langle \sigma, \mathrm{z} \rangle \notin Y \right\} \cup \\ & \left\{ \langle \sigma, \mathrm{y} \rangle \mid \langle \sigma, \mathrm{y} \rangle \in Y,\ \forall \mathrm{z} \in L, \langle \sigma, \mathrm{z} \rangle \notin X \right\} \end{aligned}$$

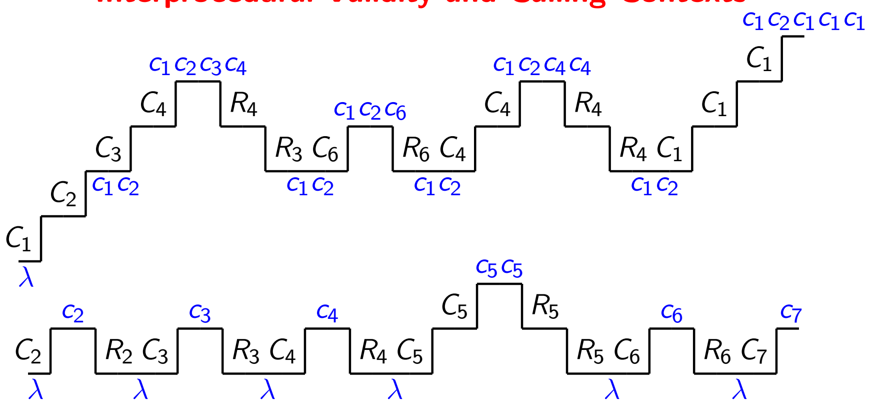
(We merge underlying data flow values only if the contexts are same.)

Interprocedural Validity and Calling Contexts



- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step.

Interprocedural Validity and Calling Contexts



- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step.
- Calling context is represented by the remaining descending steps.

Manipulating Values

- Call edge $C_i \rightarrow S_p$ (i.e. call site c_i calling procedure p).
  - Append c_i to every σ.
  - Propagate the data flow values unchanged.

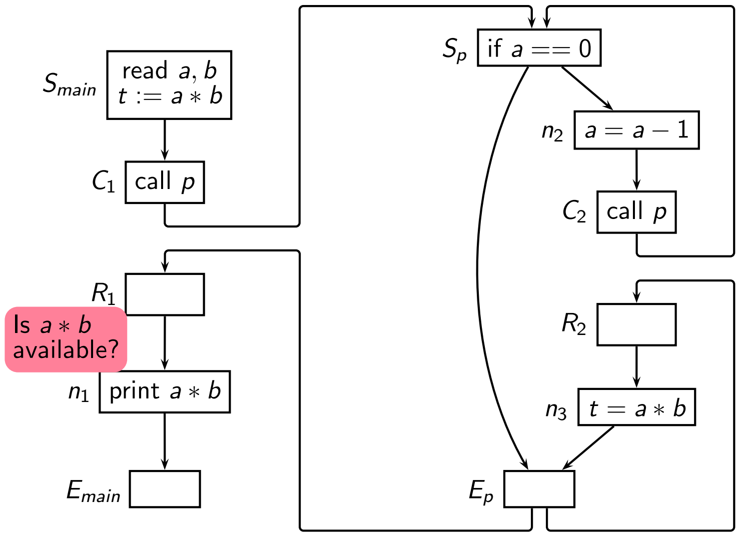
  Ascend

- Return edge $E_p \rightarrow R_i$ (i.e. p returning the control to call site c_i).
  - If the last call site is c_i, remove it and propagate the data flow value unchanged.
  - Block other data flow values.

  Descend

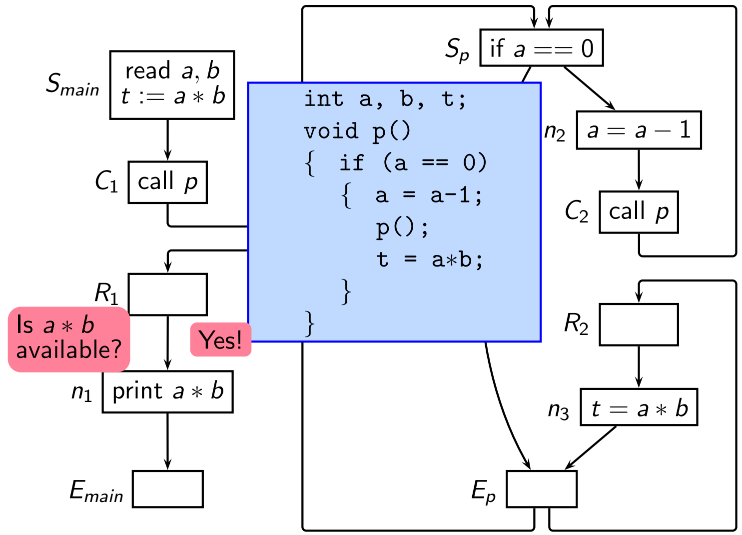
$$DepGEN_n(X) = \begin{cases} \left\{ \langle \sigma \cdot c_i, \mathrm{x} \rangle \mid \langle \sigma, \mathrm{x} \rangle \in X \right\} & n \text{ is } C_i \\ \left\{ \langle \sigma, \mathrm{x} \rangle \mid \langle \sigma \cdot c_i, \mathrm{x} \rangle \in X \right\} & n \text{ is } R_i \\ \left\{ \langle \sigma, f_n(\mathrm{x}) \rangle \mid \langle \sigma, \mathrm{x} \rangle \in X \right\} & \text{otherwise} \end{cases}$$

Available Expressions Analysis Using Call Strings Approach

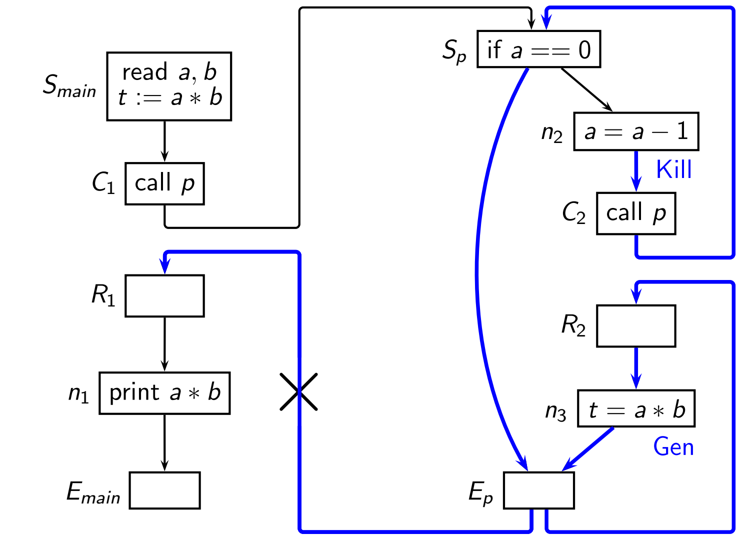


CS 618 Interprocedural DFA: Classical Call Strings Approach 48/86

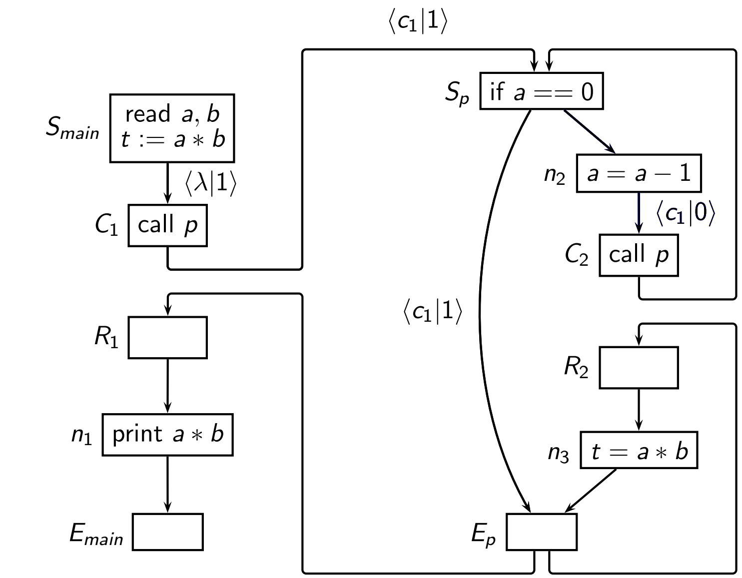
Available Expressions Analysis Using Call Strings Approach



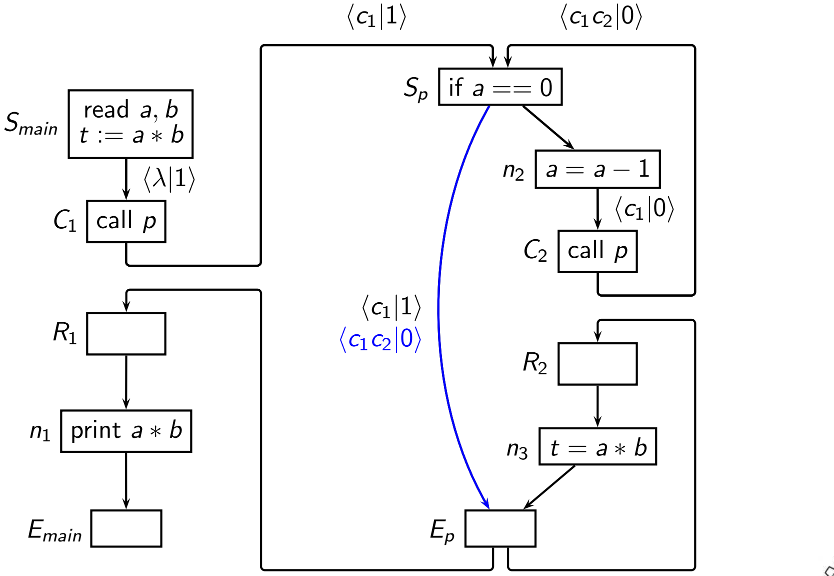
Available Expressions Analysis Using Call Strings Approach



