Interprocedural Data Flow Analysis

Uday P. Khedker

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

May 2011

Part 1

About These Slides

Copyright

Outline

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

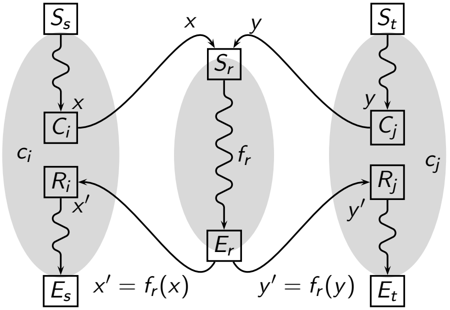
Part 3

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

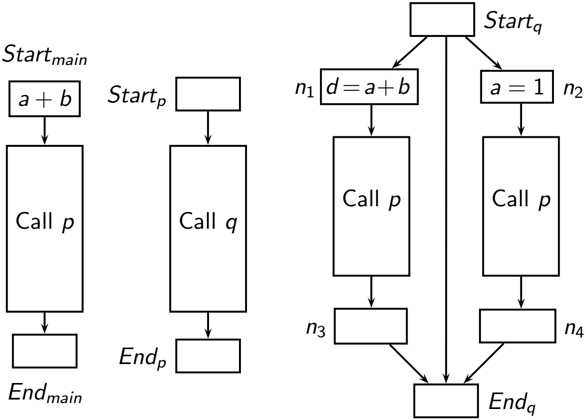
  Answering questions about formal parameters and global variables:
  - Which variables are constant?
  - Which variables aliased with each other?
  - Which locations can a pointer variable point to?

- 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)

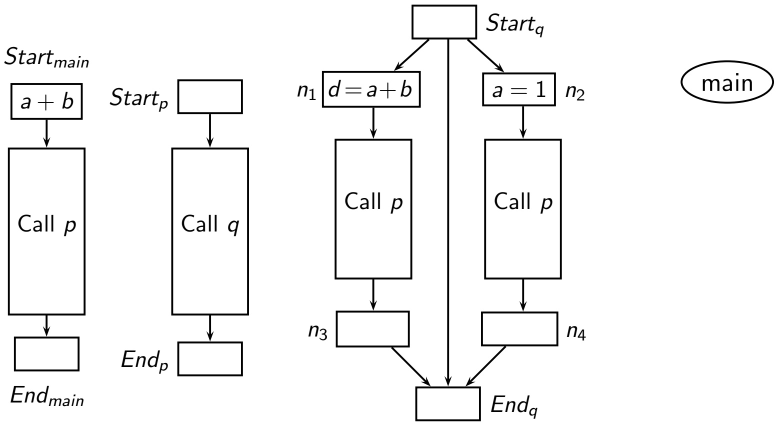
- Most of the above questions may have a *May* or *Must* qualifier.

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



Supergraphs of procedures

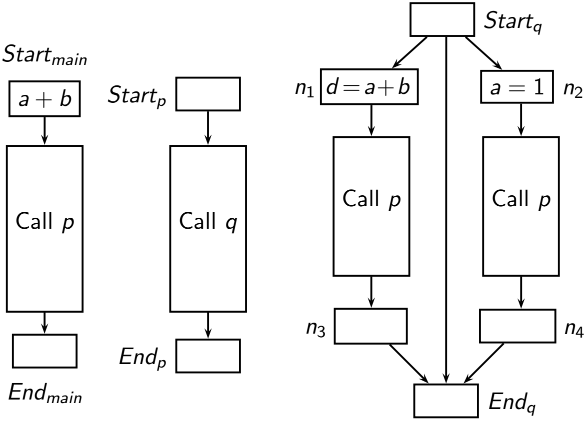
Program Representation for Interprocedural Data Flow Analysis: Call Multi-Graph



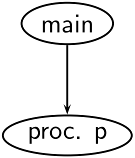
Supergraphs of procedures

Call multi-graph

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

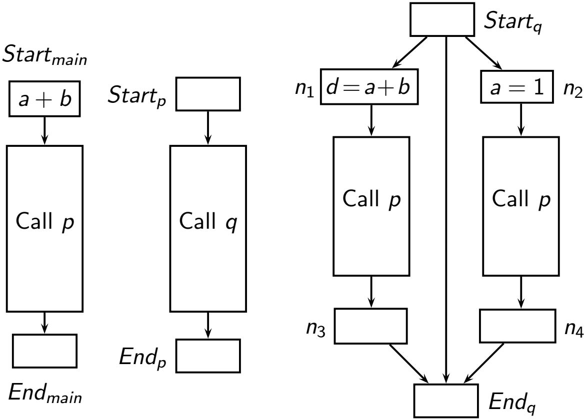


Supergraphs of procedures

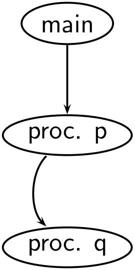


Call multi-graph

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

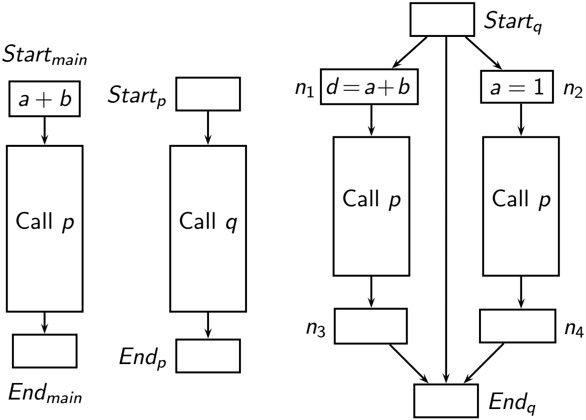


Supergraphs of procedures

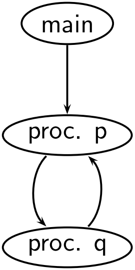


Call multi-graph

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

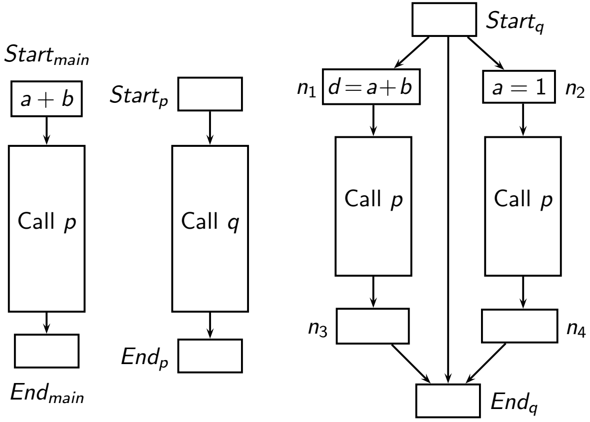


Supergraphs of procedures

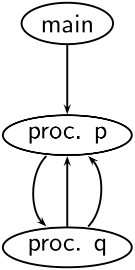


Call multi-graph

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

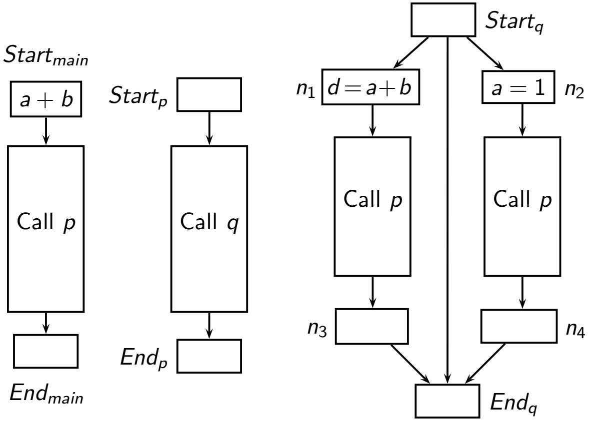


Supergraphs of procedures

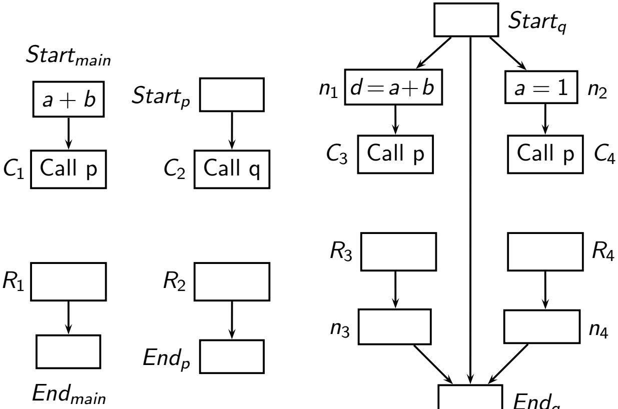


Call multi-graph

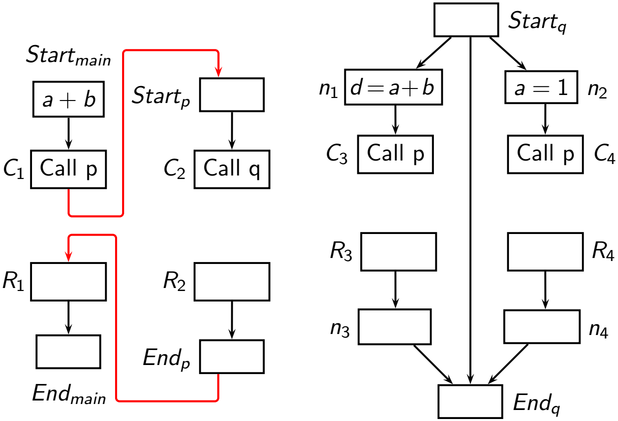
Program Representation for Interprocedural Data Flow Analysis: Supergraph



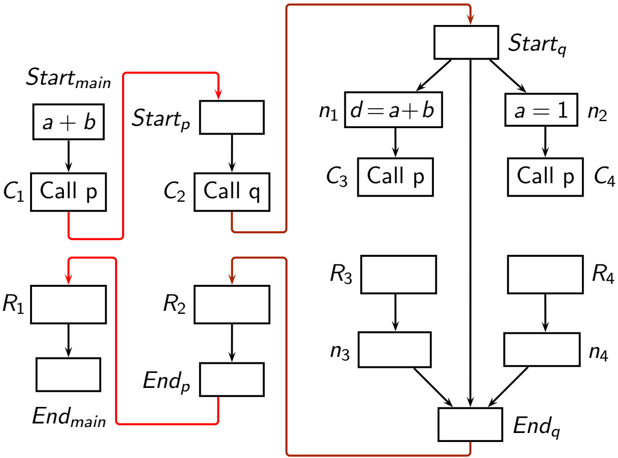
Program Representation for Interprocedural Data Flow Analysis: Supergraph



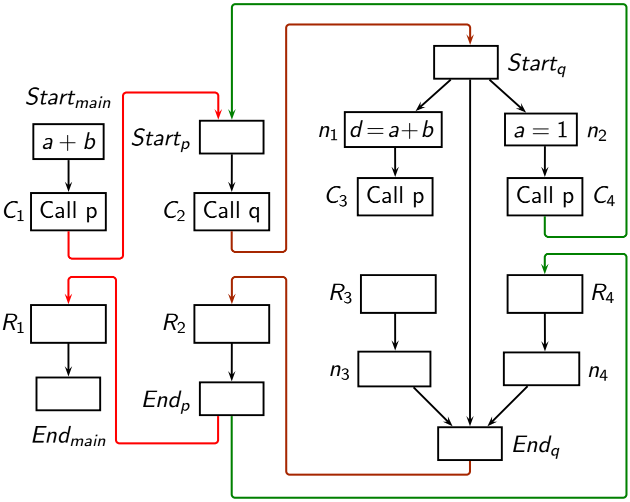
Program Representation for Interprocedural Data Flow Analysis: Supergraph



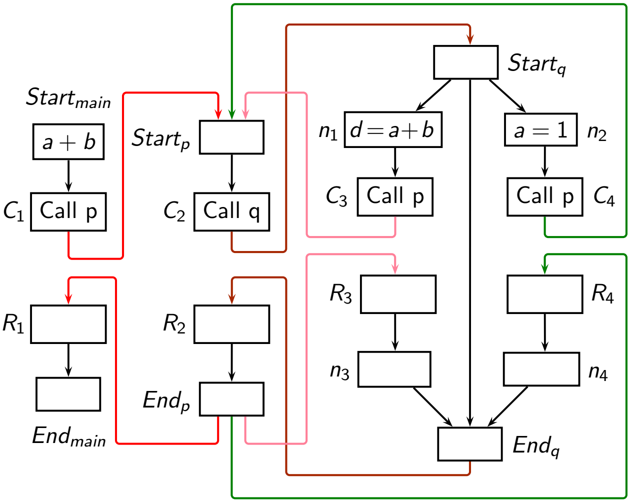
Program Representation for Interprocedural Data Flow Analysis: Supergraph



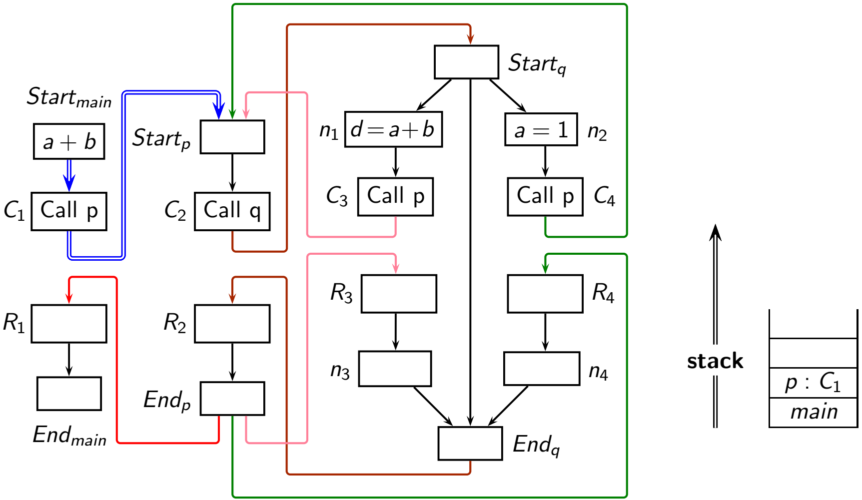
Program Representation for Interprocedural Data Flow Analysis: Supergraph



Program Representation for Interprocedural Data Flow Analysis: Supergraph

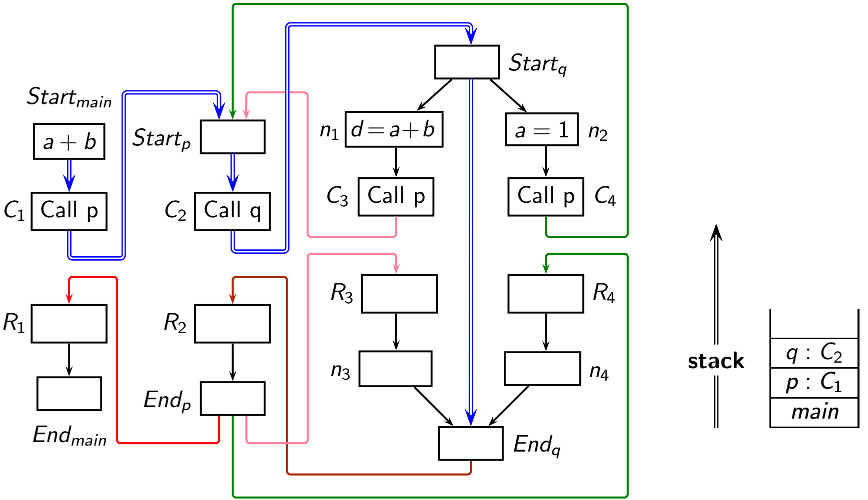


Validity of Interprocedural Control Flow Paths



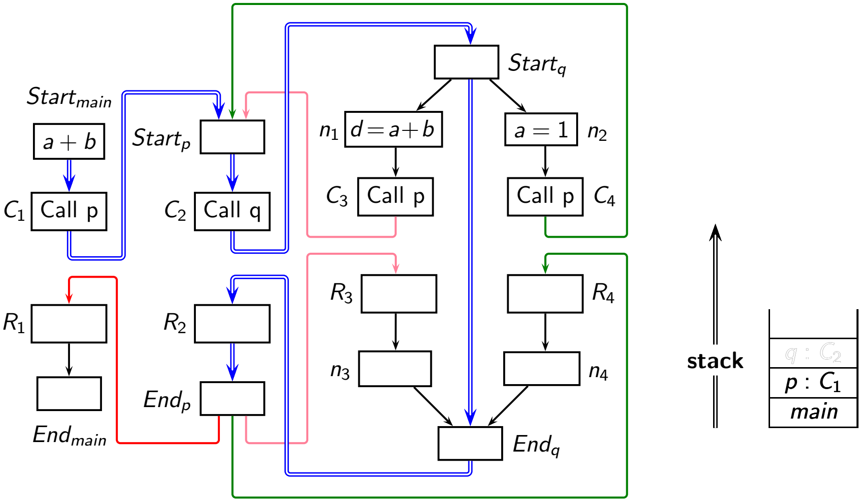
Interprocedurally valid control flow path

Validity of Interprocedural Control Flow Paths



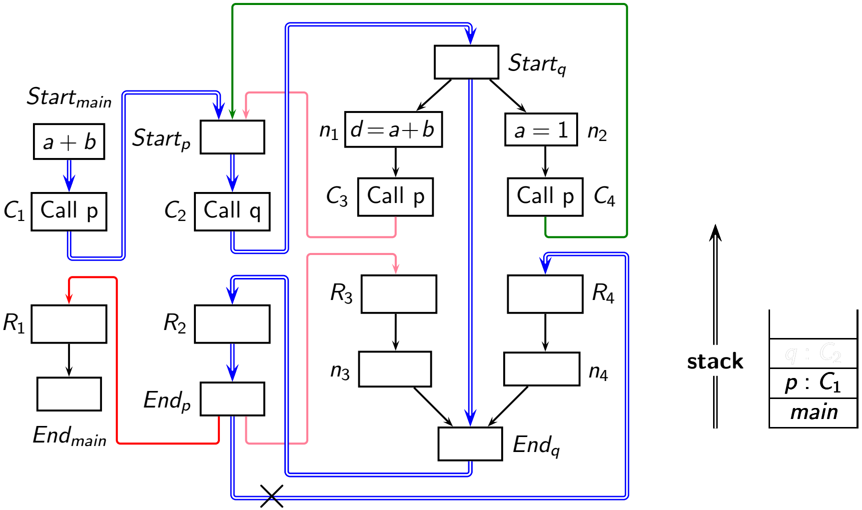
Interprocedurally valid control flow path

Validity of Interprocedural Control Flow Paths



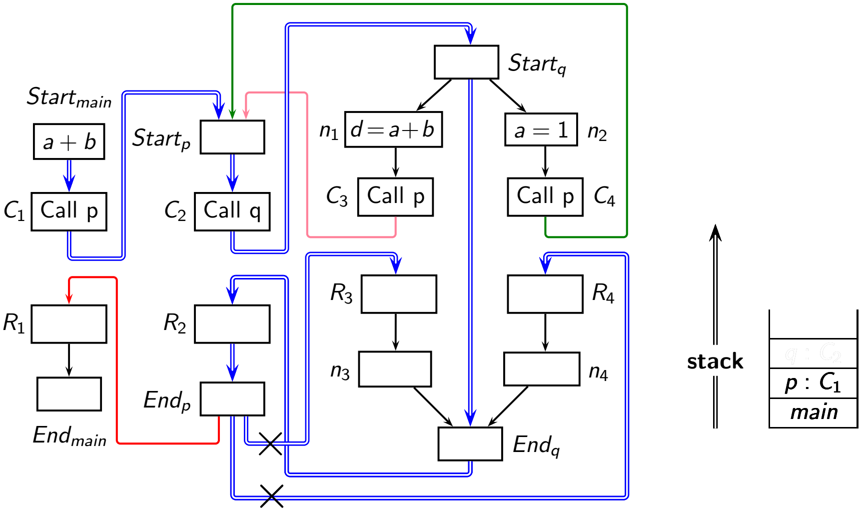
Interprocedurally valid control flow path

Validity of Interprocedural Control Flow Paths



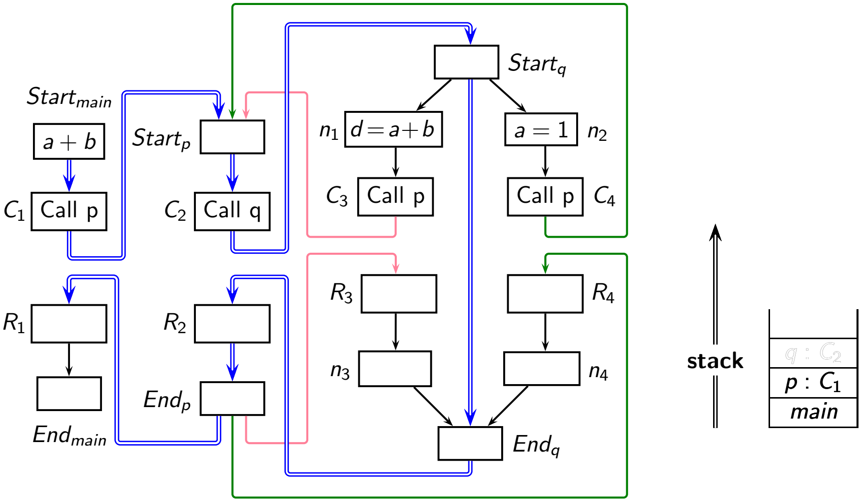
Interprocedurally invalid control flow path

Validity of Interprocedural Control Flow Paths



Interprocedurally invalid control flow path

Validity of Interprocedural Control Flow Paths



Interprocedurally valid control flow path

Safety, Precision, and Efficiency of Data Flow Analysis

- Data flow analysis uses static representation of programs to compute summary information along paths

Safety, Precision, and Efficiency of Data Flow Analysis

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

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

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.