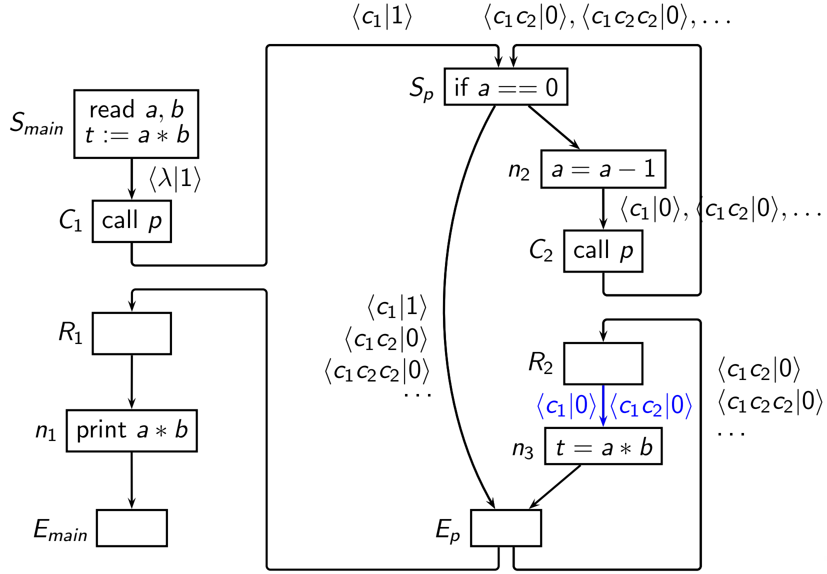
Available Expressions Analysis Using Call Strings Approach



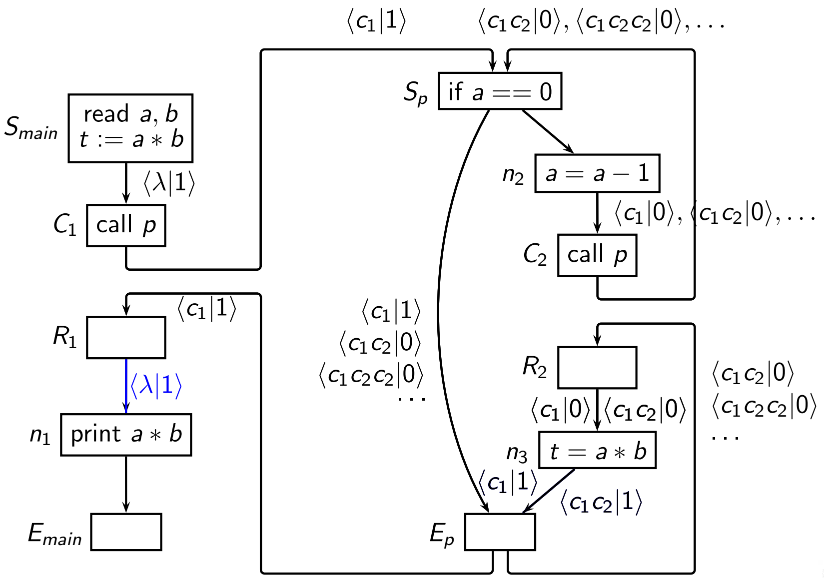
Available Expressions Analysis Using Call Strings Approach



Available Expressions Analysis Using Call Strings Approach



Available Expressions Analysis Using Call Strings Approach



Tutorial Problem

Generate a trace of the preceding example in the following format:

Step No.	Selected Node	Qualified Data Flow Value		Remaining Work List
		IN_n	OUT_n	

- Assume that call site c_i appended to a call string σ only if there are at most 2 occurences of c_i in σ
- What about work list organization?

The Need for Multiple Occurrences of a Call Site

even if data flow values in cyclic call sequence do not change

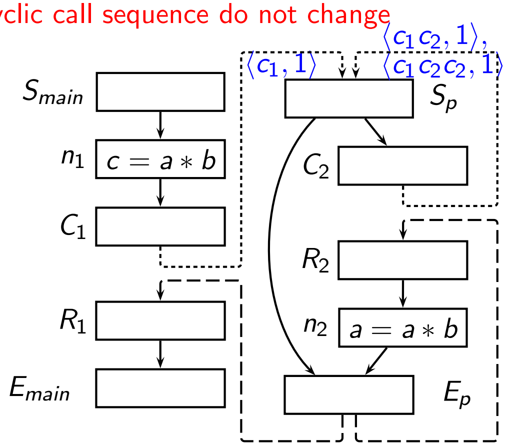
```
1. int a,b,c;
2. void main()
3. {   c = a*b;
4.     p();
5. }
6. void p()
7. {  if (...)
8.    {  p();
9.       Is a*b available?
10.        a = a*b;
11.   }
12. }
```

Path 1	Path 2
3 : Gen	3 : Gen
4	4
7	7
8	8
7	7
12	8
9	7
10 : Kill	12
11	9
12	10 : Kill
5	11
	12
	9
	10 : Kill

The Need for Multiple Occurrences of a Call Site

even if data flow values in cyclic call sequence do not change

```
1. int a,b,c;
2. void main()
3. {   c = a*b;
4.     p();
5. }
6. void p()
7. {  if (...)
8.    {  p();
9.       Is a*b available?
10.        a = a*b;
11.   }
12. }
```



- Interprocedurally valid IFP

  $S_m, n_1, C_1, S_p, C_2, S_p, C_2, S_p, E_p, R_2, \overset{\text{Kill}}{n_2}, E_p, R_2, n_2$

The Need for Multiple Occurrences of a Call Site

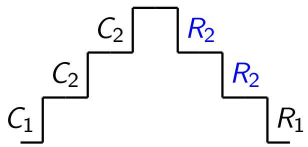
even if data flow values in cyclic call sequence do not change

In terms of staircase diagram

- Interprocedurally valid IFP

  $S_m, n_1, C_1, S_p, C_2, S_p, C_2, S_p, E_p, R_2, \overset{\text{Kill}}{n_2}, E_p, R_2, n_2$

- You cannot descend twice, unless you ascend twice



- Even if the data flow values do not change while ascending, you need to ascend because they may change while descending

Terminating Call String Construction

- For non-recursive programs: Number of call strings is finite
- For recursive programs: Number of call strings could be infinite
  Fortunately, the problem is decidable for finite lattices.
  - All call strings upto the following length *must be* constructed
    - $K \cdot (|L| + 1)^2$ for general bounded frameworks
      (L is the overall lattice of data flow values)
    - $K \cdot (|\widehat{L}| + 1)^2$ for separable bounded frameworks
      ($\widehat{L}$ is the component lattice for an entity)
    - $K \cdot 3$ for bit vector frameworks
    - 3 occurrences of any call site in a call string for bit vector frameworks

  $\Rightarrow$ Not a bound but prescribed necessary length

  $\Rightarrow$ Large number of long call strings

Classical Call String Length

- Notation
  - $IVP(n, m)$: Interprocedurally valid path from block n to block m
  - $CS(\rho)$: Number of call nodes in ρ that do not have the matching return node in ρ
    (length of the call string representing $IVP(n, m)$)
- Claim
  Let $M = K \cdot (|L| + 1)^2$ where K is the number of distinct call sites in any call chain
  Then, for any $\rho = IVP(\mathsf{S}_{main}, m)$ such that
  $CS(\rho) > M$,
  $\exists\ \rho' = IVP(\mathsf{S}_{main}, m)$ such that
  $CS(\rho') \le M$, and $f_\rho(BI) = f_{\rho'}(BI)$.

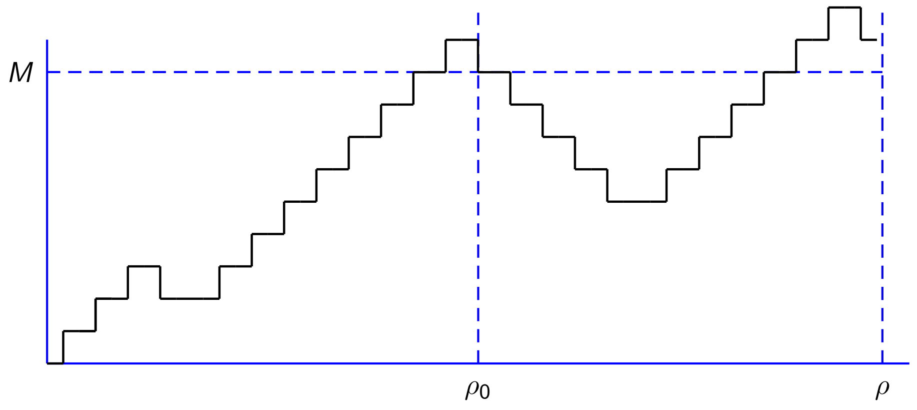
$\Rightarrow$ ρ, the longer path, is redundant for data flow analysis

Classical Call String Length

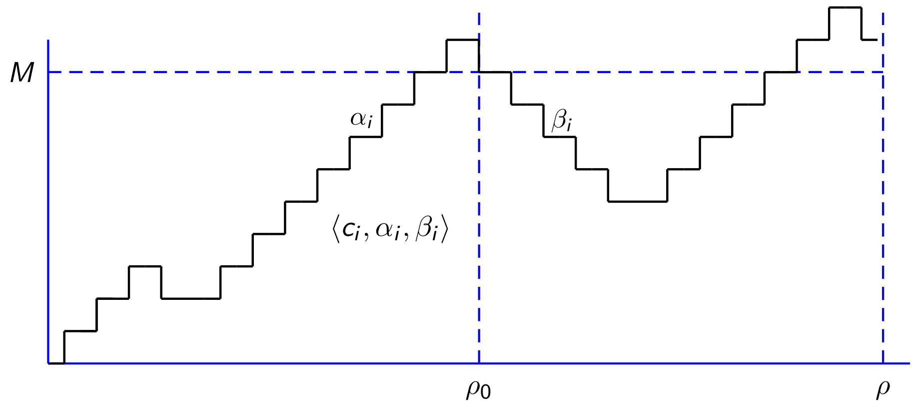
Sharir-Pnueli [1981]