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.

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

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

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

Flow and Context Sensitivity

- Flow sensitive analysis:
  Considers intraprocedurally valid paths
- Context sensitive analysis:
  Considers interprocedurally valid paths

Flow and Context Sensitivity

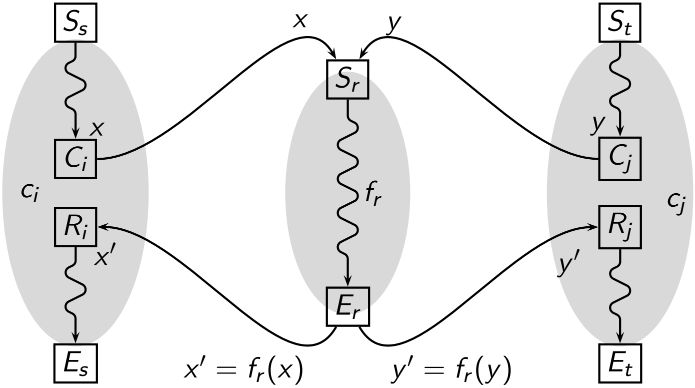
- Flow sensitive analysis:
  Considers intraprocedurally valid paths
- Context sensitive analysis:
  Considers interprocedurally valid paths
- For maximum statically attainable precision, analysis must be both flow and context sensitive.

Flow and Context Sensitivity

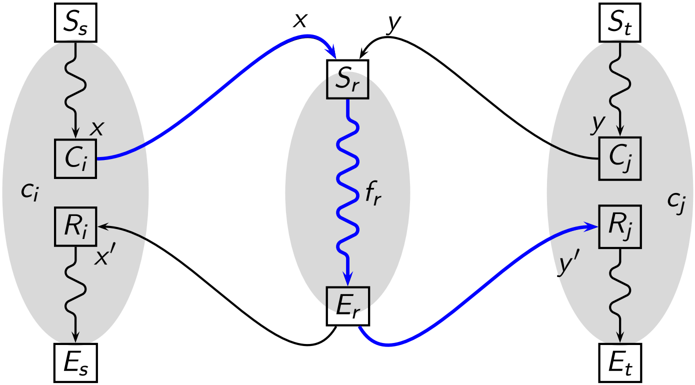
- Flow sensitive analysis:
  Considers intraprocedurally valid paths
- Context sensitive analysis:
  Considers interprocedurally valid paths
- For maximum statically attainable precision, analysis must be both flow and context sensitive.

MFP computation restricted to valid paths only

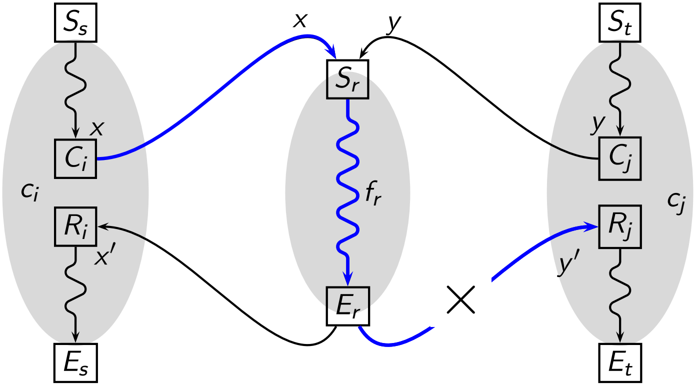
Context Sensitivity in Interprocedural Analysis



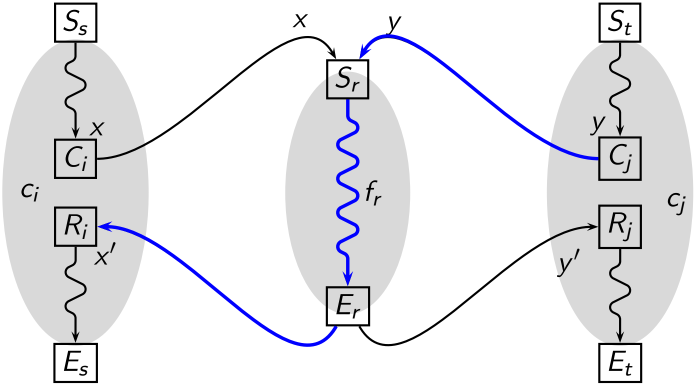
Context Sensitivity in Interprocedural Analysis



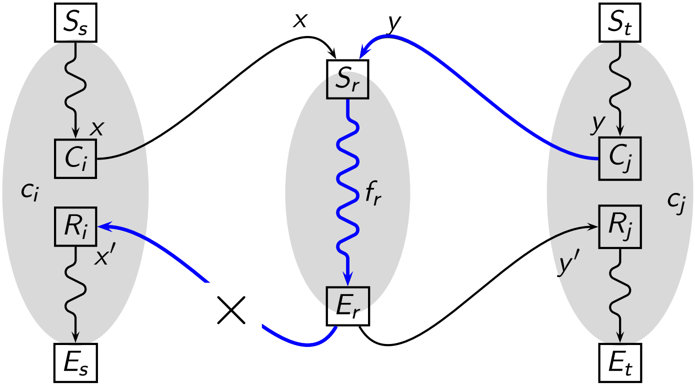
Context Sensitivity in Interprocedural Analysis



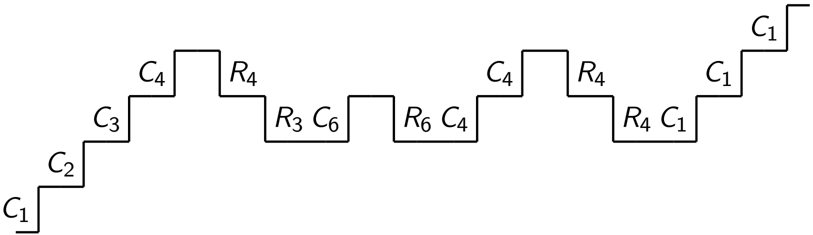
Context Sensitivity in Interprocedural Analysis



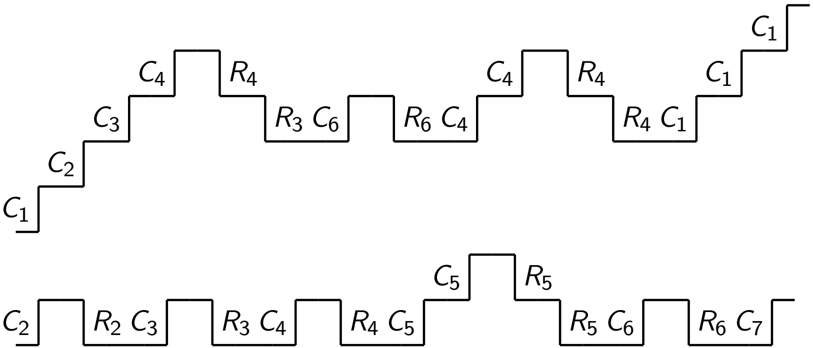
Context Sensitivity in Interprocedural Analysis



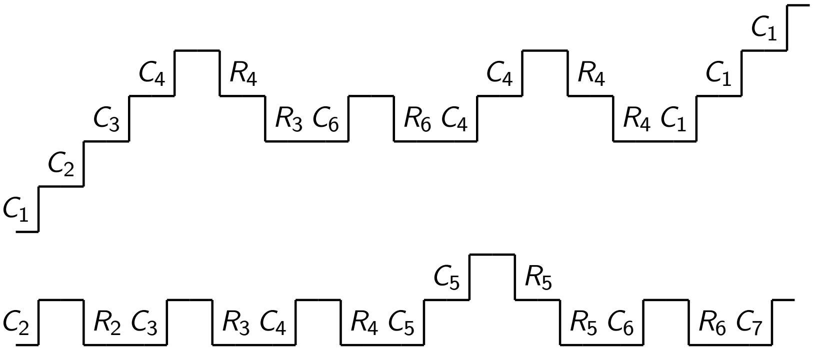
Staircase Diagrams of Interprocedurally Valid Paths



Staircase Diagrams of Interprocedurally Valid Paths

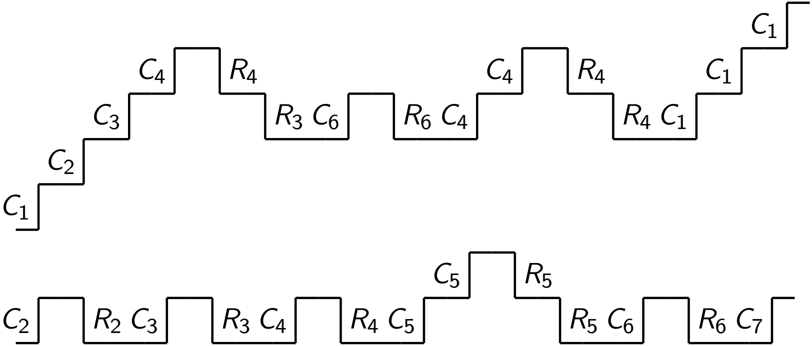


Staircase Diagrams of Interprocedurally Valid Paths



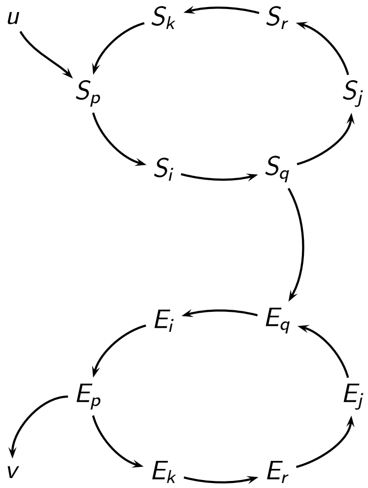
- "You can descend only as much as you have ascended!"

Staircase Diagrams of Interprocedurally Valid Paths

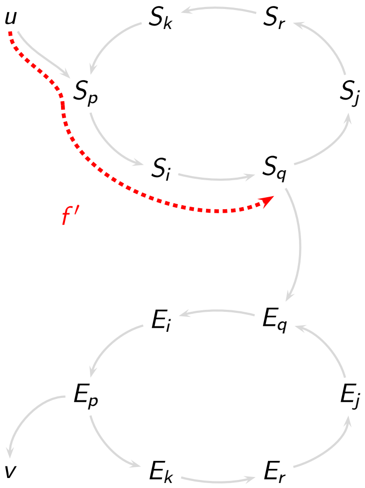


- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step.

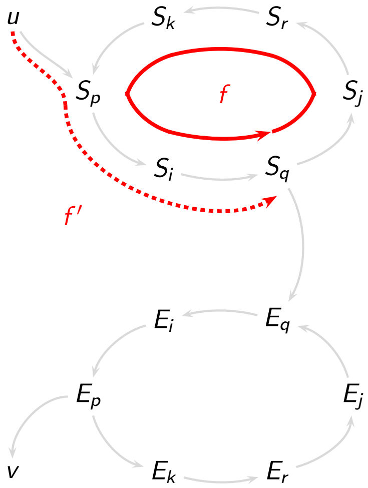
Context Sensitivity in Presence of Recursion



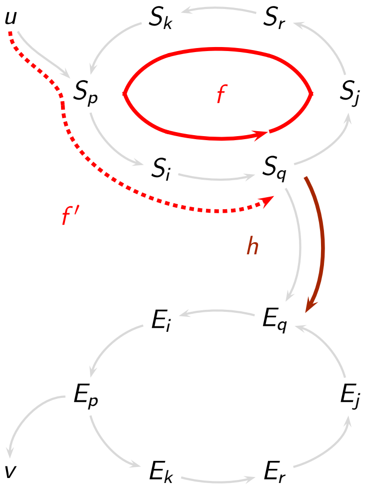
Context Sensitivity in Presence of Recursion



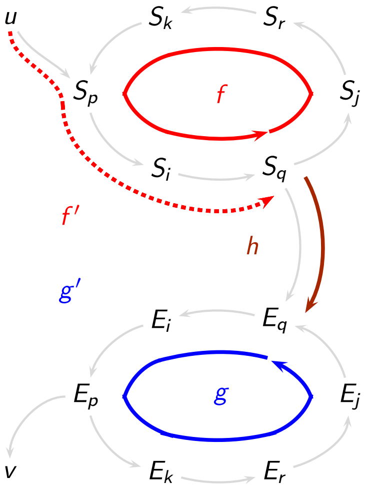
Context Sensitivity in Presence of Recursion



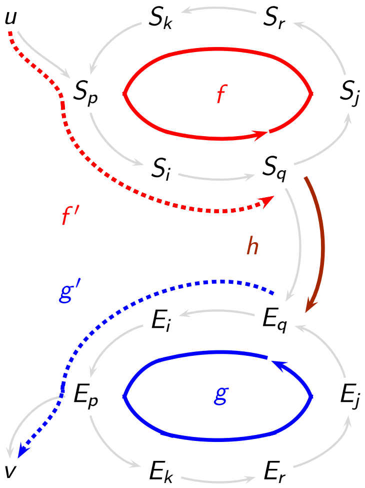
Context Sensitivity in Presence of Recursion



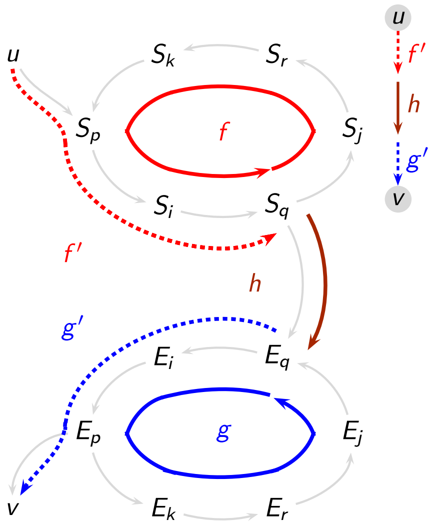
Context Sensitivity in Presence of Recursion



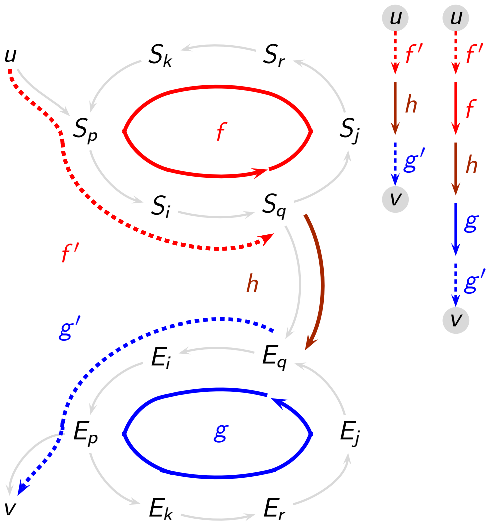
Context Sensitivity in Presence of Recursion



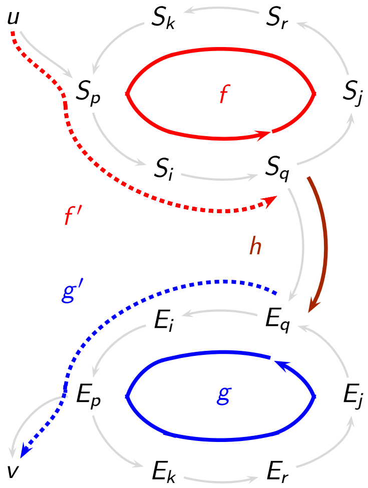
Context Sensitivity in Presence of Recursion