- Consider the smallest prefix ρ_0 of ρ such that $CS(\rho_0) > M$
- Consider a triple $\langle c_i, \alpha_i, \beta_i \rangle$ where
  - α_i is the data flow value reaching call node C_i along ρ and
  - β_i is the data flow value reaching the corresponding return node R_i along ρ
    If R_i is not in ρ, then $\beta_i = \Omega$ (undefined)

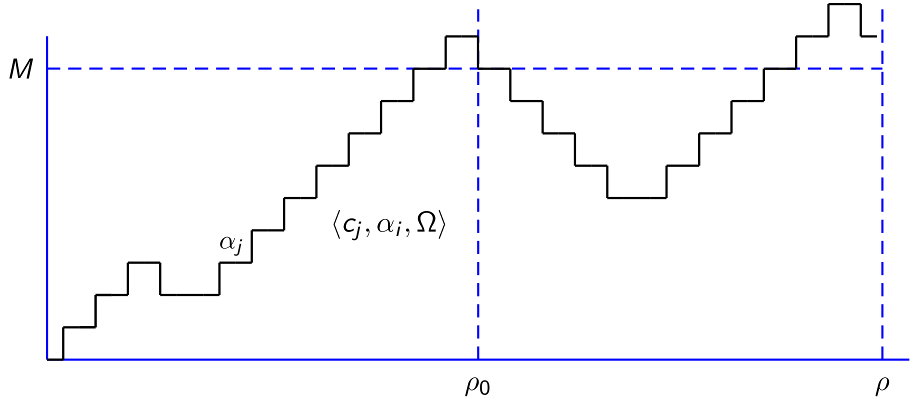
Classical Call String Length



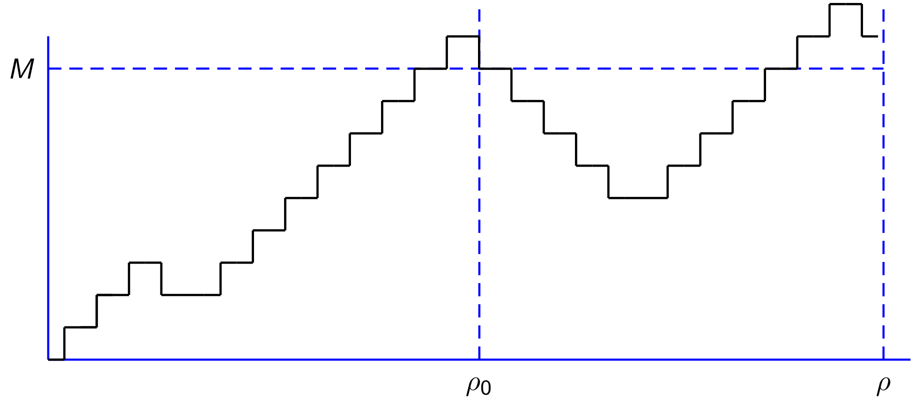
Classical Call String Length



Classical Call String Length



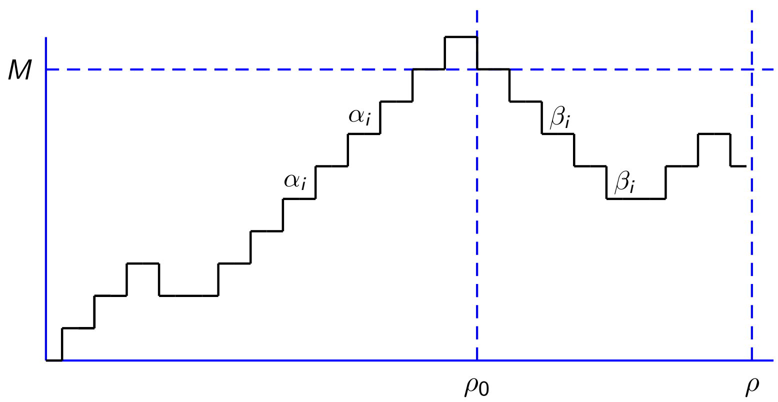
Classical Call String Length



- Number of distinct triples $\langle c_i, \alpha_i, \beta_i \rangle$ is $M = K \cdot (|L| + 1)^2$.
- There are at least two calls from the same call site that have the same effect on data flow values

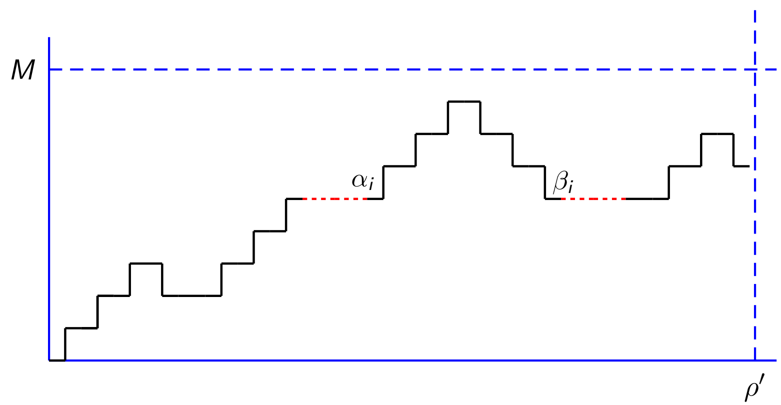
Classical Call String Length

When β_i is not Ω



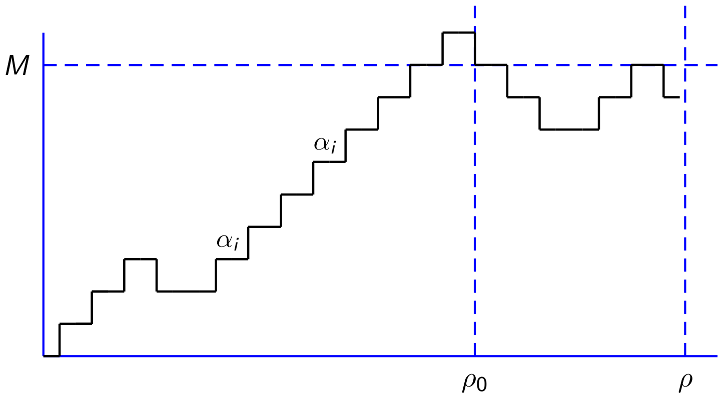
Classical Call String Length

When β_i is not Ω



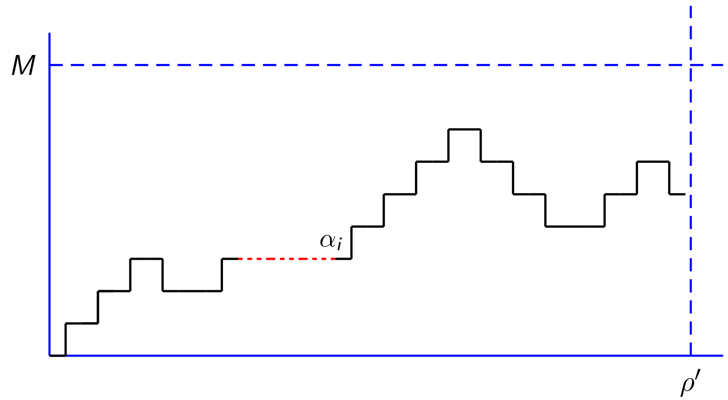
Classical Call String Length

When β_i is Ω



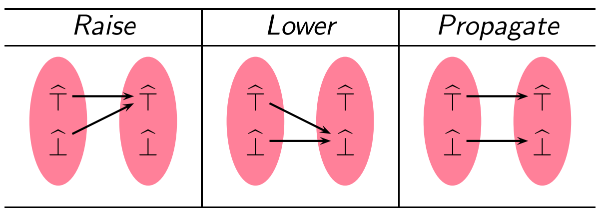
Classical Call String Length

When β_i is Ω



Tighter Bound for Bit Vector Frameworks

- $\widehat{L}$ is $\{0, 1\}$, L is $\{0, 1\}^m$
- $\widehat{\sqcap}$ is either boolean AND or boolean OR
- $\widehat{\top}$ and $\widehat{\bot}$ are 0 or 1 depending on $\widehat{\sqcap}$.
- $\widehat{h}$ is a *bit function* and could be one of the following:



Tighter Bound for Bit Vector Frameworks

Karkare Khedker 2007

- Validity constraints are imposed by the presence of return nodes
- For every cyclic path consisting on Propagate functions, there exists an acyclic path consisting of Propagate functions
- Source of information is a Raise or Lower function
- Target of is a point reachable by a series of Propagate functions
- Identifies interesting path segments that we need to consider for determining a sufficient set of call strings

Relevant Path Segments for Tigher Bound for Bit Vector Frameworks

Which paths in a supergraph are sufficient to construct maximal call strings?