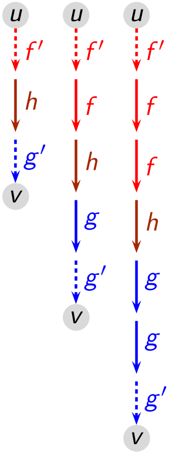


Context Sensitivity in Presence of Recursion

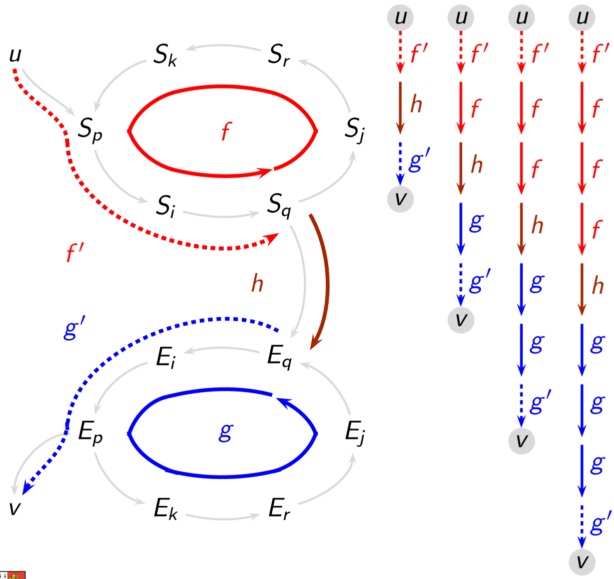


Context Sensitivity in Presence of Recursion

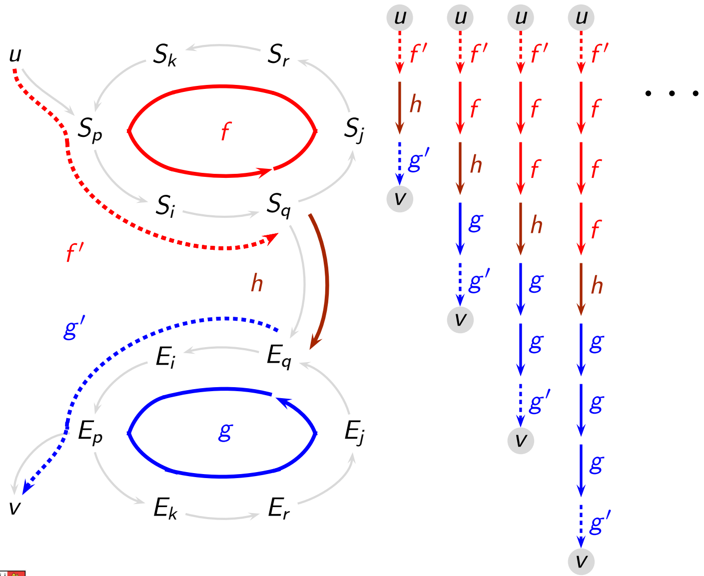




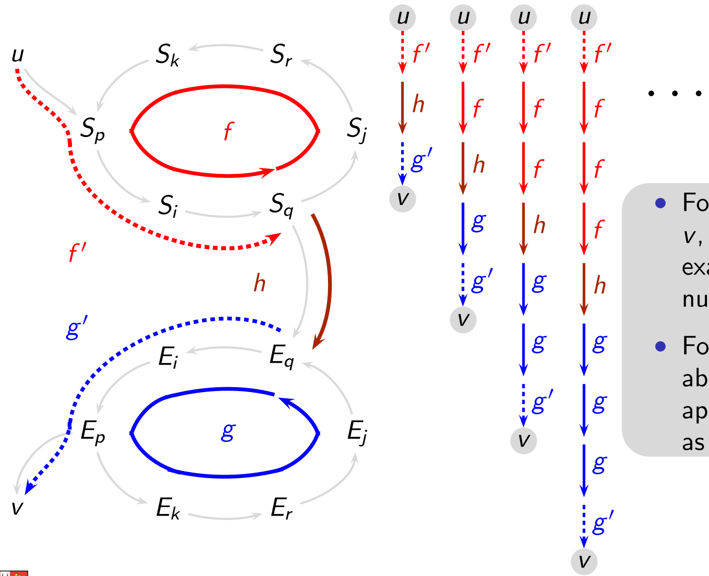
Context Sensitivity in Presence of Recursion



Context Sensitivity in Presence of Recursion

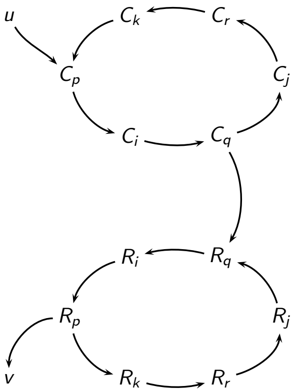


Context Sensitivity in Presence of Recursion

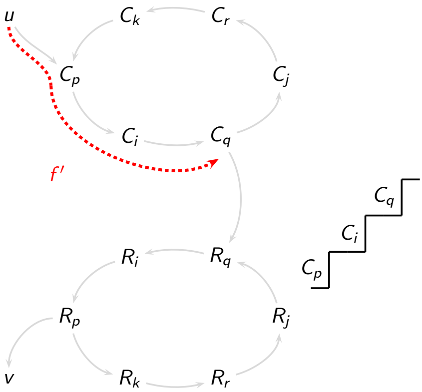


- For a path from u to v, g must be applied exactly the same number of times as f.
- For a prefix of the above path, g can be applied only at most as many times as f.

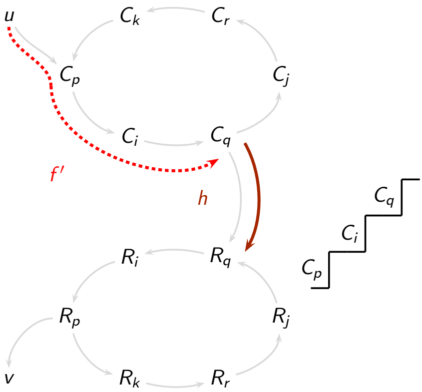
Staircase Diagrams of Interprocedurally Valid Paths



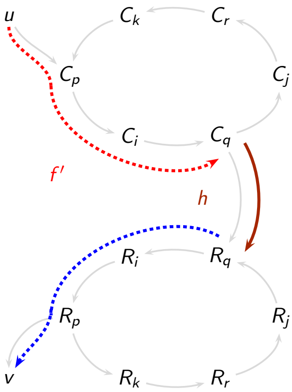
Staircase Diagrams of Interprocedurally Valid Paths

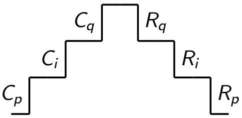


Staircase Diagrams of Interprocedurally Valid Paths

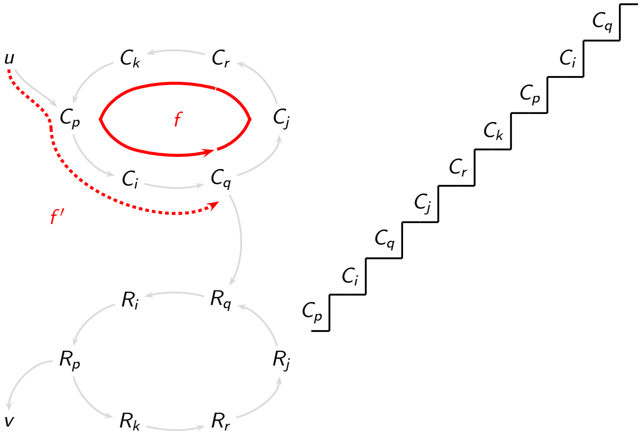


Staircase Diagrams of Interprocedurally Valid Paths

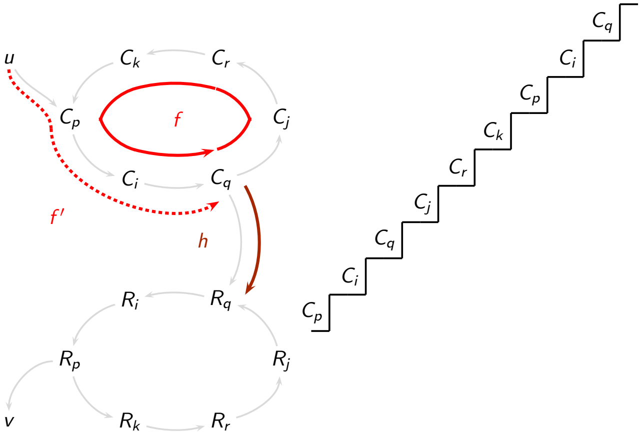




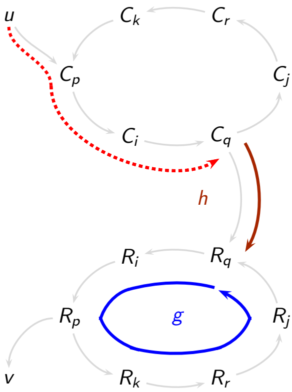
Staircase Diagrams of Interprocedurally Valid Paths

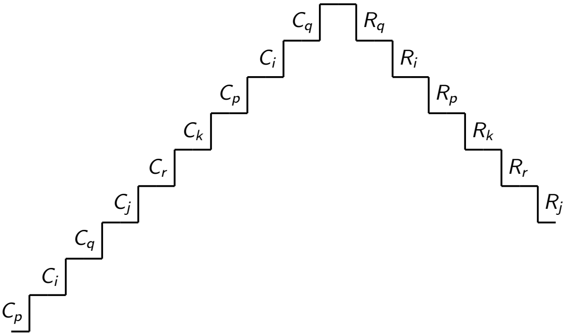


Staircase Diagrams of Interprocedurally Valid Paths



Staircase Diagrams of Interprocedurally Valid Paths



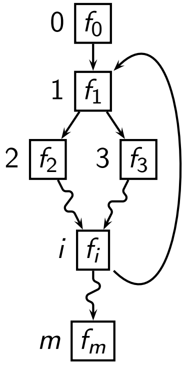


Flow Insensitivity in Data Flow Analysis

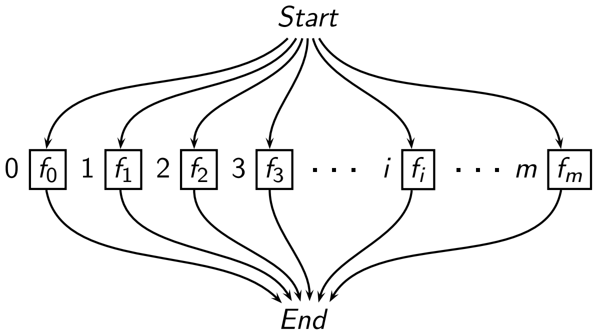
- Assumption: Statements can be executed in any order.
- Instead of computing point-specific data flow information, summary data flow information is computed.
  The summary information is required to be a safe approximation of point-specific information for each point.
- $\text{Kill}_n(\text{x})$ component 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 $DepGen_n(\mathrm{x}) = \emptyset$, and $\mathrm{Kill}_n(X)$ is ignored for all n



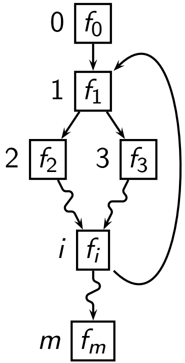
Control flow graph



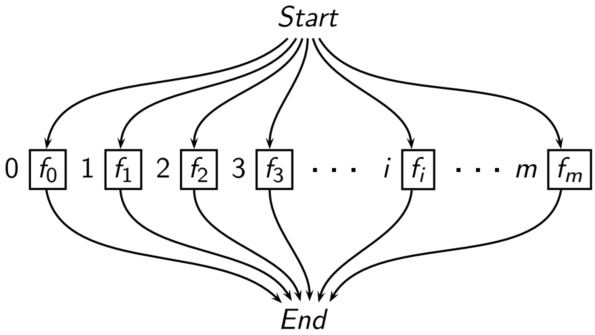
Flow insensitive analysis

Flow Insensitivity in Data Flow Analysis

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



Control flow graph

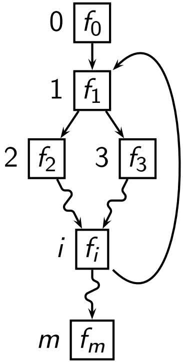


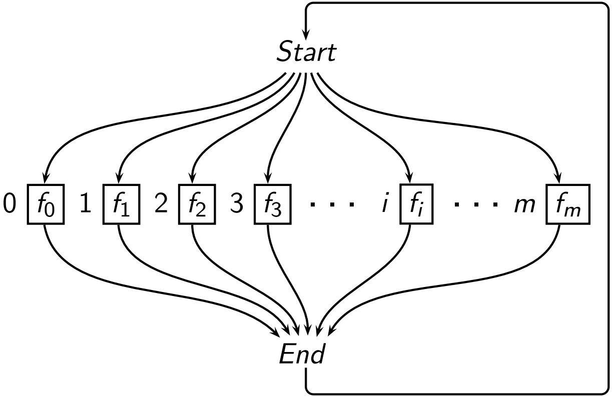
Flow insensitive analysis

Function composition is replaced by function confluence

Flow Insensitivity in Data Flow Analysis

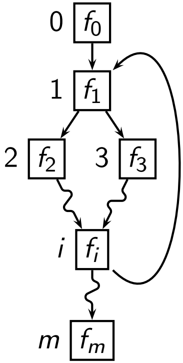
If $DepGen_n(x) \neq \emptyset$

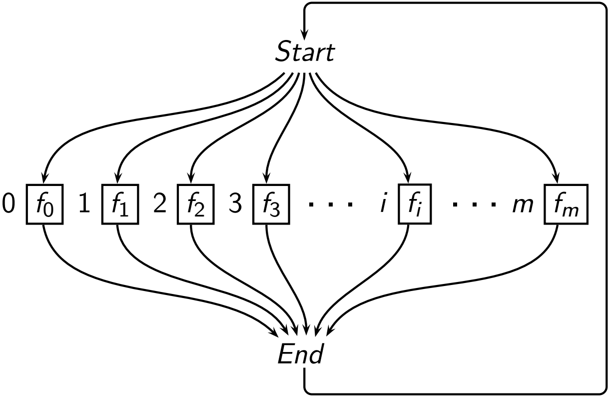




Flow Insensitivity in Data Flow Analysis

If $DepGen_n(x) \neq \emptyset$

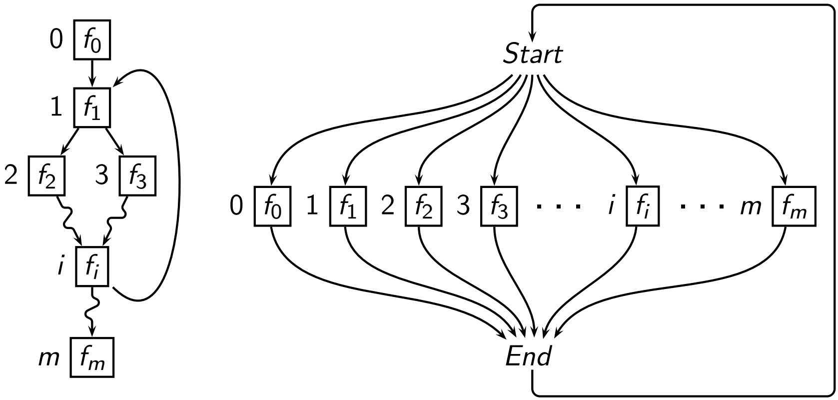




Allows arbitrary compositions of flow functions in any order $\Rightarrow$ *Flow insensitivity*

Flow Insensitivity in Data Flow Analysis

If $DepGen_n(\mathrm{x}) \neq \emptyset$



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

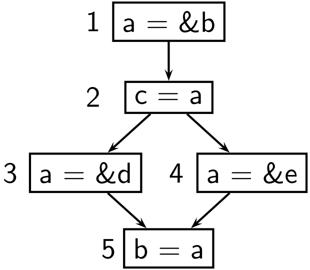
Example of Flow Insensitive Analysis

Flow insensitive points-to analysis
$\Rightarrow$ Same points-to information at each program point

Example of Flow Insensitive Analysis

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

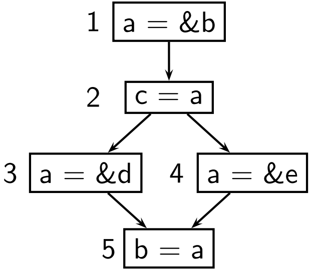
Program



Example of Flow Insensitive Analysis

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

Program



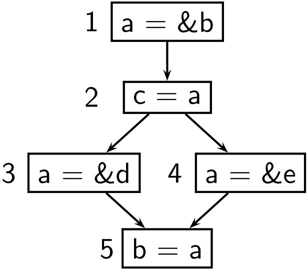
Constraints

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

Example of Flow Insensitive Analysis

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

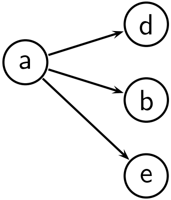
Program



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	$P_b \supseteq P_a$

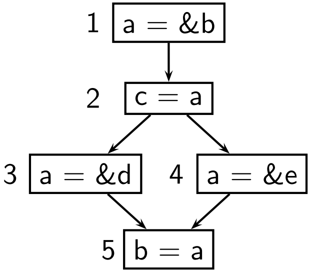
Points-to Graph



Example of Flow Insensitive Analysis

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

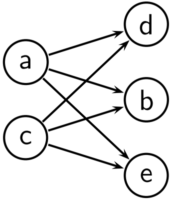
Program



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	$P_b \supseteq P_a$

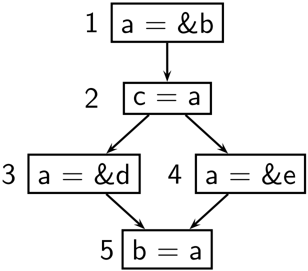
Points-to Graph



Example of Flow Insensitive Analysis

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

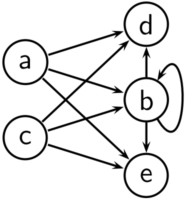
Program



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	$P_b \supseteq P_a$

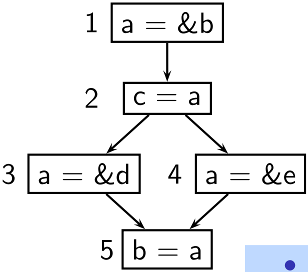
Points-to Graph



Example of Flow Insensitive Analysis

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

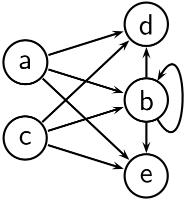
Program



Constraints

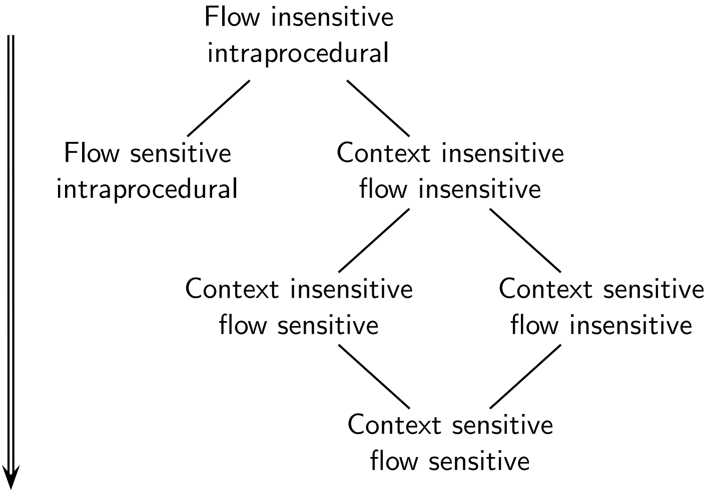
Node	Constraint
1	$P_a \supseteq \{b\}$
2	$P_c \supseteq P_a$
3	$P_a \supseteq \{d\}$
4	$P_a \supseteq \{e\}$
5	$P_b \supseteq P_a$

Points-to Graph

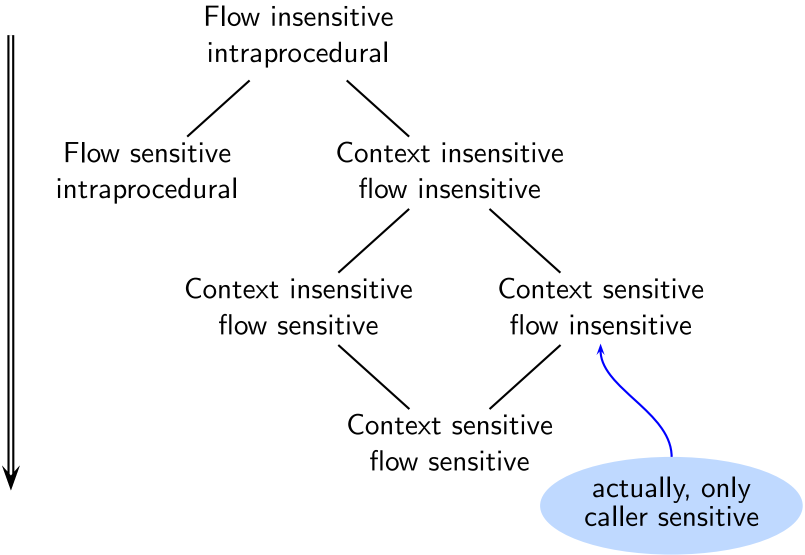


- c does not point to any location in block 1
- a does not point b in block 5
- b does not point to itself at any time

Increasing Precision in Data Flow Analysis



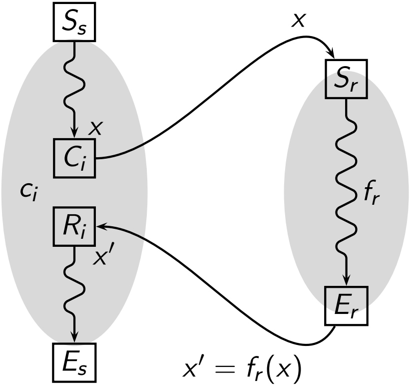
Increasing Precision in Data Flow Analysis



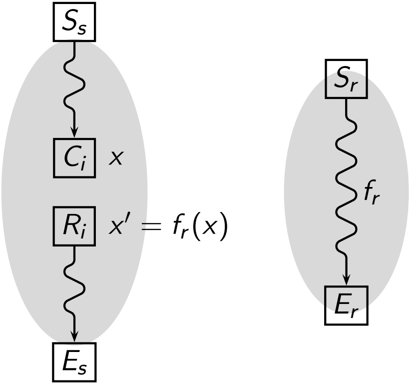
Part 4

Classical Functional Approach

Functional Approach



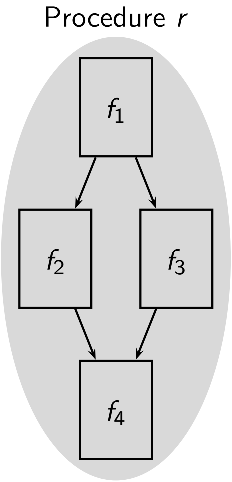
Functional Approach



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

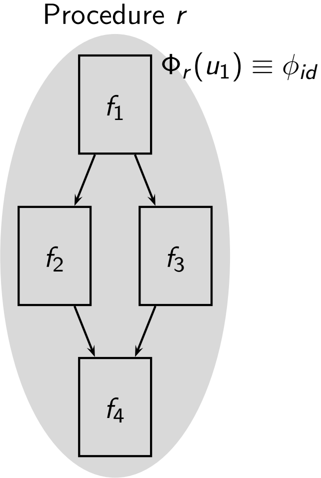
Notation for Summary Flow Function

For simplicity forward flow is assumed.



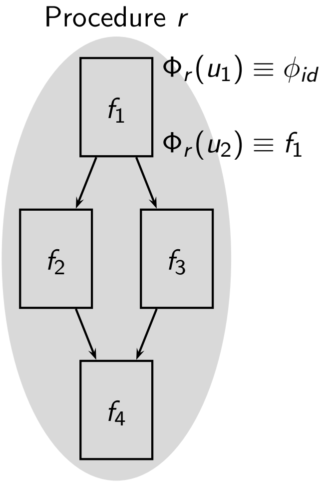
Notation for Summary Flow Function

For simplicity forward flow is assumed.



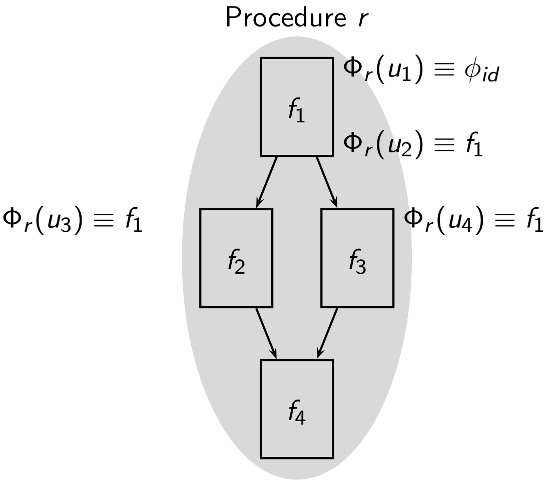
Notation for Summary Flow Function

For simplicity forward flow is assumed.



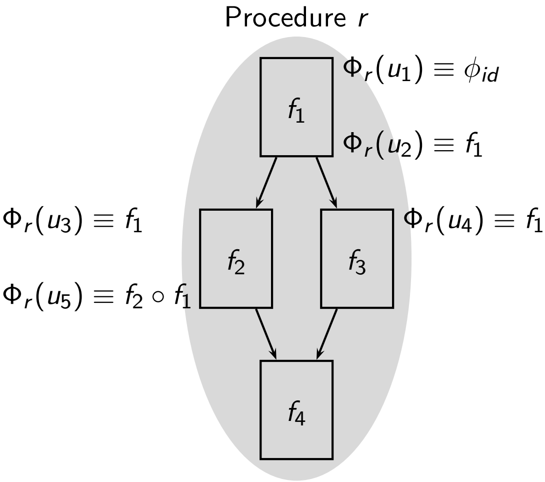
Notation for Summary Flow Function

For simplicity forward flow is assumed.



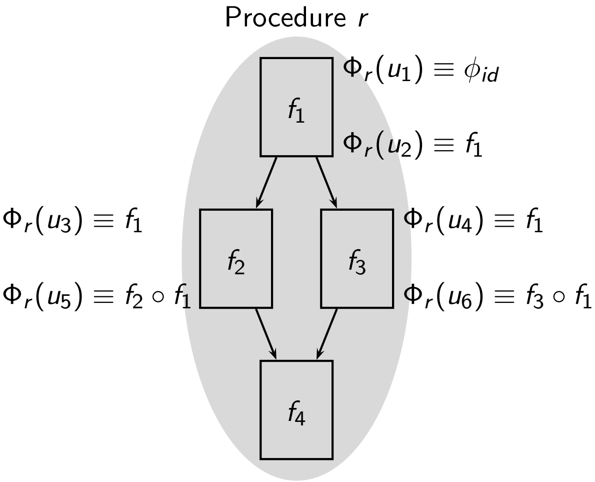
Notation for Summary Flow Function

For simplicity forward flow is assumed.



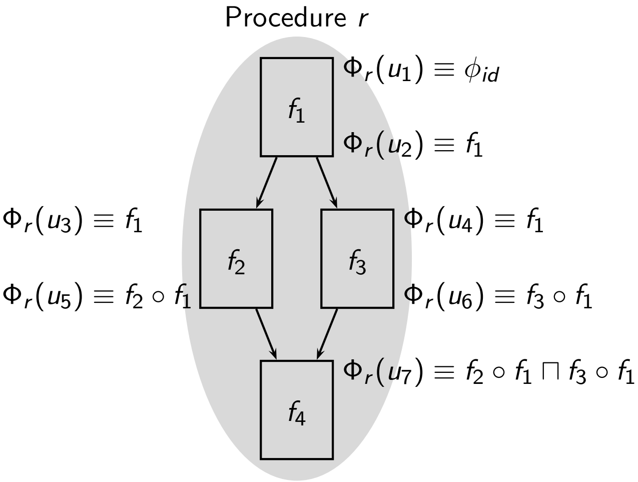
Notation for Summary Flow Function

For simplicity forward flow is assumed.



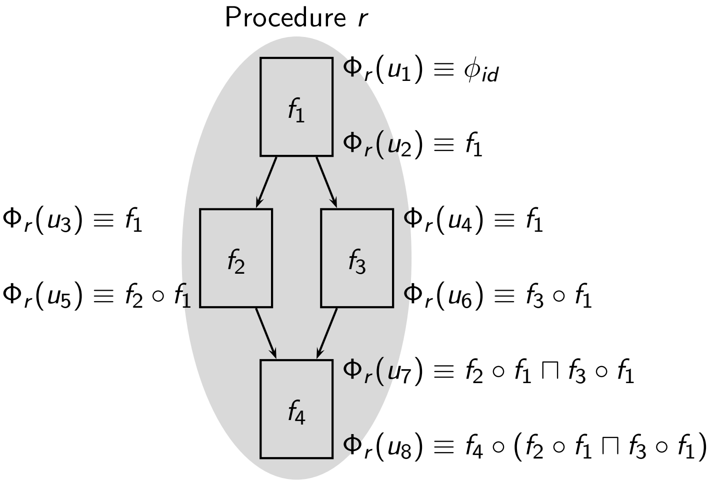
Notation for Summary Flow Function

For simplicity forward flow is assumed.



Notation for Summary Flow Function

For simplicity forward flow is assumed.



Reducing Flow Compositions and Meets

$$f_2 \circ f_1 = f_3 \quad \Leftrightarrow \quad \forall x \in L,\ f_2(f_1(x)) = f_3(x)$$
$$f_2 \sqcap f_1 = f_3 \quad \Leftrightarrow \quad \forall x \in L,\ f_2(x) \sqcap f_1(x) = f_3(x)$$

Reducing Function Compositions

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

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

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 $\cup$,

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

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 $\mathit{ConstKill}_n$ and Gen_n is $\mathit{ConstGen}_n$.

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

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

Hence

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

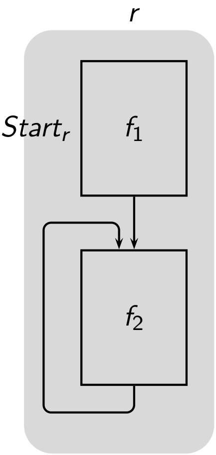
Constructing Summary Flow Function

For simplicity forward flow is assumed.

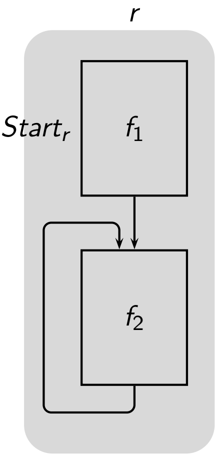
$$\Phi_r(Entry(n)) = \begin{cases} \phi_{id} & \text{if } n \text{ is } Start_r \\ \displaystyle\bigsqcap_{p \in pred(n)} \Big(\Phi_r(Exit(p))\Big) & \text{otherwise} \end{cases}$$

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

Constructing Summary Flow Functions



Constructing Summary Flow Functions



Iteration #1

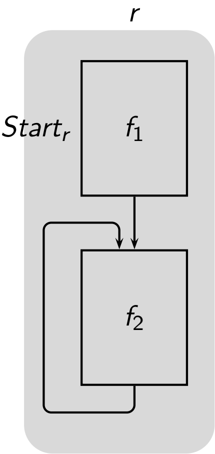
$\Phi_r(u_1) = \phi_{id}$

$\Phi_r(u_2) = f_1$

$\Phi_r(u_3) = f_1$

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

Constructing Summary Flow Functions



Iteration #2

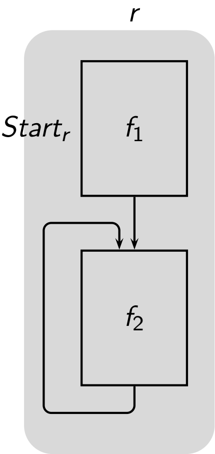
$\Phi_r(u_1) = \phi_{id}$

$\Phi_r(u_2) = f_1$

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

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

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

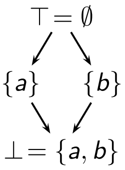
Component functions (i.e. for a single variable)

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

Lattice of Flow Functions for Live Variables Analysis

Flow functions for two variables

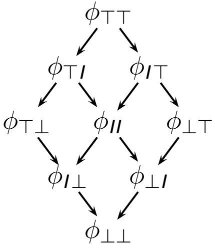
Lattice of data flow values



All possible flow functions

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

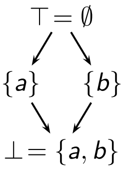
Lattice of flow functions



Lattice of Flow Functions for Live Variables Analysis

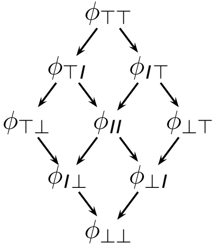
Flow functions for two variables

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

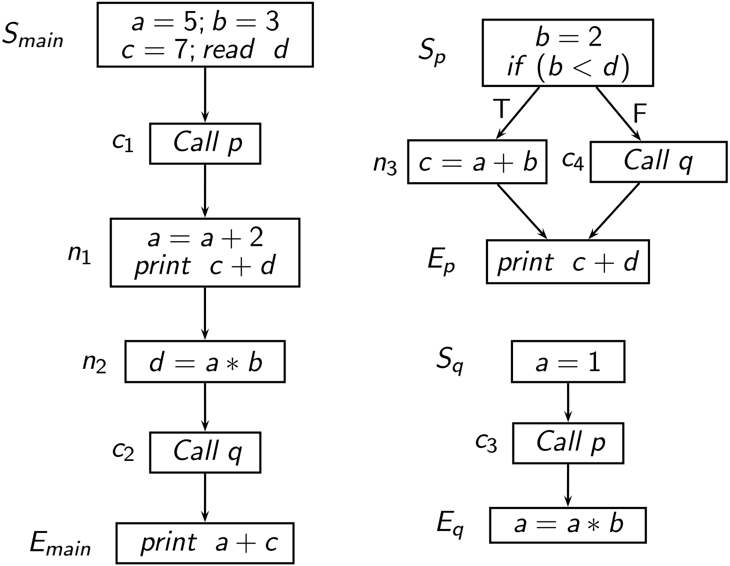


Gen_n	$Kill_n$	f_n	Gen_n	$Kill_n$	f_n
$\emptyset$	$\emptyset$	ϕ_{II}	$\{b\}$	$\emptyset$	$\phi_{I\bot}$
$\emptyset$	$\{a\}$	[illegible]	[illegible]	[illegible]	$\phi_{\top\bot}$
$\emptyset$	$\{b\}$	[illegible]	[illegible]	[illegible]	$\phi_{I\bot}$
$\emptyset$	$\{a, b\}$	[illegible]	[illegible]	[illegible]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

Essentially, a product lattice of the two component lattices



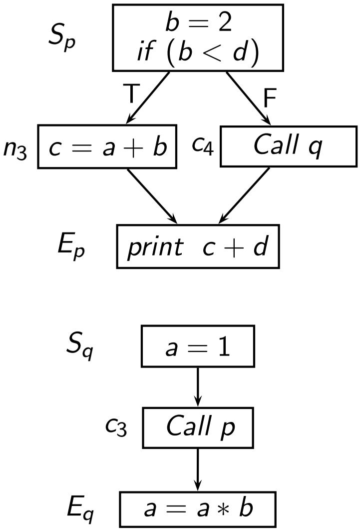
An Example of Interprocedural Liveness Analysis



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}	$\{c, d\}$	$\emptyset$		
	$\Phi_p(n_3)$	$f_{n_3} \circ \Phi_p(E_p)$	$\{a, b, d\}$	$\{c\}$		
	$\Phi_p(c_4)$	$f_q \circ \Phi_p(E_p) = \phi_\top$	$\emptyset$	$\{a, b, c, d\}$	$\{d\}$	$\{a, b, c\}$
	$\Phi_p(S_p)$	$f_{S_p} \circ \left(\Phi_p(n_3) \sqcap \Phi_p(c_4)\right)$	$\{a, d\}$	$\{b, c\}$		
	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	
$\Phi_p(E_p)$	$BI_p \cup \{c, d\}$
$\Phi_p(n_3)$	$\left(BI_p - \{c\}\right) \cup \{a, b, d\}$
$\Phi_p(c_4)$	$\left(BI_p - \{a, b, c\}\right) \cup \{d\}$
$\Phi_p(S_p)$	$\left(BI_p - \{b, c\}\right) \cup \{a, d\}$
$\Phi_q(E_q)$	$\left(BI_q - \{a\}\right) \cup \{a, b\}$
$\Phi_q(c_3)$	$\left(BI_q - \{a, b, c\}\right) \cup \{a, d\}$
$\Phi_q(S_q)$	$\left(BI_q - \{a, b, c\}\right) \cup \{d\}$

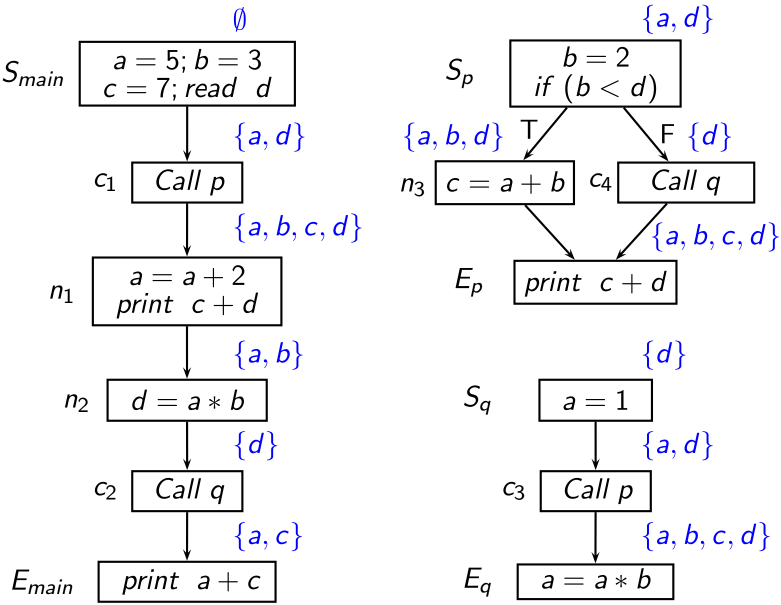
Result of Interprocedural Liveness Analysis

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}	$\Phi_m(c_2)$	$\left(BI_m - \{a, b, c\}\right) \cup \{d\}$	$\{d\}$
In_{n_2}	$\Phi_m(n_2)$	$\left(BI_m - \{a, b, c, d\}\right) \cup \{a, b\}$	$\{a, b\}$
In_{n_1}	$\Phi_m(n_1)$	$\left(BI_m - \{a, b, c, d\}\right) \cup \{a, b, c, d\}$	$\{a, b, c, d\}$
In_{c_1}	$\Phi_m(c_1)$	$\left(BI_m - \{a, b, c, d\}\right) \cup \{a, d\}$	$\{a, d\}$
In_{S_m}	$\Phi_m(S_m)$	$BI_m - \{a, b, c, d\}$	$\emptyset$

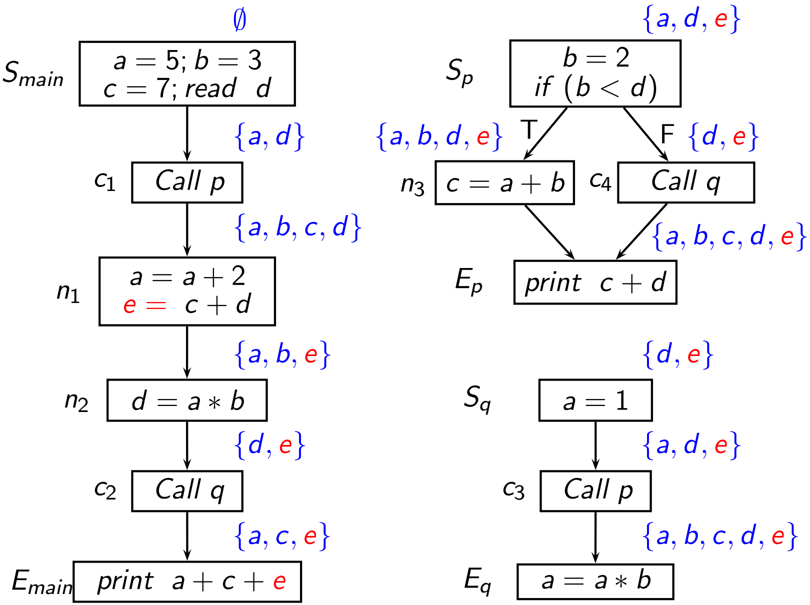
Result of Interprocedural Liveness Analysis