- All paths from C_i to R_i are abstracted away when a new call node C_j is to be suffixed to a call string
- We should consider maximal interprocedurally valid paths in which there is no path from a return node to a call node
- Consider all four combinations

Case A: Source is a call node and target is a call node
Case B: Source is a call node and target is a return node
Case C: Source is a return node and target is also a return node
Case D: Source is a return node and target is a call node: Not relevant

Tighter Length for Bit Vector Frameworks

Case A:
Source is a call node and target is also a call node $P(Entry \rightsquigarrow C_S \rightsquigarrow C_T)$

- No return node, no validity constraints
- Paths $P(Entry \rightsquigarrow C_S)$ and Paths $P(C_S \rightsquigarrow C_T)$ can be acyclic
- A call node may be common to both segments
- At most 2 occurrences of a call site

Tighter Length for Bit Vector Frameworks

Case B:
Source is a call node C_S and target is some return node R_T

- $P(Entry \rightsquigarrow C_S \rightsquigarrow C_T \rightsquigarrow R_T)$
  - Call strings are derived from the paths $P(Entry \rightsquigarrow C_S \rightsquigarrow C_T \rightsquigarrow C_L)$ where C_L is the last call node
  - Thus there are three acyclic segments $P(Entry \rightsquigarrow C_S)$ $P(C_S \rightsquigarrow C_T)$, and $P(C_T \rightsquigarrow C_L)$
  - A call node may be shared in all three
  - At most 3 occurrences of a call site
- $P(Entry \rightsquigarrow C_T \rightsquigarrow C_S \rightsquigarrow R_S \rightsquigarrow R_T)$
  - C_T is required because of validity constraints
  - Call strings are derived from the paths $P(Entry \rightsquigarrow C_T \rightsquigarrow C_S \rightsquigarrow C_L)$ where C_L is the last call node
  - Again, there are three acyclic segments and at most 3 occurrences of a call site

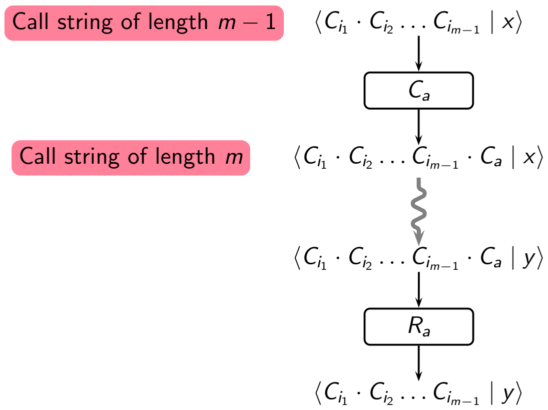
Tighter Length for Bit Vector Frameworks

Case C:
Source is a return node R_S and target is also some return node R_T

- $P(Entry \rightsquigarrow C_T \rightsquigarrow C_S \rightsquigarrow R_S \rightsquigarrow R_T)$
- C_T and C_S are required because of validity constraints
- Call strings are derived from the paths $P(Entry \rightsquigarrow C_T \rightsquigarrow C_S \rightsquigarrow C_L)$ where C_L is the last call node
- Again, there are three acyclic segments and at most 3 occurrences of a call site

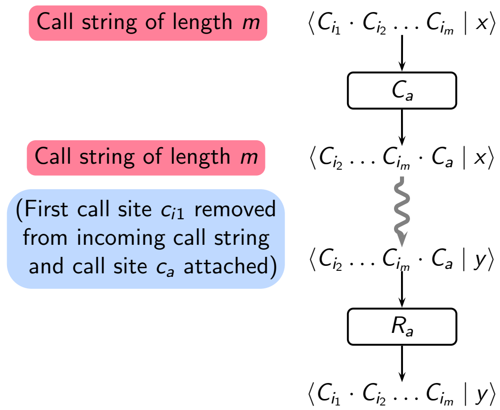
Classical Approximate Approach

- Maintain call string suffixes of upto a given length m.



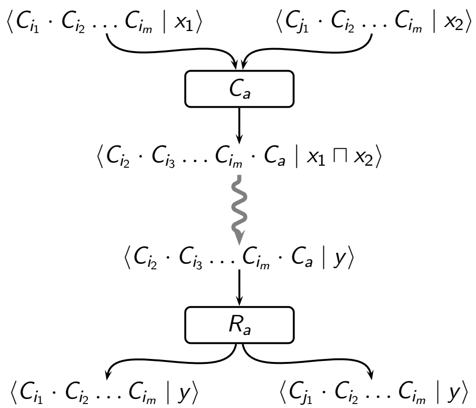
Classical Approximate Approach

- Maintain call string suffixes of upto a given length m.



Classical Approximate Approach

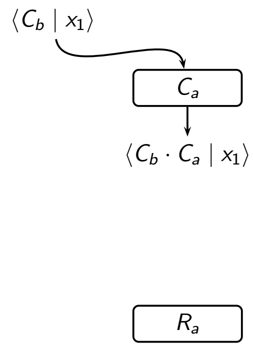
- Maintain call string suffixes of upto a given length m.



- Practical choices of m have been 1 or 2.

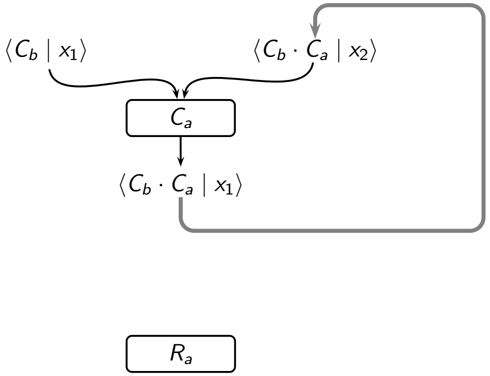
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$



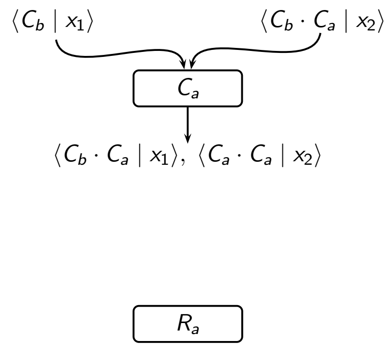
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$



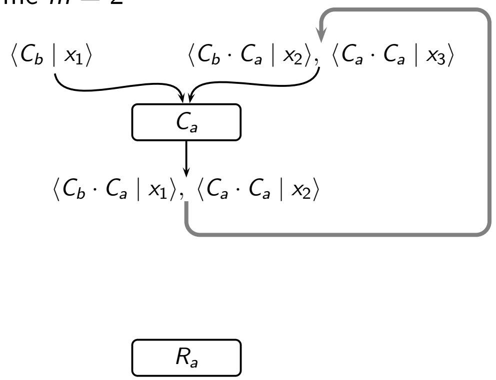
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$



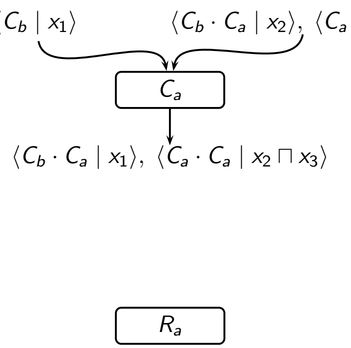
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$



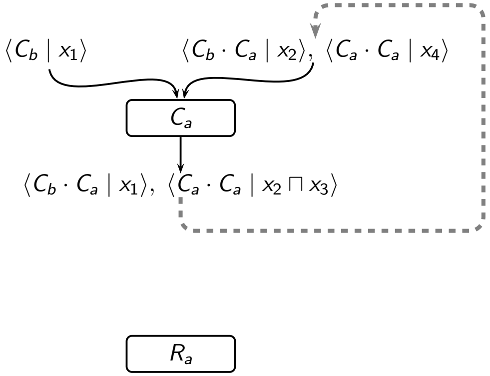
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$



Approximate Call Strings in Presence of Recursion

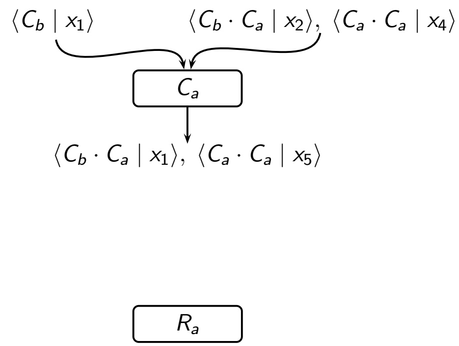
- For simplicity, assume $m = 2$



CS 618 Interprocedural DFA: Classical Call Strings Approach 68/86

Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

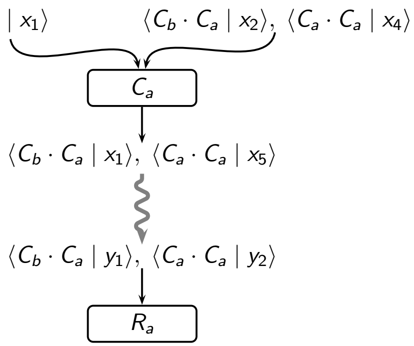


Oct 2009 IIT Bombay

CS 618 Interprocedural DFA: Classical Call Strings Approach 68/86

Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

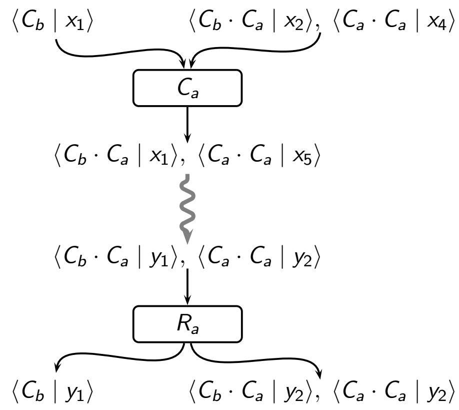


Oct 2009 IIT Bombay

CS 618 Interprocedural DFA: Classical Call Strings Approach 68/86

Approximate Call Strings in Presence of Recursion

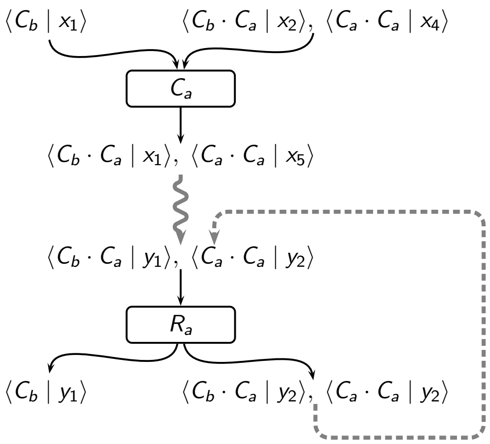
- For simplicity, assume $m = 2$



Oct 2009 IIT Bombay

Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$



Part 6

Modified Call Strings Method

An Overview

- Clearly identifies the exact set of call strings required.
- Value based termination of call string construction. No need to construct call strings upto a fixed length.
- Only as many call strings are constructed as are required.
- Significant reduction in space and time.
- Worst case call string length becomes linear in the size of the lattice instead of the original quadratic.