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

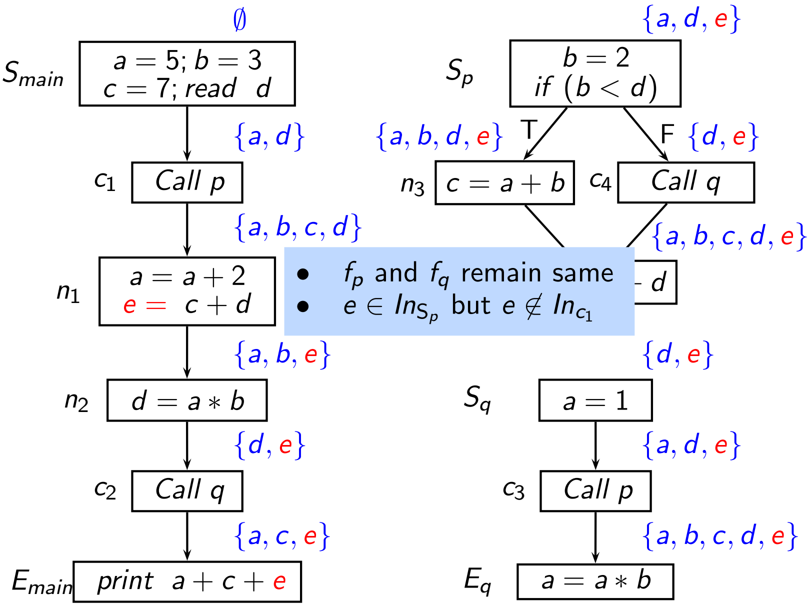
Result of Interprocedural Liveness Analysis



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

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

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

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

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 $\{\langle \sigma, \text{x} \rangle \mid \sigma \text{ is a call string }, \text{x} \in L\}$
  - The final data flow information is

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

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

- 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

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

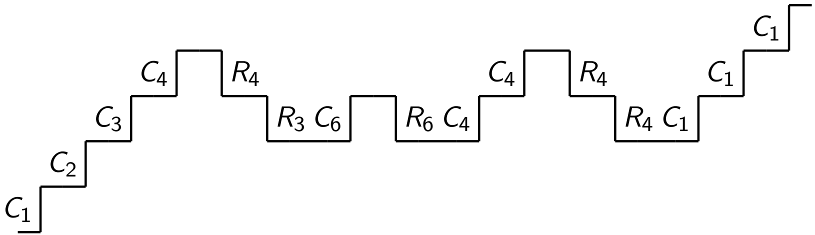
$$\text{OUT}_n = DepGEN_n$$

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

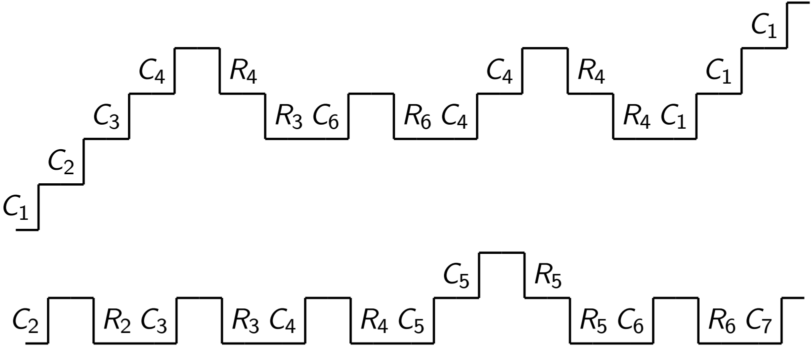
$$\begin{aligned} X \uplus Y = & \left\{ \langle \sigma, \text{x} \sqcap \text{y} \rangle \mid \langle \sigma, \text{x} \rangle \in X,\ \langle \sigma, \text{y} \rangle \in Y \right\} \cup \\ & \left\{ \langle \sigma, \text{x} \rangle \mid \langle \sigma, \text{x} \rangle \in X,\ \forall \text{z} \in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

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

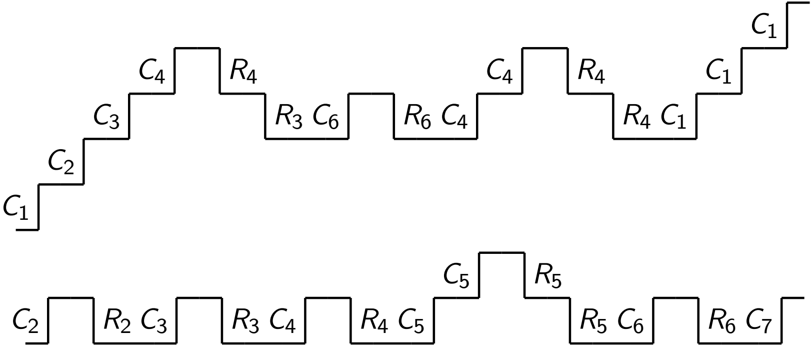
Interprocedural Validity and Calling Contexts



Interprocedural Validity and Calling Contexts

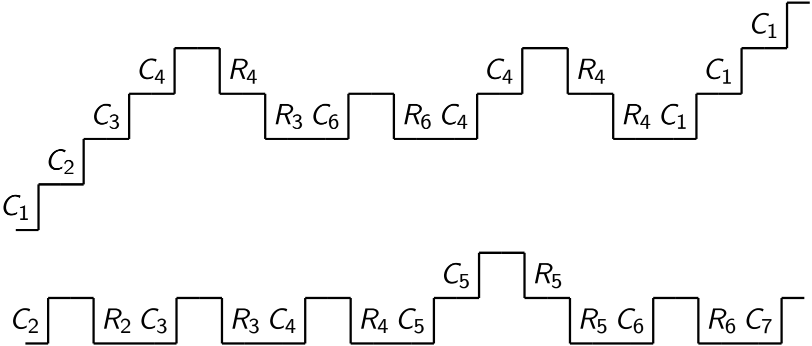


Interprocedural Validity and Calling Contexts



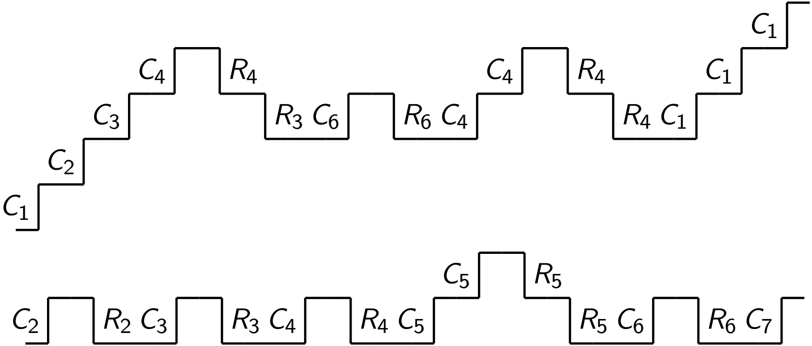
- "You can descend only as much as you have ascended!"

Interprocedural Validity and Calling Contexts



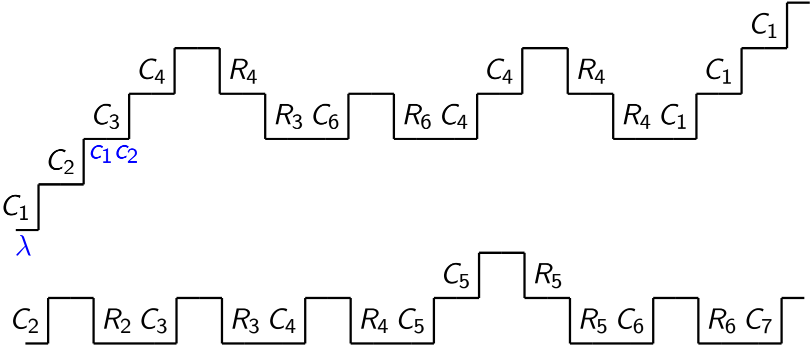
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step.

Interprocedural Validity and Calling Contexts



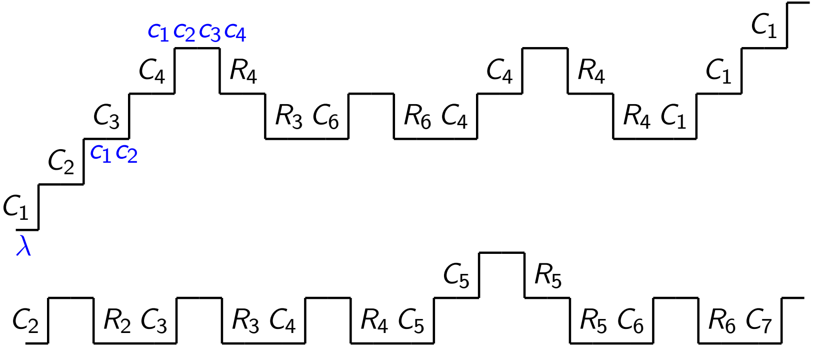
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step.
- Calling context is represented by the remaining descending steps.

Interprocedural Validity and Calling Contexts



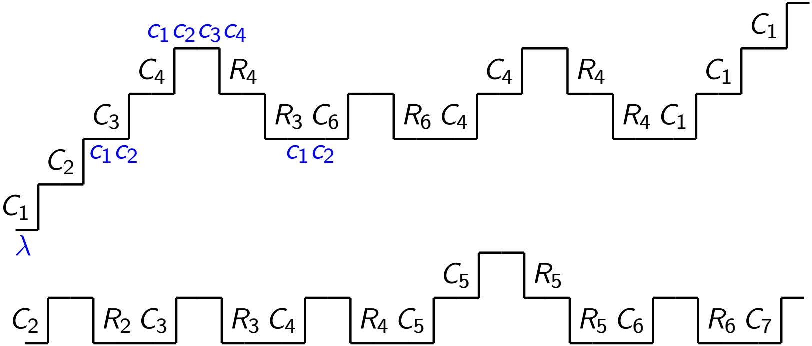
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step.
- Calling context is represented by the remaining descending steps.

Interprocedural Validity and Calling Contexts



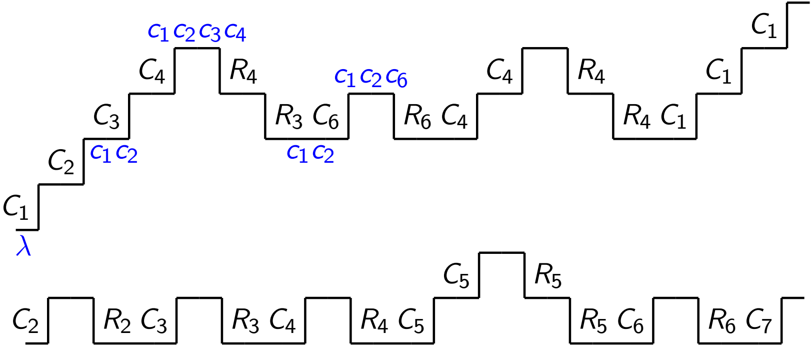
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step.
- Calling context is represented by the remaining descending steps.

Interprocedural Validity and Calling Contexts



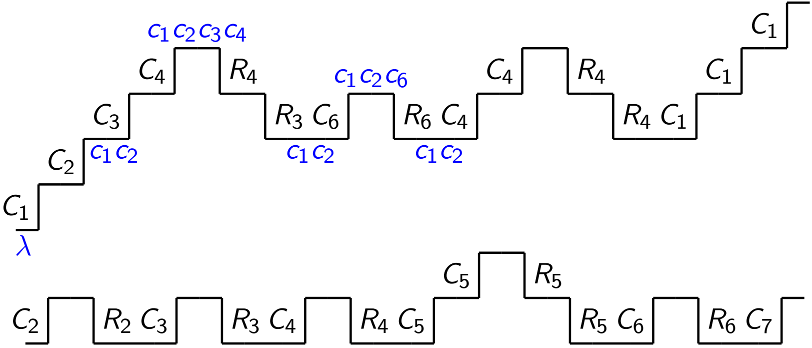
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step.
- Calling context is represented by the remaining descending steps.

Interprocedural Validity and Calling Contexts



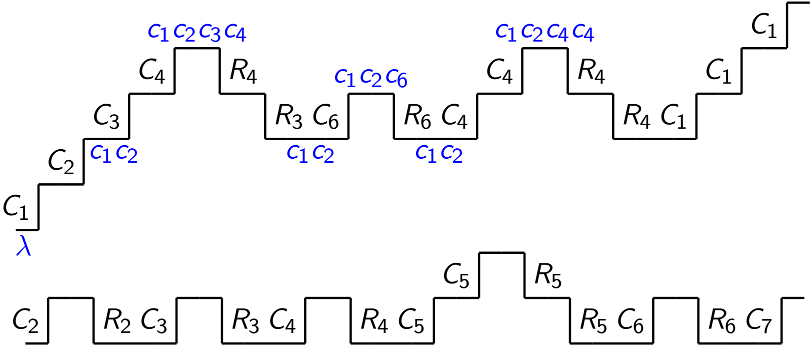
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step.
- Calling context is represented by the remaining descending steps.

Interprocedural Validity and Calling Contexts



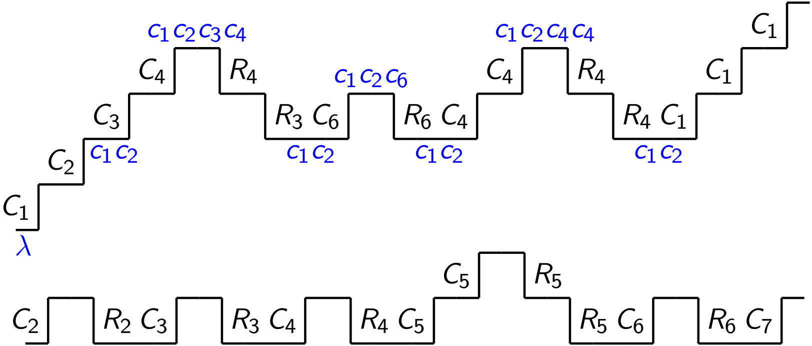
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step.
- Calling context is represented by the remaining descending steps.

Interprocedural Validity and Calling Contexts



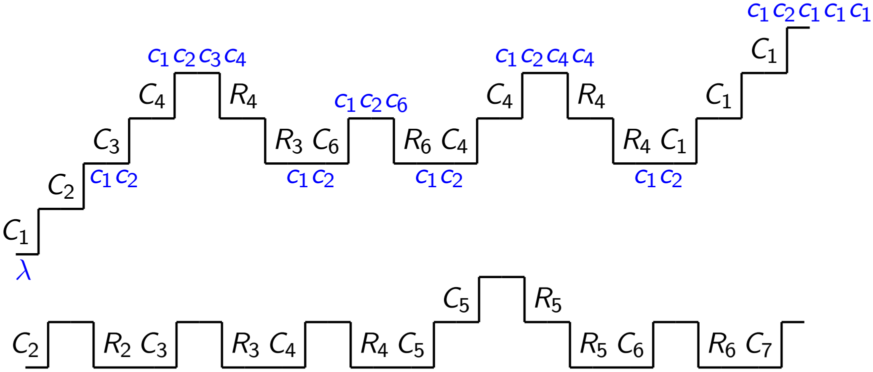
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step.
- Calling context is represented by the remaining descending steps.

Interprocedural Validity and Calling Contexts



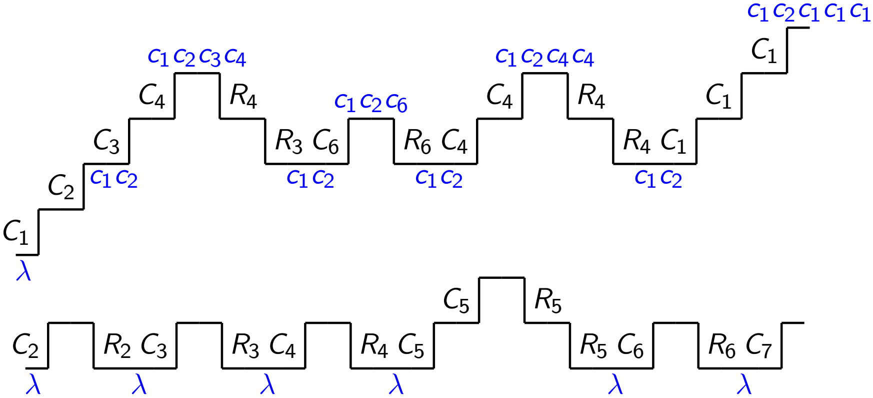
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step.
- Calling context is represented by the remaining descending steps.

Interprocedural Validity and Calling Contexts



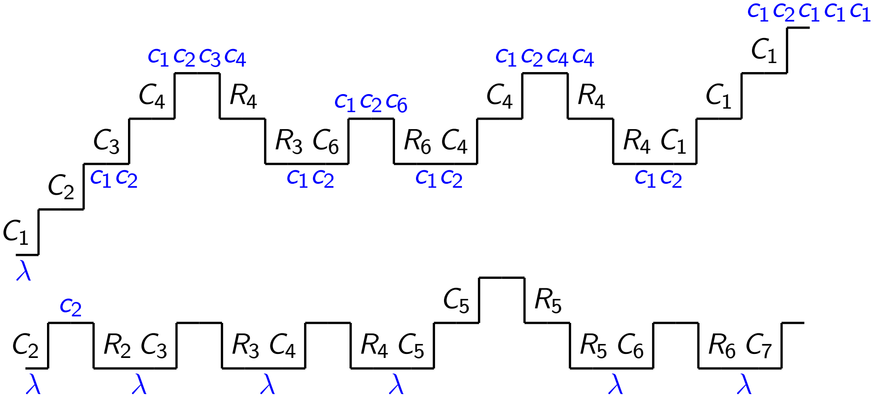
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step.
- Calling context is represented by the remaining descending steps.

Interprocedural Validity and Calling Contexts



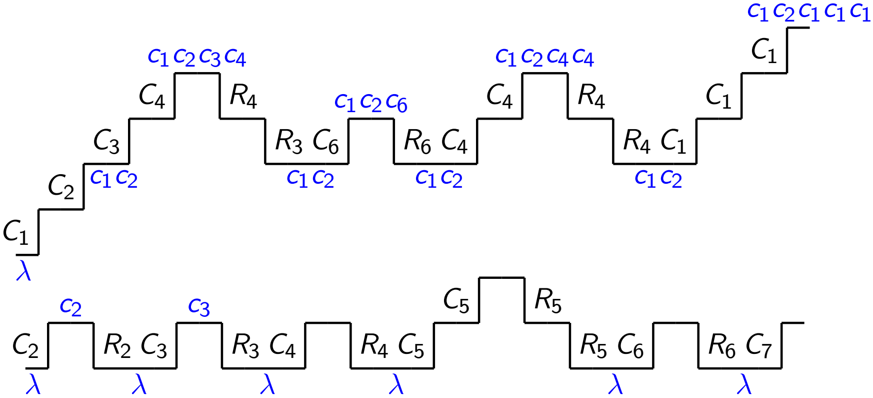
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step.
- Calling context is represented by the remaining descending steps.

Interprocedural Validity and Calling Contexts



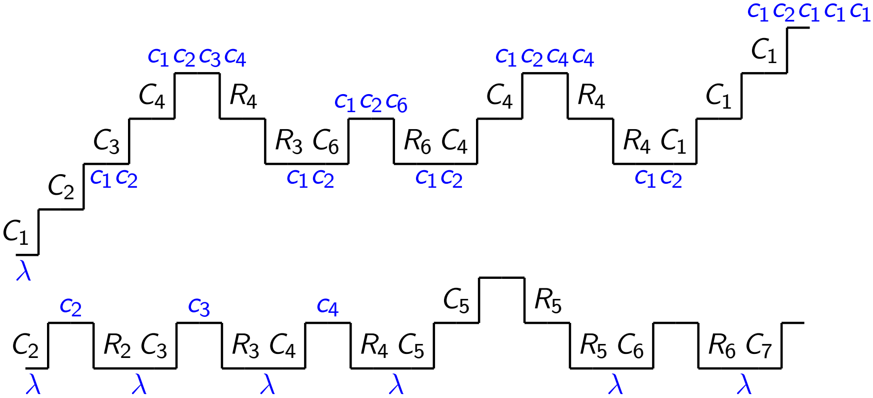
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step.
- Calling context is represented by the remaining descending steps.

Interprocedural Validity and Calling Contexts



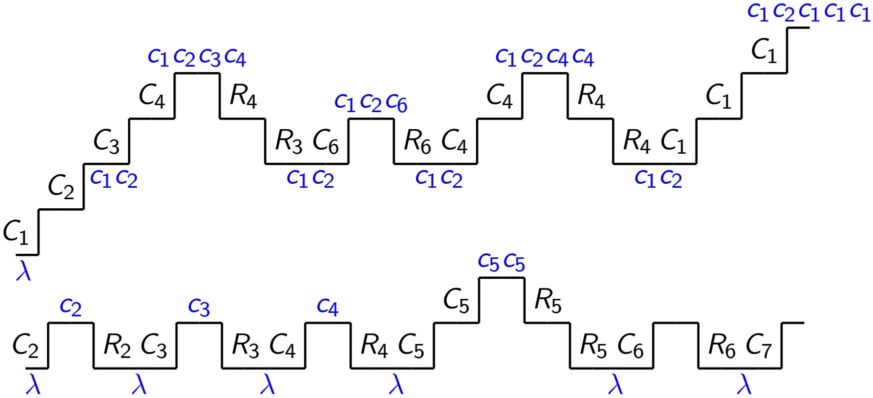
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step.
- Calling context is represented by the remaining descending steps.

Interprocedural Validity and Calling Contexts



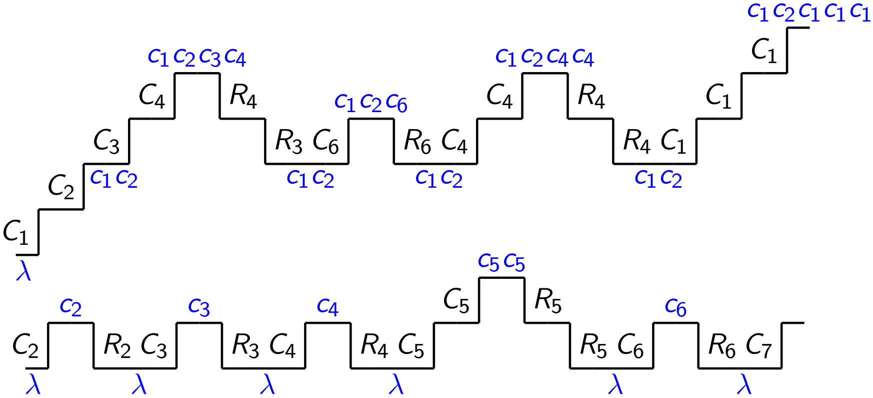
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step.
- Calling context is represented by the remaining descending steps.

Interprocedural Validity and Calling Contexts



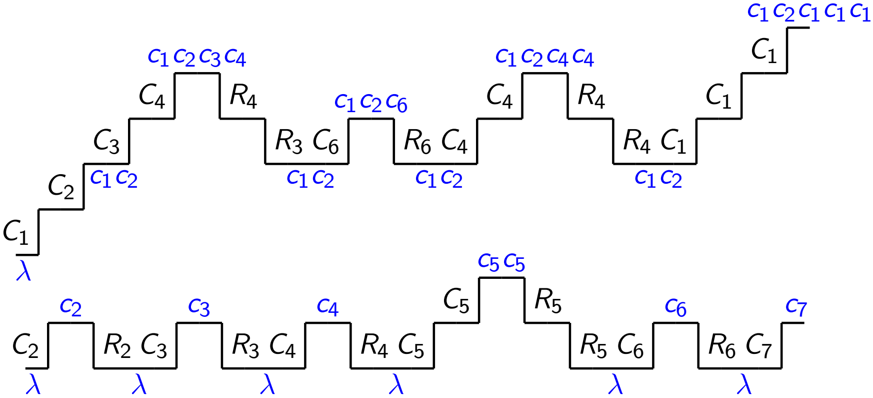
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step.
- Calling context is represented by the remaining descending steps.

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.

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.

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.
- 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.

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

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.

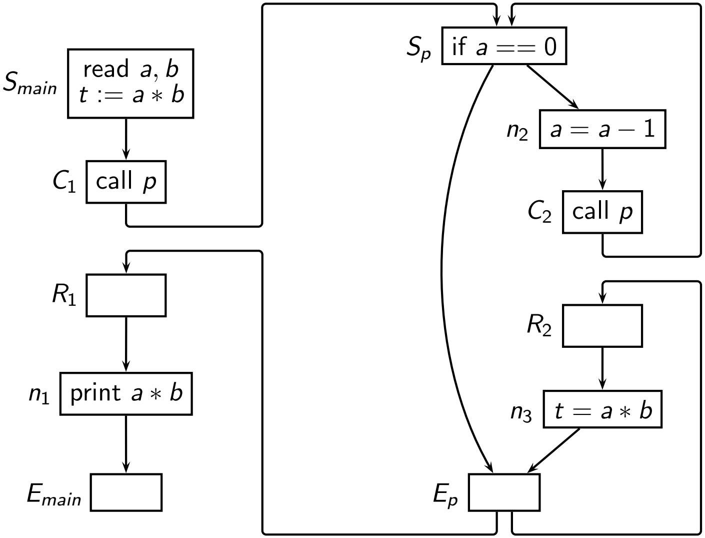
Ascend

- 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.

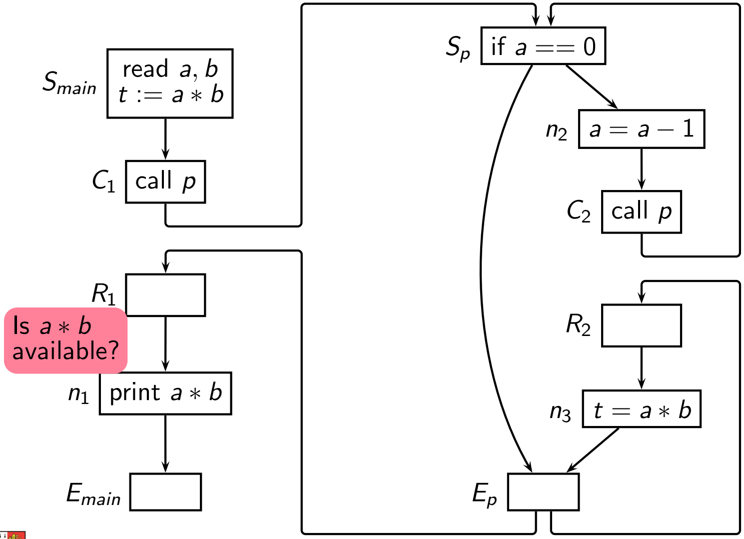
Descend

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

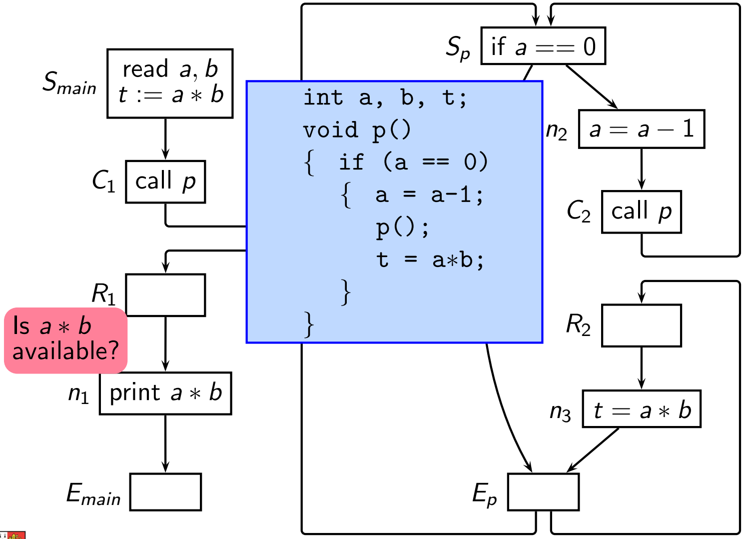
Available Expressions Analysis Using Call Strings Approach



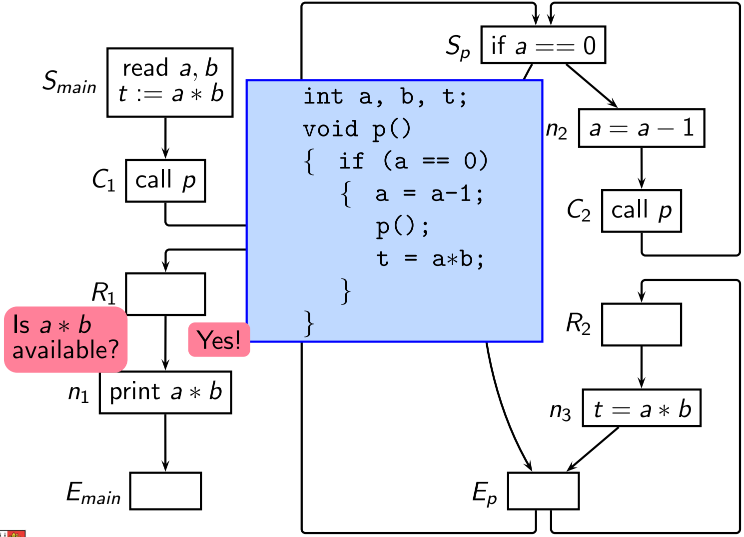
Available Expressions Analysis Using Call Strings Approach



Available Expressions Analysis Using Call Strings Approach

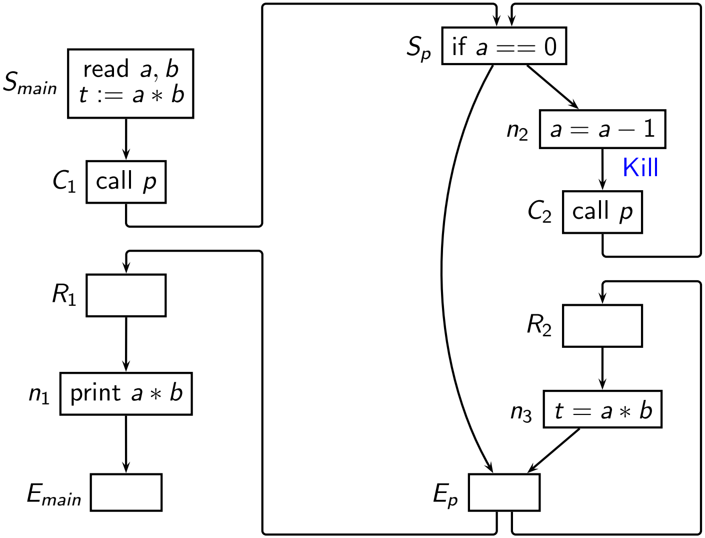


Available Expressions Analysis Using Call Strings Approach

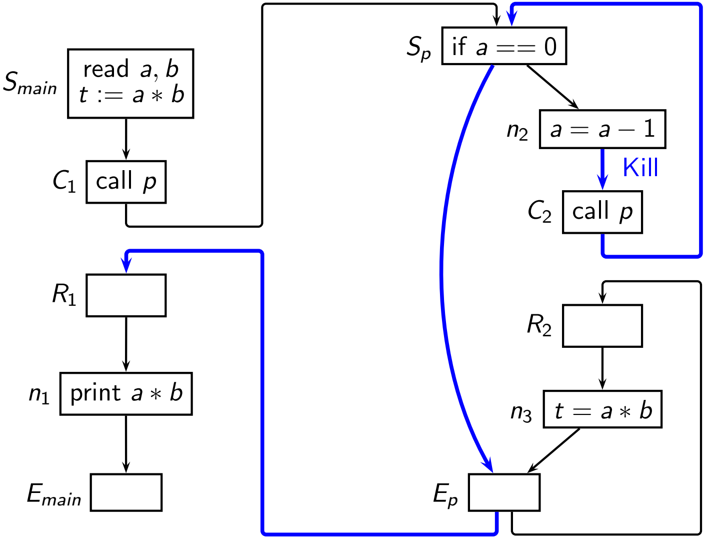



```
int a, b, t;
void p()
{  if (a == 0)
   {  a = a-1;
      p();
      t = a*b;
   }
}
```

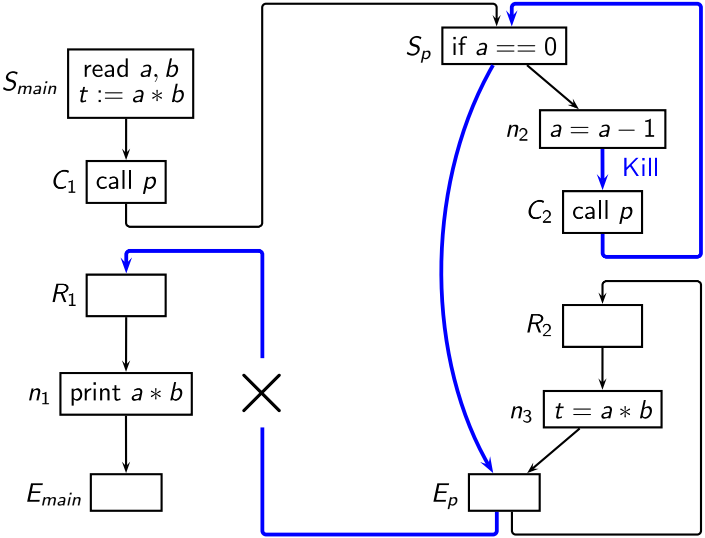
Available Expressions Analysis Using Call Strings Approach



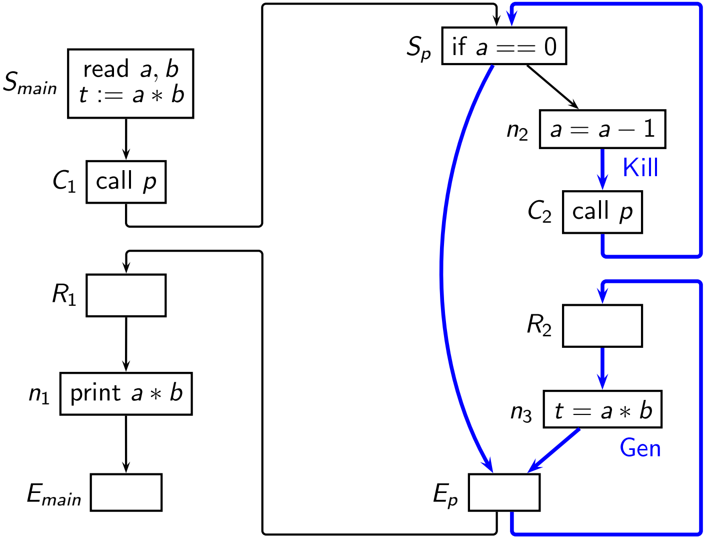
Available Expressions Analysis Using Call Strings Approach



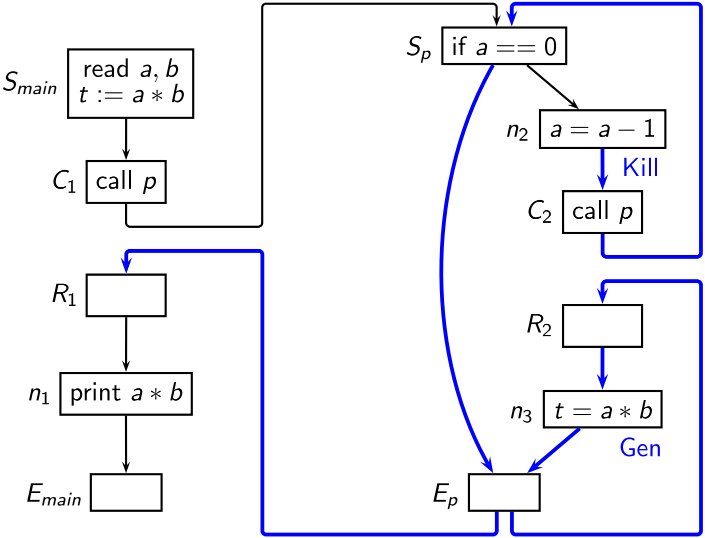
Available Expressions Analysis Using Call Strings Approach



Available Expressions Analysis Using Call Strings Approach

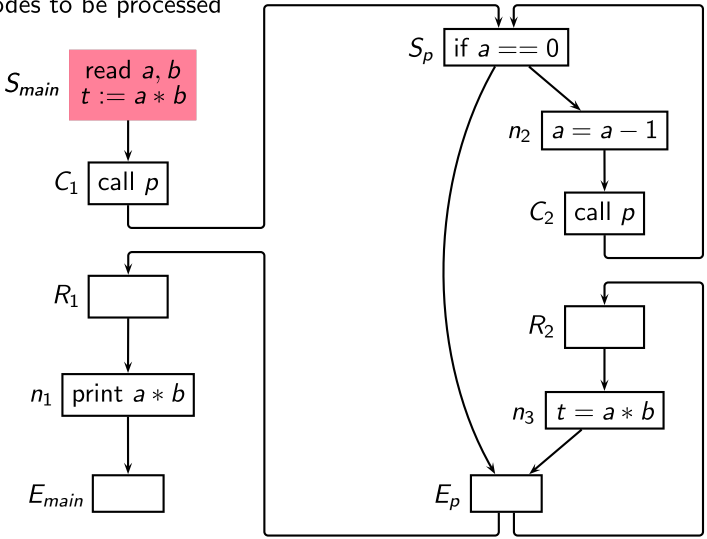


Available Expressions Analysis Using Call Strings Approach



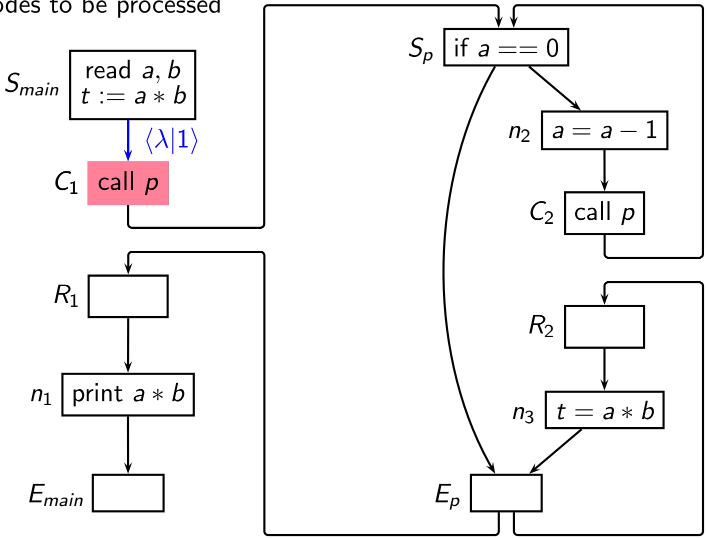
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed



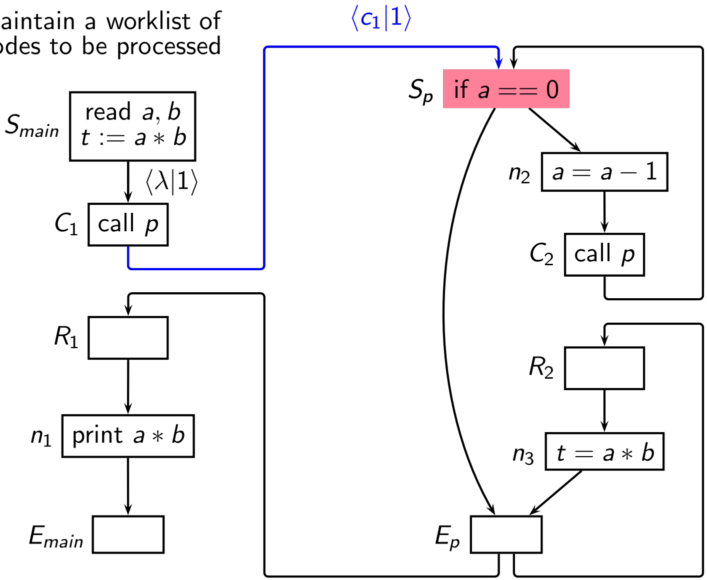
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed



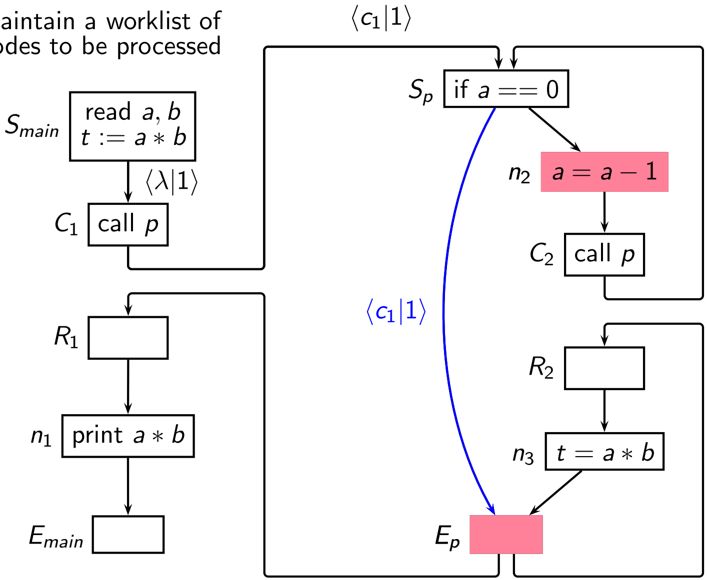
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed



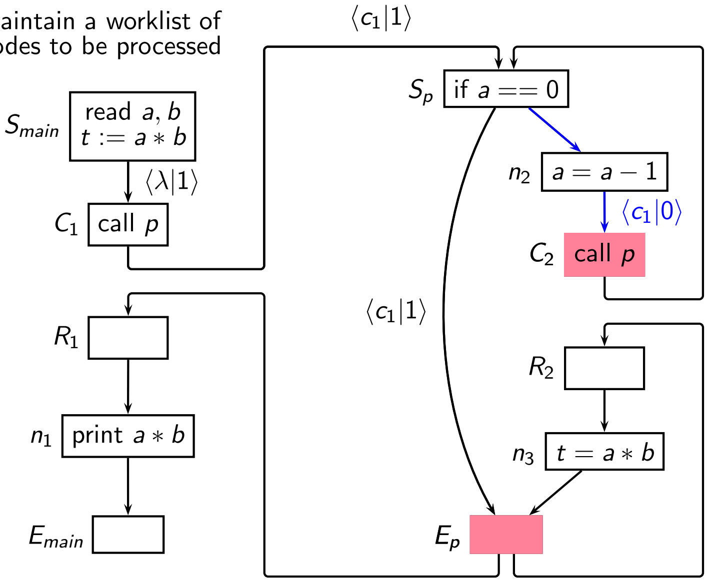
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed



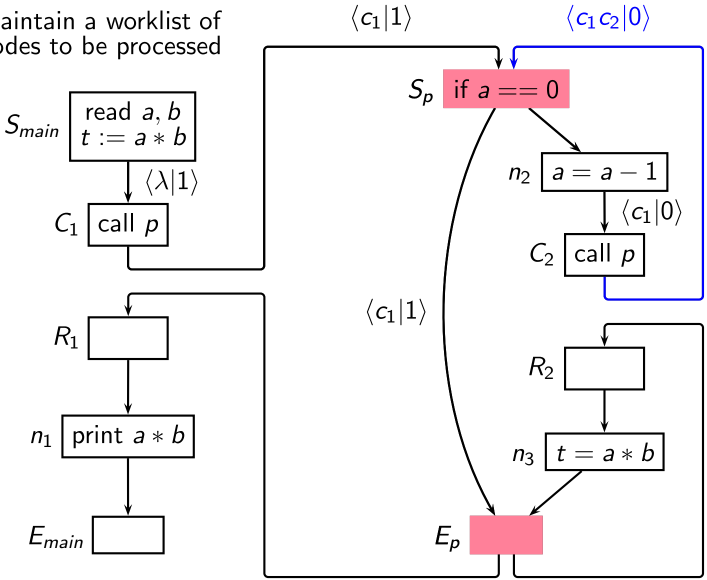
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed



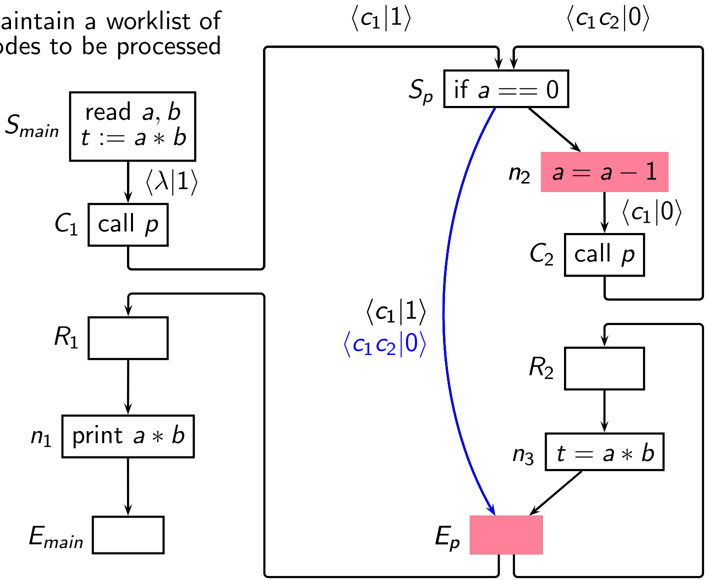
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed



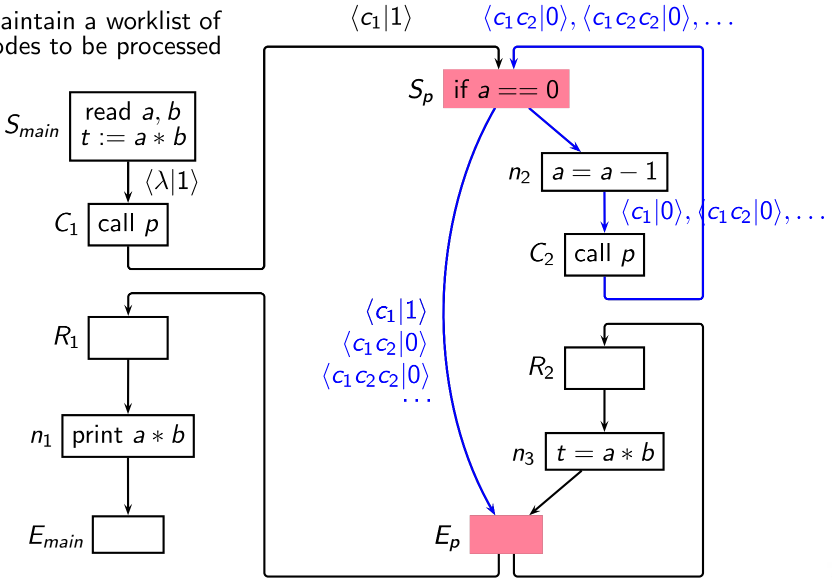
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed



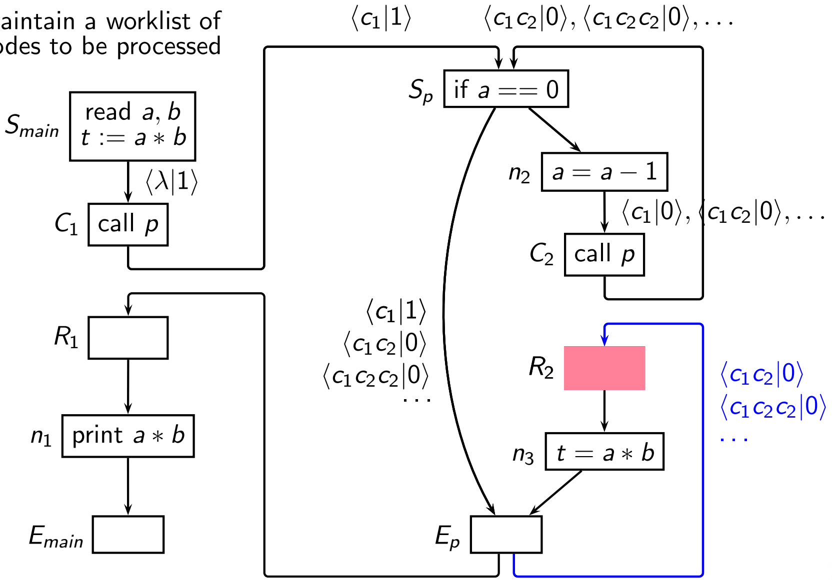
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed



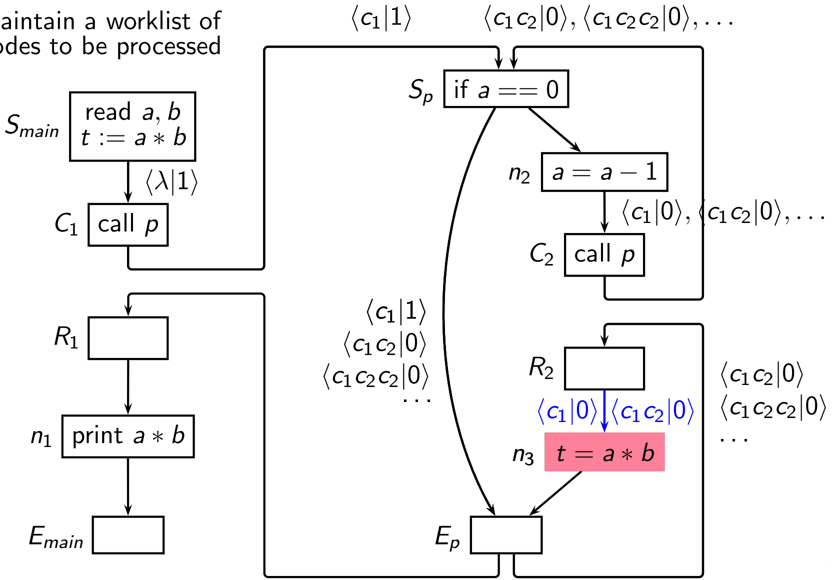
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed



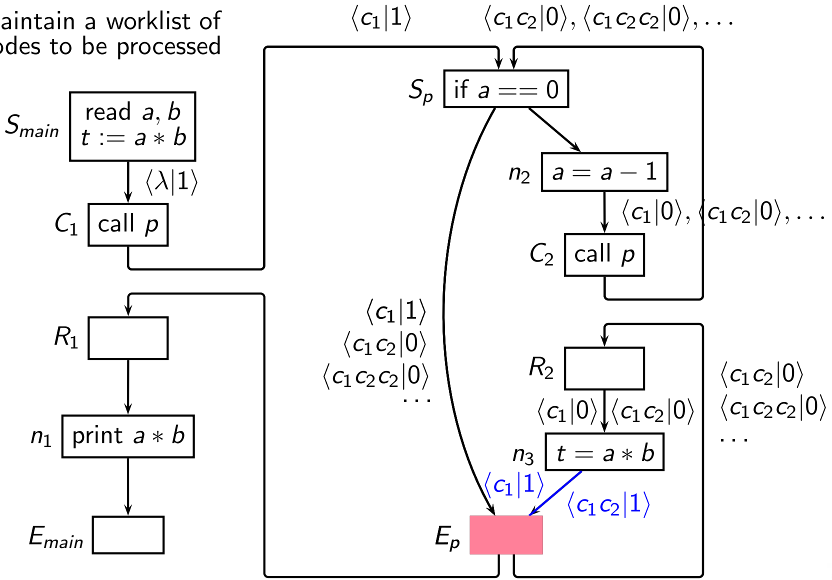
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed



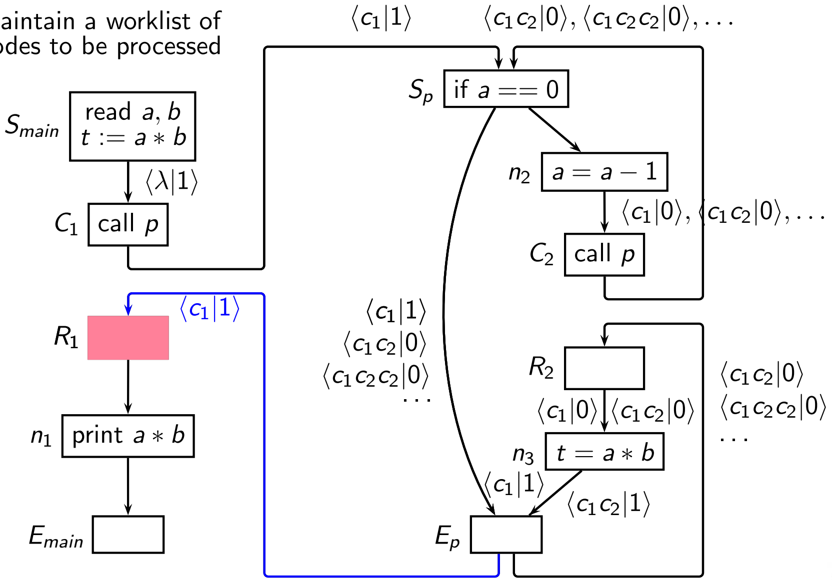
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed



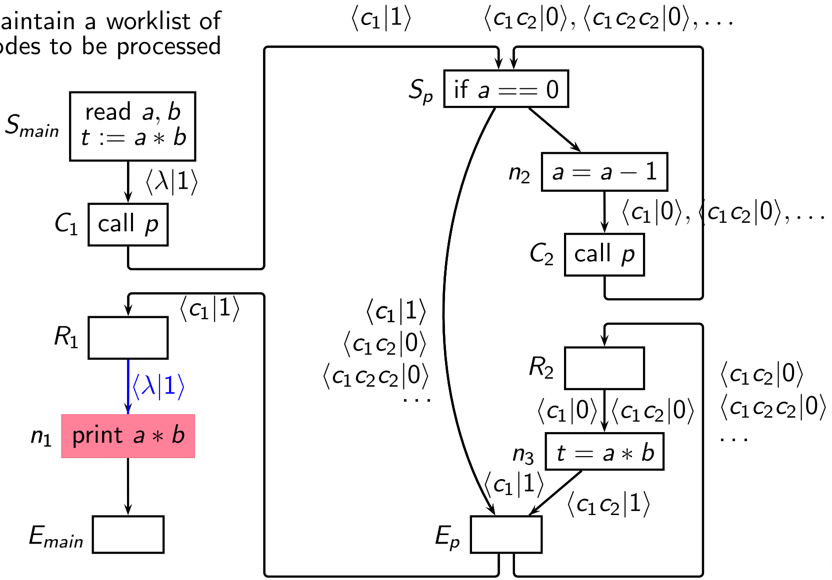
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed



Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed



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

```
int a,b,c;
void main()
{   c = a*b;
    p();
}
void p()
{ if (...)
  { p();
  Is a*b available?
      a = a*b;
  }
}
```

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

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

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

Path 2

3 : Gen
4
7
8
7
8
7
12
9
10 : Kill
11
12
9
10 : Kill

The Need for Multiple Occurrences of a Call Site

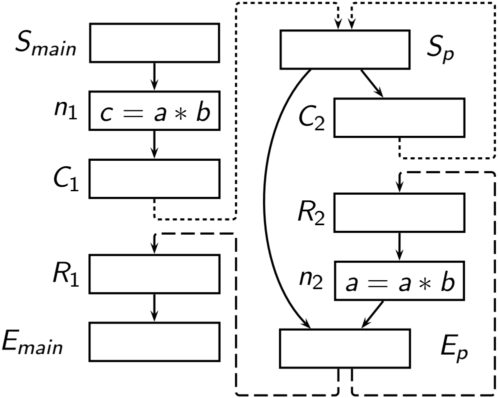
Even if data flow values in cyclic call sequence do not change

```
int a,b,c;
void main()
{   c = a*b;
    p();
}
void p()
{  if (...)
   {  p();
   Is a*b available?
      a = a*b;
   }
}
```

The Need for Multiple Occurrences of a Call Site

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

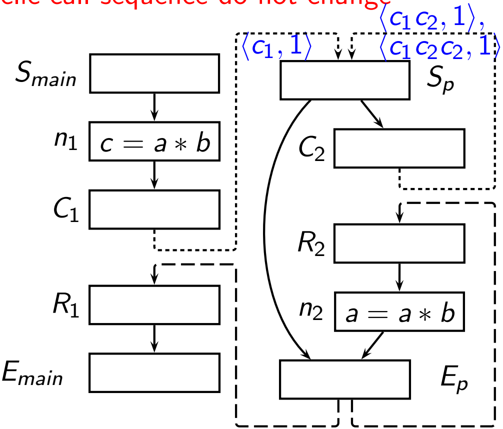
```
int a,b,c;
void main()
{   c = a*b;
    p();
}
void p()
{ if (...)
   { p();
   Is a*b available?
      a = a*b;
   }
}
```



The Need for Multiple Occurrences of a Call Site

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

```
int a,b,c;
void main()
{   c = a*b;
    p();
}
void p()
{ if (...)
   { p();
   Is a*b available?
      a = a*b;
   }
}
```