All this is achieved by a simple change without compromising on the precision, simplicity, and generality of the classical method.

The Limitation of the Classical Call Strings Method

Required langth of the call string is:

- K for non-recursive programs
- $K \cdot (|L| + 1)^2$ for recursive programs

The Modified Algorithm

- Use exactly the same method with this small change:
  - discard redundant call strings at the start of every procedure, and
  - simulate regeneration of call strings at the end of every procedure.
- Intuition:
  - If σ_1 and σ_2 have equal values at S_p,
  - Then, since σ_1 and σ_2 are transformed in the same manner by traversing the same set of paths,
  - The values associated with them will also be transformed in the same manner and will continue to remain equal at E_p.
- Can equivalence classes change?
  - During the analysis, equivalence classes may change in the sense that some call strings may move out of one class and may belong to some other class.
  - However, the invariant that the equivalence classes are same at S_p and E_p still holds.

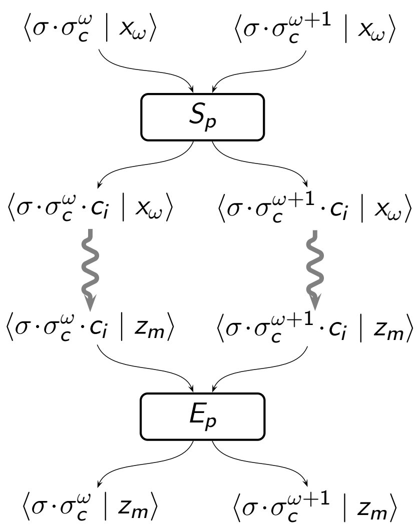
Representation and Regeneration of Call Strings

- Let *shortest*(σ, u) denote the shortest call string which has the same value as σ at u.

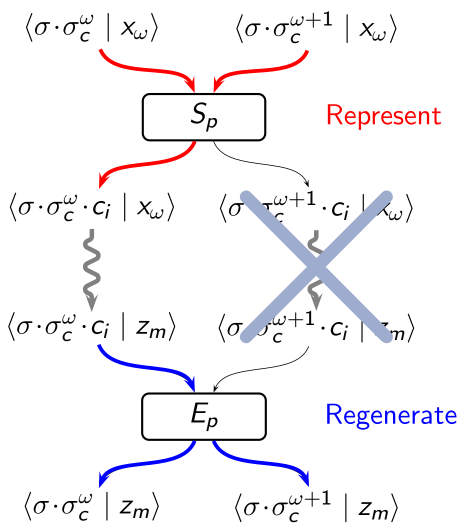
$$\begin{aligned} \textit{represent}(\langle \sigma, x\rangle, S_p) &= \langle \textit{shortest}(\sigma, S_p), x\rangle \\ \textit{regenerate}(\langle \sigma, y\rangle, E_p) &= \{\langle \sigma', y\rangle \mid \sigma \text{ and } \sigma' \text{ have the same value at } S_p\} \end{aligned}$$

- Correctness requirement: Whenever representation is performed at S_p, E_p must be added to the work list
- Efficiency consideration: Desirable order of processing of nodes Intraprocedural noddes $\rightarrow$ call nodes $\rightarrow$ return ndoes

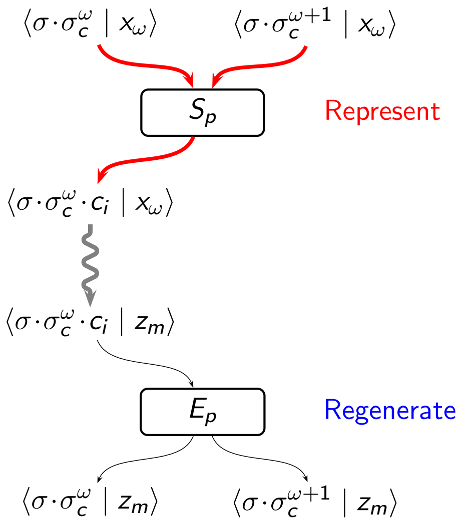
Safety and Precision of Representation and Regeneration



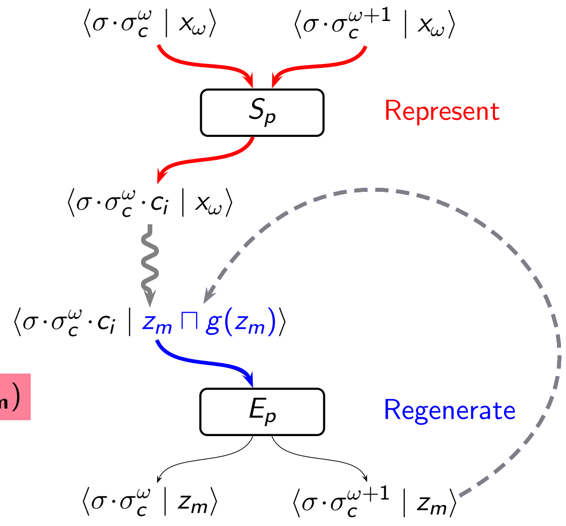
Safety and Precision of Representation and Regeneration



Safety and Precision of Representation and Regeneration

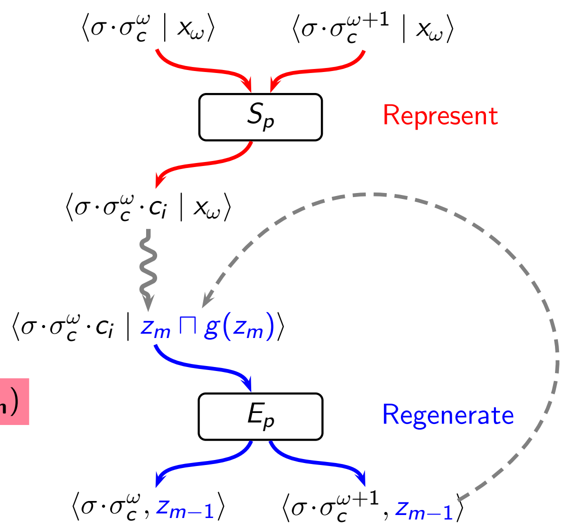


Safety and Precision of Representation and Regeneration



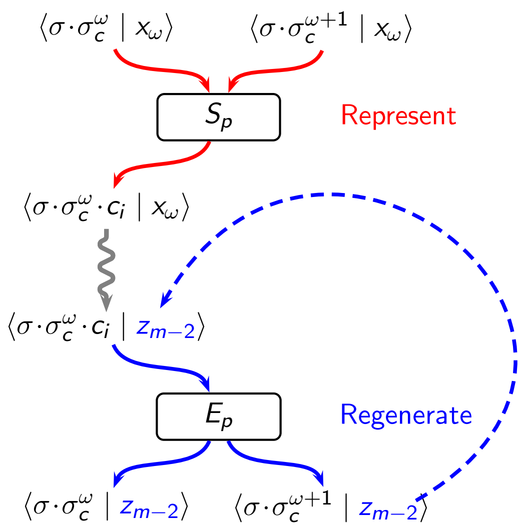
$\mathbf{z_{m-1} = z_m \sqcap g(z_m)}$

Safety and Precision of Representation and Regeneration



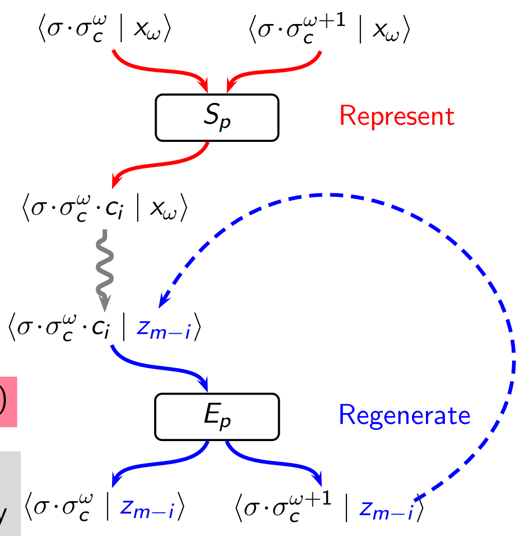
$\mathbf{z_{m-1} = z_m \sqcap g(z_m)}$

Safety and Precision of Representation and Regeneration



$\mathbf{z_{m-2} = z_m \sqcap g(z_{m-1})}$

Safety and Precision of Representation and Regeneration

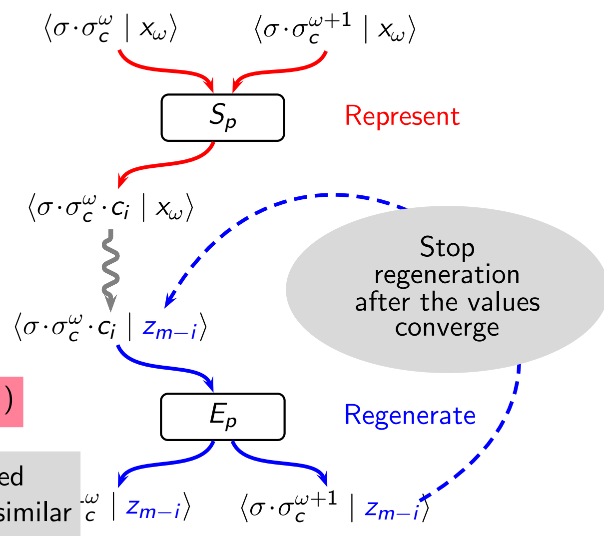


$\mathbf{z_{m-i} = z_m \sqcap g(z_{m-(i+1)})}$

These values are identical to the values computed by the full call strings method