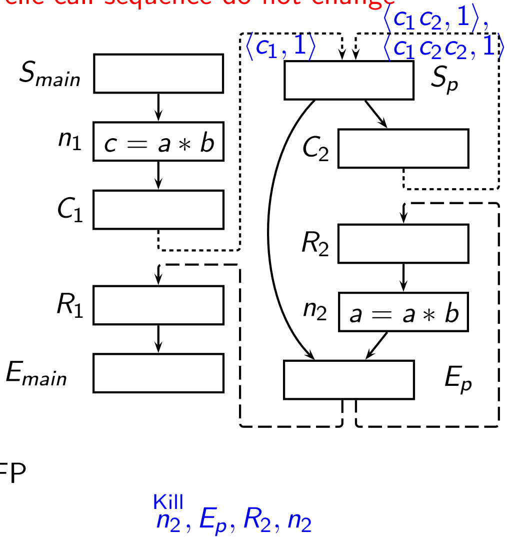


The Need for Multiple Occurrences of a Call Site

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

```
int a,b,c;
void main()
{   c = a*b;
    p();
}
void p()
{ if (...)
  { p();
    Is a*b available?
      a = a*b;
  }
}
```

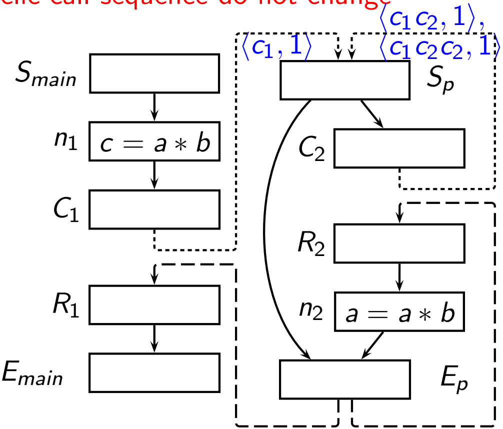
- Interprocedurally valid IFP



The Need for Multiple Occurrences of a Call Site

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

```
int a,b,c;
void main()
{   c = a*b;
    p();
}
void p()
{ if (...)
  { p();
    Is a*b available?
      a = a*b;
  }
}
```



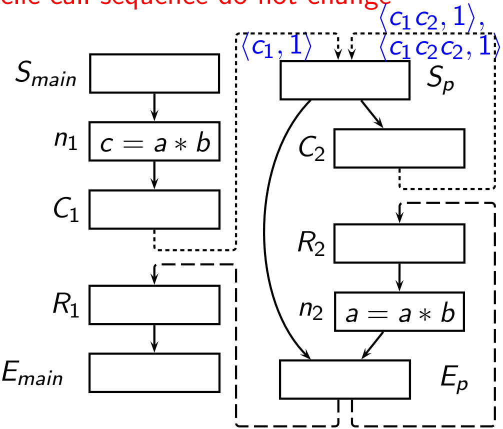
- Interprocedurally valid IFP

$C_2, S_p, E_p, R_2,$ 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

```
int a,b,c;
void main()
{   c = a*b;
    p();
}
void p()
{ if (...)
  { p();
    Is a*b available?
      a = a*b;
  }
}
```



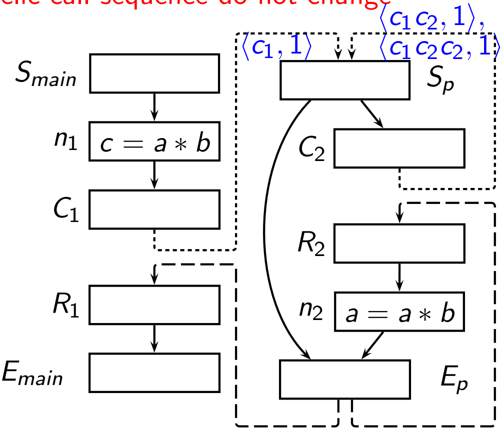
- Interprocedurally valid IFP

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

The Need for Multiple Occurrences of a Call Site

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

```
int a,b,c;
void main()
{   c = a*b;
    p();
}
void p()
{  if (...)
   {  p();
      Is a*b available?
        a = a*b;
   }
}
```



- Interprocedurally valid IFP

$S_m, n_1, C_1, S_p, C_2, S_p, C_2, S_p, E_p, R_2,$ 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

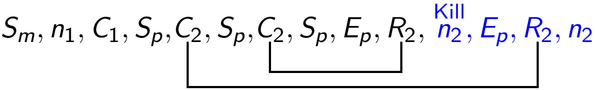
  $S_m, n_1, C_1, S_p, C_2, S_p, C_2, S_p, E_p, R_2, \overset{\text{Kill}}{n_2}, E_p, R_2, n_2$

The Need for Multiple Occurrences of a Call Site

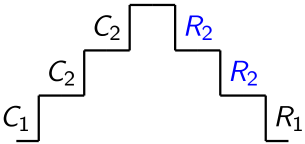
Even if data flow values in cyclic call sequence do not change

In terms of staircase diagram

- Interprocedurally valid IFP



- You cannot descend twice, unless you ascend twice



The Need for Multiple Occurrences of a Call Site

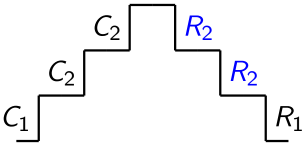
Even if data flow values in cyclic call sequence do not change

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

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.

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

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)

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)

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

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

⇒ Not a bound but prescribed necessary length

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

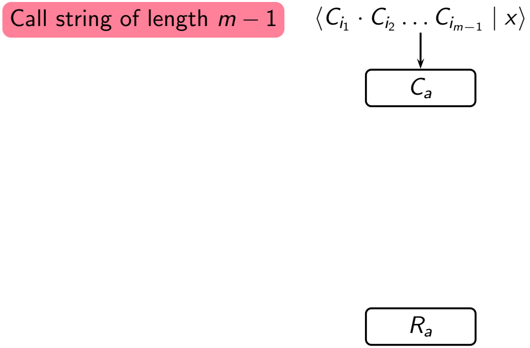
$\Rightarrow$ Large number of long call strings

Classical 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.

Call string of length $m-1$ $\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_{m-1}} \mid x \rangle$

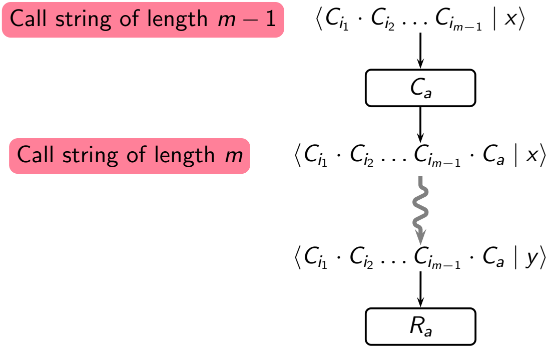
C_a

Call string of length m $\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_{m-1}} \cdot C_a \mid x \rangle$

R_a

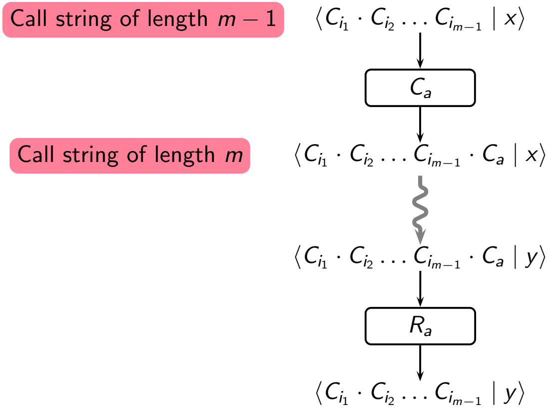
Classical Approximate Approach

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



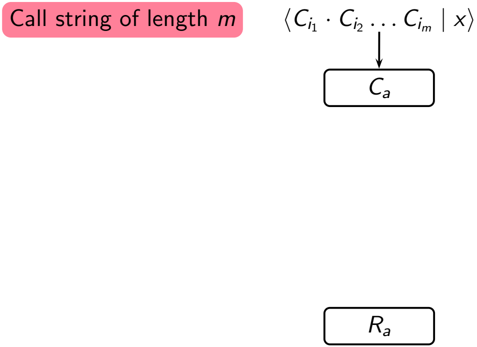
Classical Approximate Approach

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



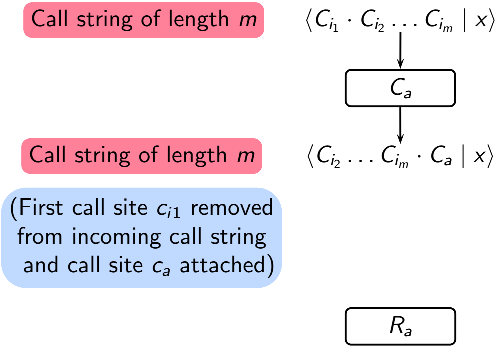
Classical Approximate Approach

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



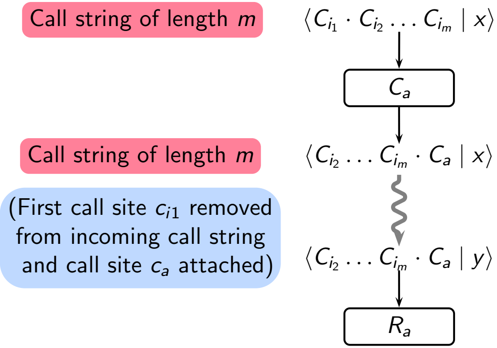
Classical Approximate Approach

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



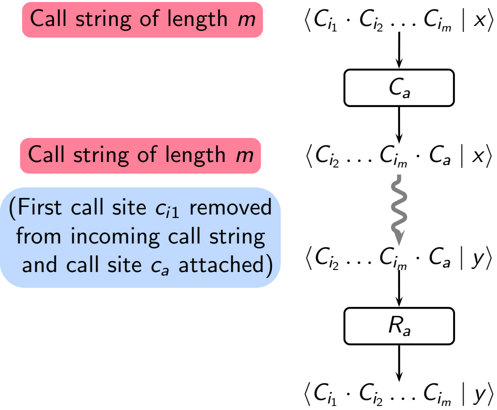
Classical Approximate Approach

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



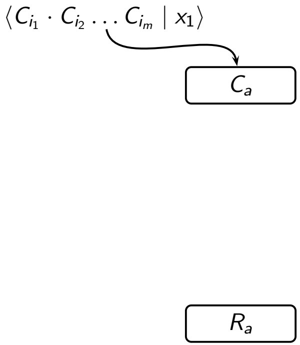
Classical Approximate Approach

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



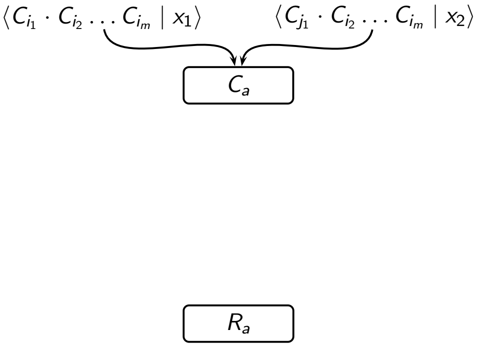
Classical Approximate Approach

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



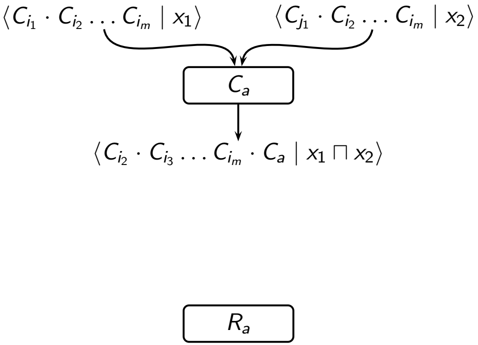
Classical Approximate Approach

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



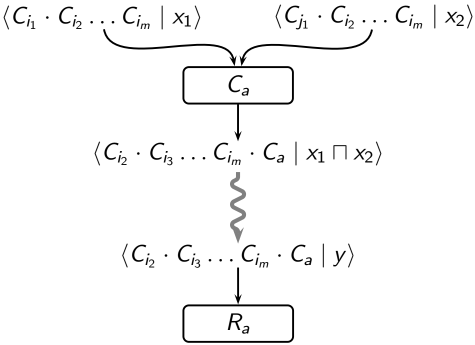
Classical Approximate Approach

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



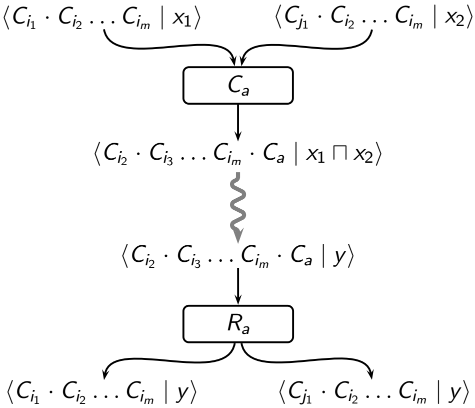
Classical Approximate Approach

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



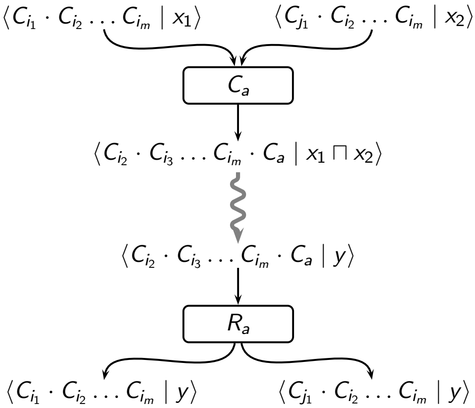
Classical Approximate Approach

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



Classical Approximate Approach

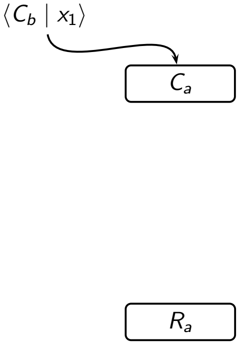
- Maintain call string suffixes of upto a given length m.



- Practical choices of m have been 1 or 2.

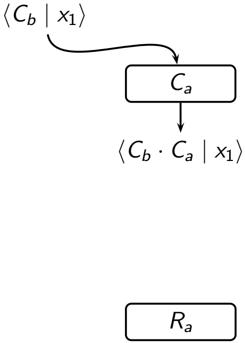
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$



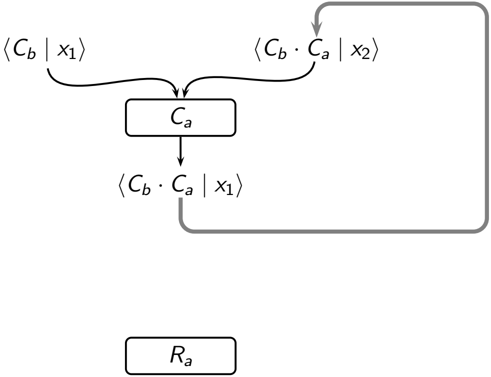
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$



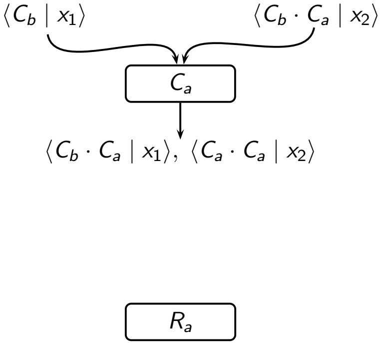
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$



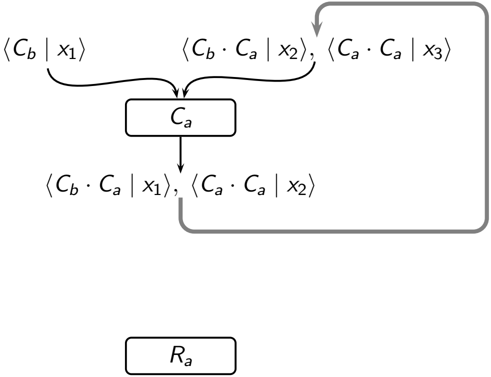
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$



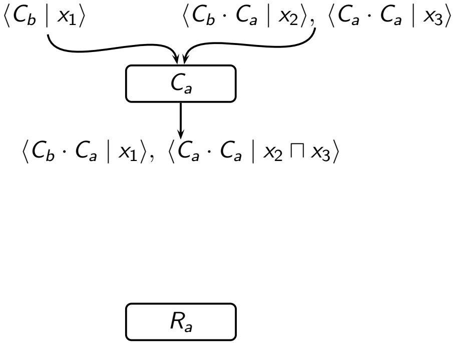
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$



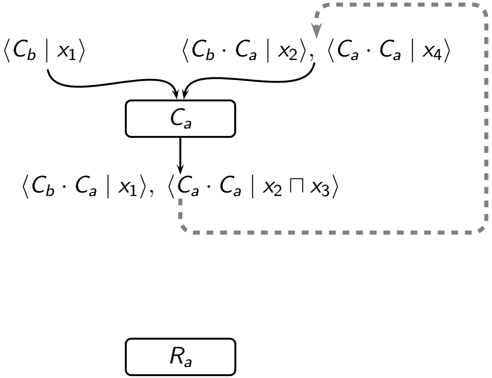
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$



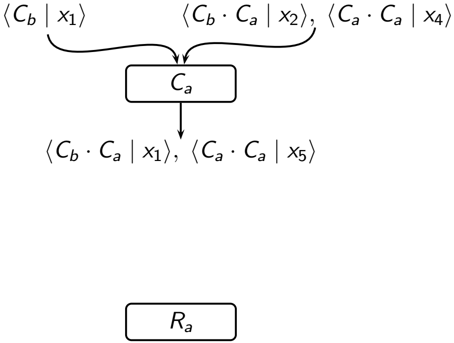
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$



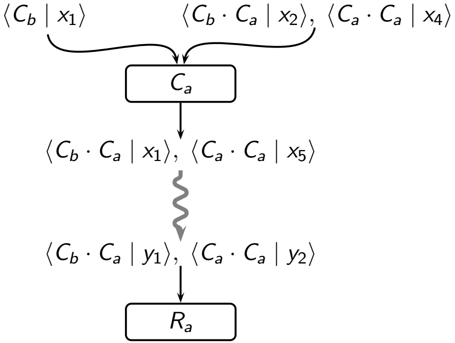
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$



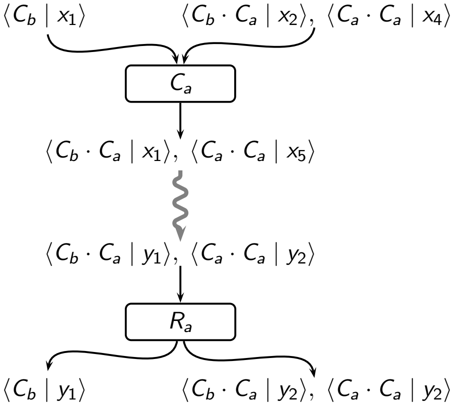
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$



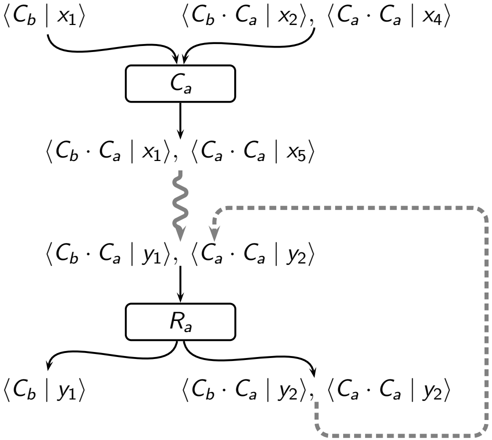
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$



Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$



Value Based Termination of Call String Construction

- 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.

Value Based Termination of Call String Construction

- 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.

Value Based Termination of Call String Construction

- 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.

Value Based Termination of Call String Construction

- 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.

Value Based Termination of Call String Construction

- 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.

Value Based Termination of Call String Construction

- 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.

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.