CS 618 Interprocedural DFA: Modified Call Strings Method 73/86

Safety and Precision of Representation and Regeneration



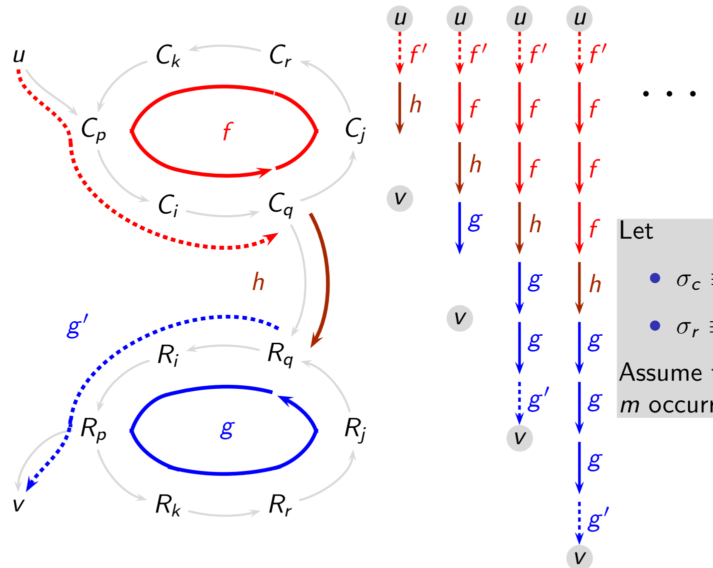
$\mathbf{z_{m-i} = z_m \sqcap g(z_{m-(i+1)})}$

Other values are computed with smaller call strings similar to the full call strings method

Equivalence of The Two Methods

- For non-recursive programs, equivalence is obvious
- For recursive program, we prove equivalence using staircase diagrams

Call Strings for Recursive Contexts

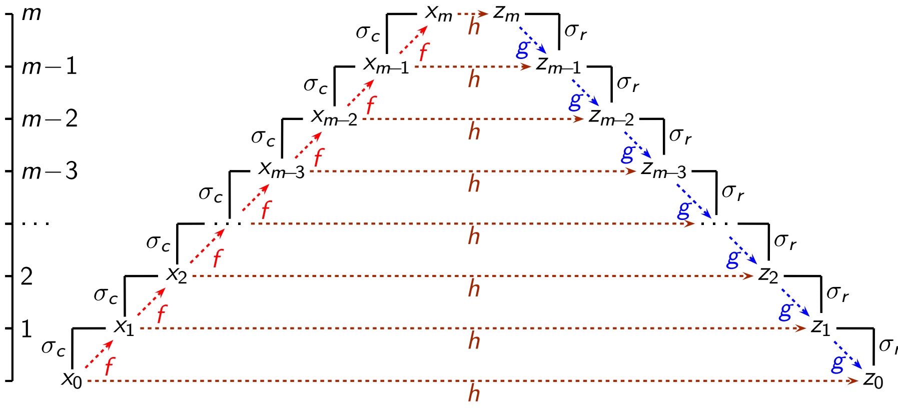


Let

- $\sigma_c \equiv c_j c_r c_k c_p c_i c_q$
- $\sigma_r \equiv r_q r_i r_p r_k r_r r_j$

Assume that we allow upto m occurrences of σ_c

Computing Data Flow Values along Recursive Paths



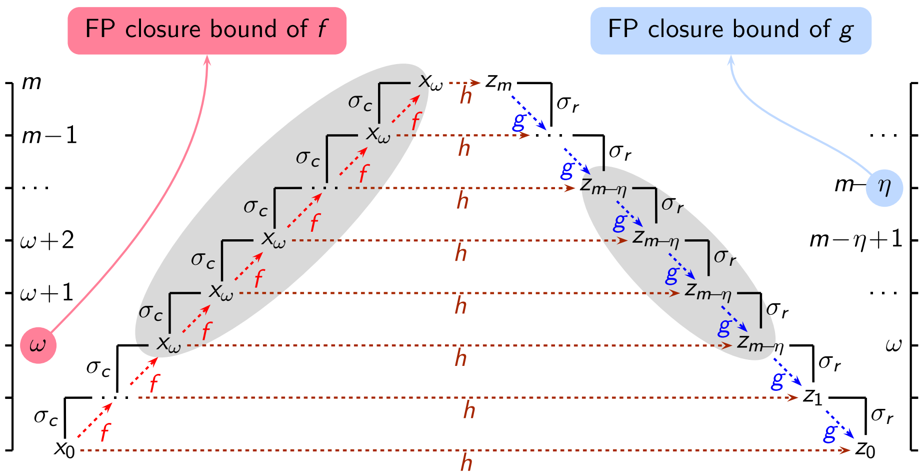
$x_i = f^i(x_0)$ $\qquad$ $z_{m-j} = h(x_{m-j}) \sqcap g(z_{m-j+1})$

Fixed Bound Closure Bound of Flow Function

- $n > 0$ is the fixed point closure bound of $h : L \mapsto L$ if it is the smallest number such that

$$\forall x \in L,\ h^{n+1}(x) = h^n(x)$$

Computation of Data Flow Values along Recursive Paths



$$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases} \qquad z_{m-j} = \begin{cases} h(x_\omega) \sqcap g(z_{m-j+1}) & 0 \le j \le \eta \\ h(x_\omega) \sqcap g(z_{m-\eta}) & \eta < j \le (m-\omega) \\ h(x_j) \sqcap g(z_{m-j+1}) & \text{otherwise} \end{cases}$$

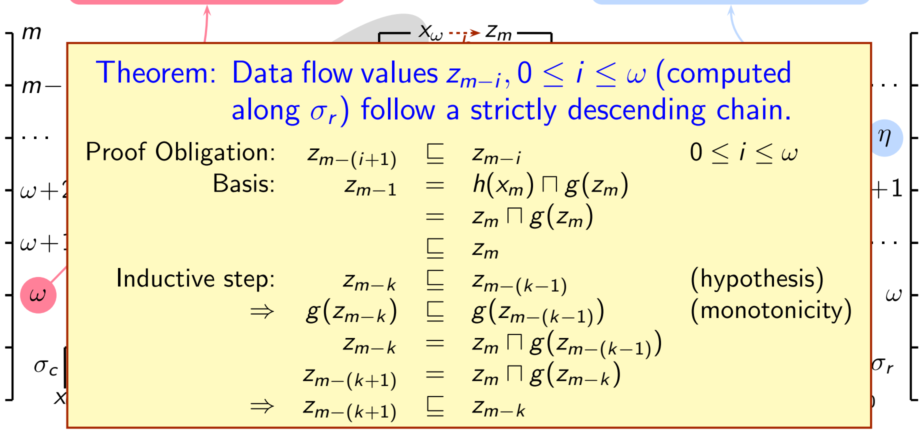
The Moral of the Story

- In the cyclic call sequence, computation begins from the first call string and influences successive call strings.
- In the cyclic return sequence, computation begins from the last call string and influences the preceding call strings.

Bounding the Call String Length Using Data Flow Values

FP closure bound of f

FP closure bound of g



Theorem: Data flow values $z_{m-i}, 0 \leq i \leq \omega$ (computed along σ_r) follow a strictly descending chain.

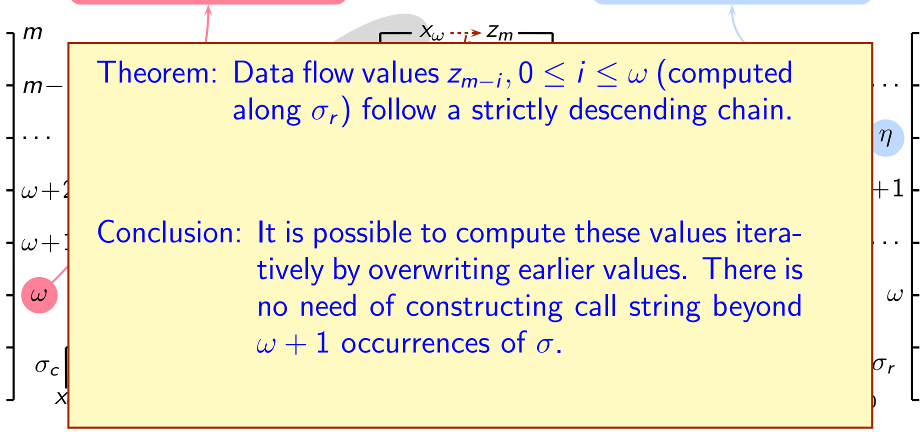
$$
\begin{array}{rrcll}
\text{Proof Obligation:} & z_{m-(i+1)} & \sqsubseteq & z_{m-i} & 0 \leq i \leq \omega \\
\text{Basis:} & z_{m-1} & = & h(x_m) \sqcap g(z_m) & \\
& & = & z_m \sqcap g(z_m) & \\
& & \sqsubseteq & z_m & \\
\text{Inductive step:} & z_{m-k} & \sqsubseteq & z_{m-(k-1)} & \text{(hypothesis)} \\
\Rightarrow & g(z_{m-k}) & \sqsubseteq & g(z_{m-(k-1)}) & \text{(monotonicity)} \\
& z_{m-k} & = & z_m \sqcap g(z_{m-(k-1)}) & \\
& z_{m-(k+1)} & = & z_m \sqcap g(z_{m-k}) & \\
\Rightarrow & z_{m-(k+1)} & \sqsubseteq & z_{m-k} &
\end{array}
$$

$$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases} \qquad z_{m-j} = \begin{cases} h(x_\omega) \sqcap g(z_{m-j+1}) & 0 \leq j \leq \eta \\ h(x_\omega) \sqcap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\ h(x_j) \sqcap g(z_{m-j+1}) & \text{otherwise} \end{cases}$$

Bounding the Call String Length Using Data Flow Values

FP closure bound of f

FP closure bound of g



Theorem: Data flow values $z_{m-i}, 0 \leq i \leq \omega$ (computed along σ_r) follow a strictly descending chain.

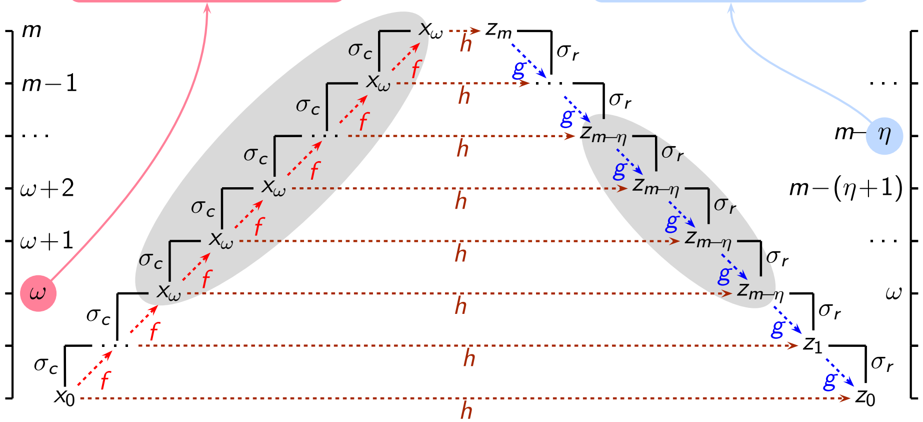
Conclusion: It is possible to compute these values iteratively by overwriting earlier values. There is no need of constructing call string beyond $\omega + 1$ occurrences of σ.

$$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases} \qquad z_{m-j} = \begin{cases} h(x_\omega) \sqcap g(z_{m-j+1}) & 0 \leq j \leq \eta \\ h(x_\omega) \sqcap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\ h(x_j) \sqcap g(z_{m-j+1}) & \text{otherwise} \end{cases}$$

Bounding the Call String Length Using Data Flow Values

FP closure bound of f

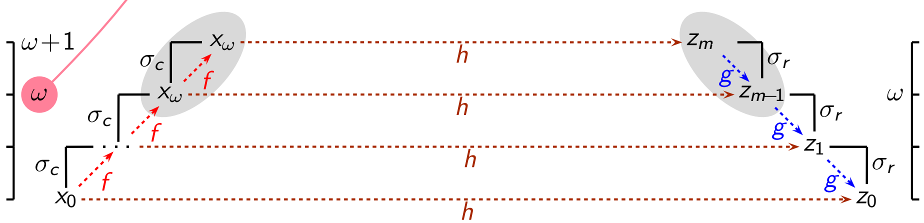
FP closure bound of g



$$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases} \qquad z_{m-j} = \begin{cases} h(x_\omega) \sqcap g(z_{m-j+1}) & 0 \leq j \leq \eta \\ h(x_\omega) \sqcap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\ h(x_j) \sqcap g(z_{m-j+1}) & \text{otherwise} \end{cases}$$

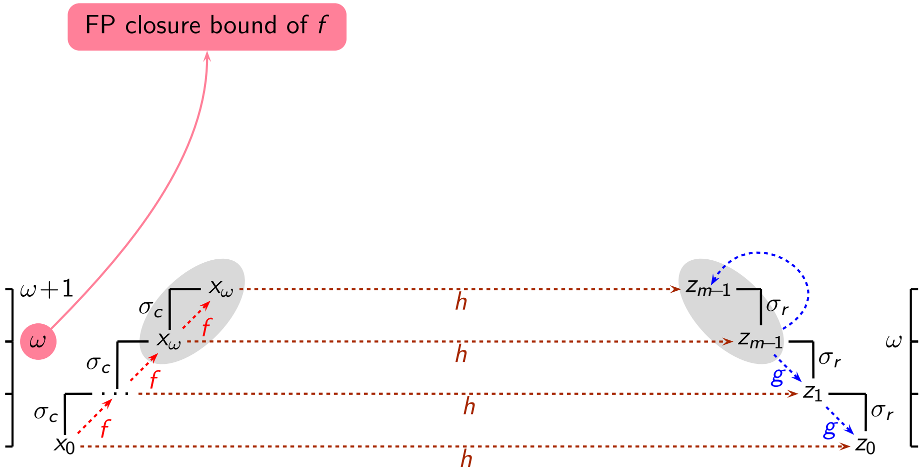
Bounding the Call String Length Using Data Flow Values

FP closure bound of f



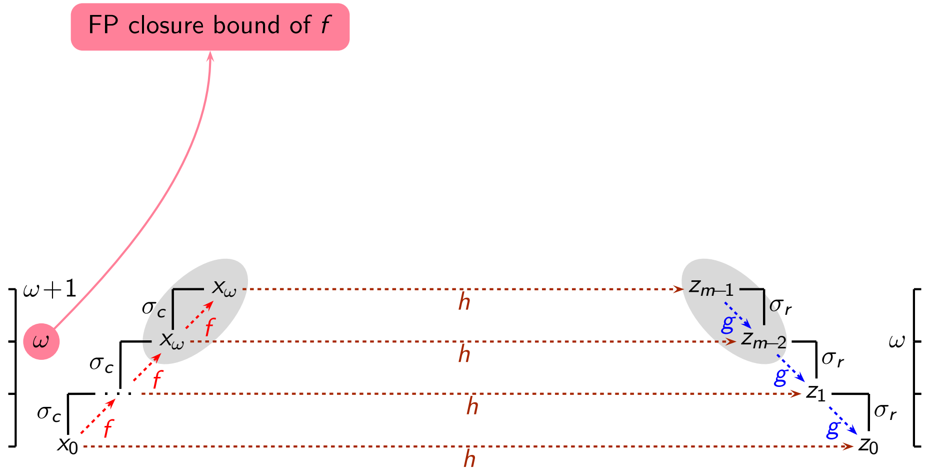
$$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases} \qquad z_{m-j} = \begin{cases} h(x_\omega) \sqcap g(z_{m-j+1}) & 0 \leq j \leq \eta \\ h(x_\omega) \sqcap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\ h(x_j) \sqcap g(z_{m-j+1}) & \text{otherwise} \end{cases}$$

Bounding the Call String Length Using Data Flow Values



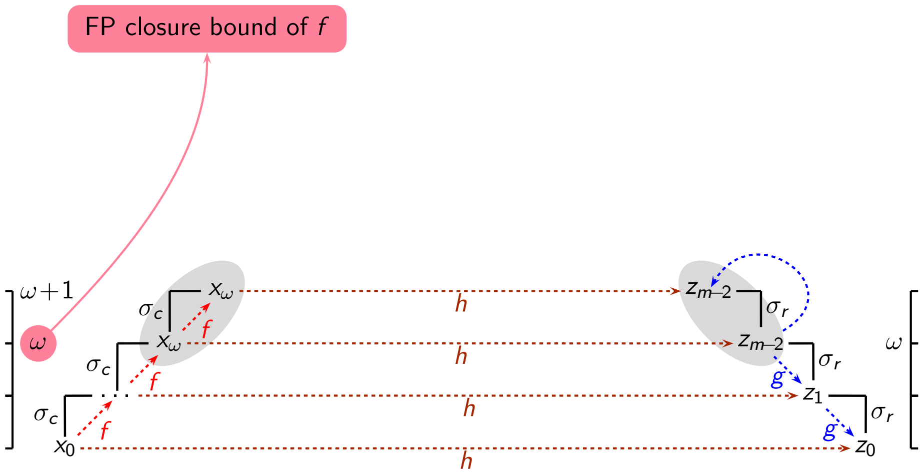
$$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases} \qquad z_{m-j} = \begin{cases} h(x_\omega) \sqcap g(z_{m-j+1}) & 0 \le j \le \eta \\ h(x_\omega) \sqcap g(z_{m-\eta}) & \eta < j \le (m-\omega) \\ h(x_j) \sqcap g(z_{m-j+1}) & \text{otherwise} \end{cases}$$

Bounding the Call String Length Using Data Flow Values



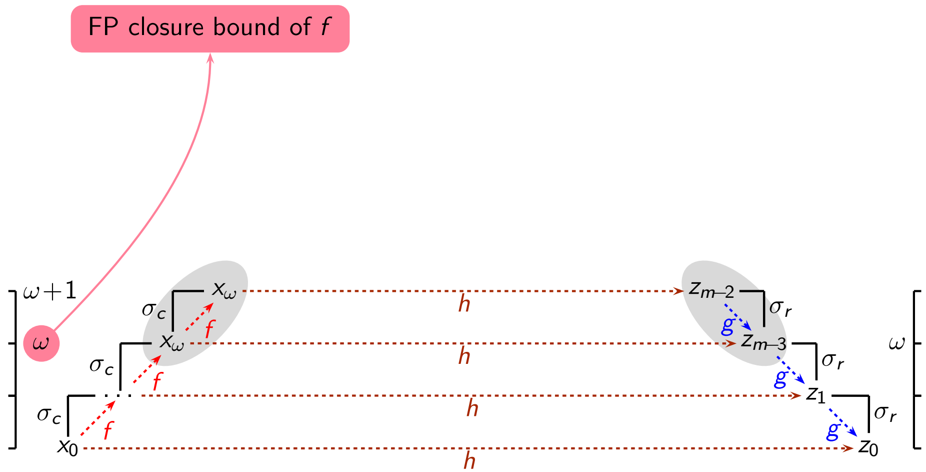
$$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases} \qquad z_{m-j} = \begin{cases} h(x_\omega) \sqcap g(z_{m-j+1}) & 0 \le j \le \eta \\ h(x_\omega) \sqcap g(z_{m-\eta}) & \eta < j \le (m-\omega) \\ h(x_j) \sqcap g(z_{m-j+1}) & \text{otherwise} \end{cases}$$

Bounding the Call String Length Using Data Flow Values



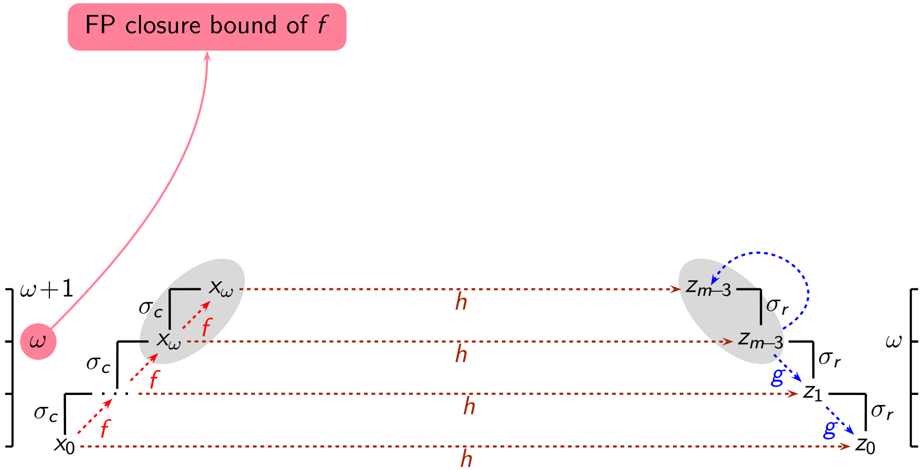
$$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases} \qquad z_{m-j} = \begin{cases} h(x_\omega) \sqcap g(z_{m-j+1}) & 0 \le j \le \eta \\ h(x_\omega) \sqcap g(z_{m-\eta}) & \eta < j \le (m-\omega) \\ h(x_j) \sqcap g(z_{m-j+1}) & \text{otherwise} \end{cases}$$

Bounding the Call String Length Using Data Flow Values



$$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases} \qquad z_{m-j} = \begin{cases} h(x_\omega) \sqcap g(z_{m-j+1}) & 0 \le j \le \eta \\ h(x_\omega) \sqcap g(z_{m-\eta}) & \eta < j \le (m-\omega) \\ h(x_j) \sqcap g(z_{m-j+1}) & \text{otherwise} \end{cases}$$

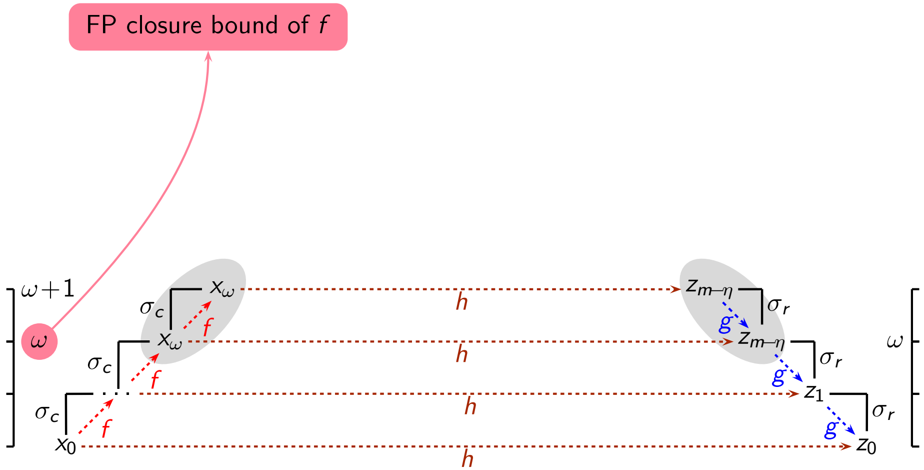
Bounding the Call String Length Using Data Flow Values



$$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases} \qquad z_{m-j} = \begin{cases} h(x_\omega) \sqcap g(z_{m-j+1}) & 0 \le j \le \eta \\ h(x_\omega) \sqcap g(z_{m-\eta}) & \eta < j \le (m-\omega) \\ h(x_j) \sqcap g(z_{m-j+1}) & \text{otherwise} \end{cases}$$

CS 618 Interprocedural DFA: Modified Call Strings Method 81/86

Bounding the Call String Length Using Data Flow Values

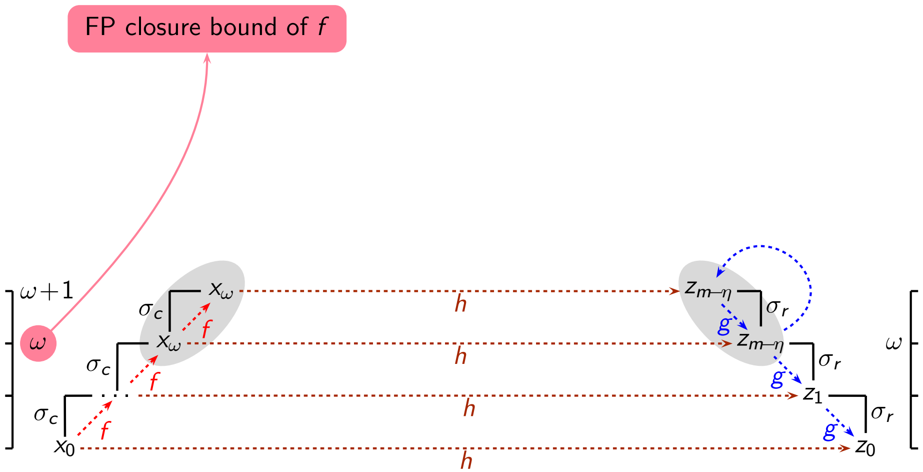


$$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases} \qquad z_{m-j} = \begin{cases} h(x_\omega) \sqcap g(z_{m-j+1}) & 0 \le j \le \eta \\ h(x_\omega) \sqcap g(z_{m-\eta}) & \eta < j \le (m-\omega) \\ h(x_j) \sqcap g(z_{m-j+1}) & \text{otherwise} \end{cases}$$

Oct 2009 IIT Bombay

CS 618 Interprocedural DFA: Modified Call Strings Method 81/86

Bounding the Call String Length Using Data Flow Values



$$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases} \qquad z_{m-j} = \begin{cases} h(x_\omega) \sqcap g(z_{m-j+1}) & 0 \le j \le \eta \\ h(x_\omega) \sqcap g(z_{m-\eta}) & \eta < j \le (m-\omega) \\ h(x_j) \sqcap g(z_{m-j+1}) & \text{otherwise} \end{cases}$$

Oct 2009 IIT Bombay

Worst Case Length Bound

- Consider a call string $\sigma = \ldots (C_i)^1 \ldots (C_i)^2 \ldots (C_i)^3 \ldots (C_i)^j \ldots$
  Let $j \ge |L| + 1$
  Let C_i call procedure p
- All call string ending with C_i reach entry S_p
- Since only $|L|$ distinct values are possible, by the pigeon hole principle, at least two prefixes ending with C_i will carry the same data flow value to S_p.
  - The longer prefix will get represented by the shorter prefix
  - Since one more C_i is may be suffixed to discover fixed point, $j \le |L| + 1$
- Worst case length in the proposed variant $= K \times (|L| + 1)$
- Original required length $= K \times (|L| + 1)^2$

Approximate Version

- For framework with infinite lattices, a fixed point for cyclic call sequence may not exist.
- Use a demand driven approach:
  - After a dynamically definable limit (say a number j),
  - Start merging the values and associate them with the last call string
  - Let

$$\sigma_j = \ldots (C_i)^1 \ldots (C_i)^2 \ldots (C_i)^3 \ldots (C_i)^j \ldots$$
$$\sigma_{j+1} = \ldots (C_i)^1 \ldots (C_i)^2 \ldots (C_i)^3 \ldots (C_i)^j \ldots (C_i)^{j+1} \ldots$$

  - Represent $\langle \sigma_j \mid x_j \rangle$ and $\langle \sigma_{j+1} \mid x_{j+1} \rangle$ by $\langle \sigma^j \mid x_j \sqcap x_{j+1} \rangle$
- Context sensitive for a depth j of recursion. Context insensitive beyond that.
- Assumption: Height of the lattice is finite.

Reaching Definitions Analysis in GCC 4.0

Program	LoC	$\#F$	$\#C$	$3K$ length bound				Proposed Approach		
				K	#CS	Max	Time	#CS	Max	Time
hanoi	33	2	4	4	100000+	99922	3973×10^3	8	7	2.37
bit_gray	53	5	11	7	100000+	31374	2705×10^3	17	6	3.83
analyzer	288	14	20	2	21	4	20.33	21	4	1.39
distray	331	9	21	6	96	28	322.41	22	4	1.11
mason	350	9	13	8	100000+	22143	432×10^3	14	4	0.43
fourinarow	676	17	45	5	510	158	397.76	46	7	1.86
sim	1146	13	45	8	100000+	33546	1427×10^3	211	105	234.16
181_mcf	1299	17	24	6	32789	32767	484×10^3	41	11	5.15
256_bzip2	3320	63	198	7	492	63	258.33	406	34	200.19

- LoC is the number of lines of code,
- $\#F$ is the number of procedures,
- $\#C$ is the number of call sites,
- $\#CS$ is the number of call strings
- Max denotes the maximum number of call strings reaching any node.
- Analysis time is in milliseconds.

(Implementation was carried out by Seema Ravandale.)

Some Observations

- Compromising on precision may not be necessary for efficiency.
- Separating the necessary information from redundant information is much more significant.
- Data flow propagation in real programs seems to involve only a small subset of all possible values. Much fewer changes than the theoretically possible worst case number of changes.
- A precise modelling of the process of analysis is often an eye opener.

Tutorial Problem

Perform may points-to analysis using modified call strings method. Make conservative assumptions about must points-to information.

```
main()
{  x = &y;
   z = &x;
   y = &z;
   p(); /* C1 */
}

p()
{  if (...)
   {  p(); /* C2 */
      x = *x;
   }
}
```

- Number of distinct call sites in a call chain $K = 2$.
- Number of variables: 3
- Number of distinct points-to pairs: $3 \times 3 = 9$
- L is powerset of all points-to pairs
- $\mid L \mid = 2^9$
- Length of the longest call string in Sharir-Pnueli method $2 \times (|L| + 1)^2 = 2^{19} + 2^{10} + 1 = 5,25,313$
- All call strings upto this length must be constructed by the Sharir-Pnueli method!

Tutorial Problem

Perform may points-to analysis using modified call strings method. Make conservative assumptions about must points-to information.

```
main()
{  x = &y;
   z = &x;
   y = &z;
   p(); /* C1 */
}

p()
{  if (...)
   {  p(); /* C2 */
      x = *x;
   }
}
```

- Modified call strings method requires only three call strings: λ, c_1, and c_1c_2