Interprocedural Data Flow Analysis

Uday Khedker

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

October 2018

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.
  (Indian edition published by Ane Books in 2013) *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
- Classical call strings approach
- Value context based approach

Part 2

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

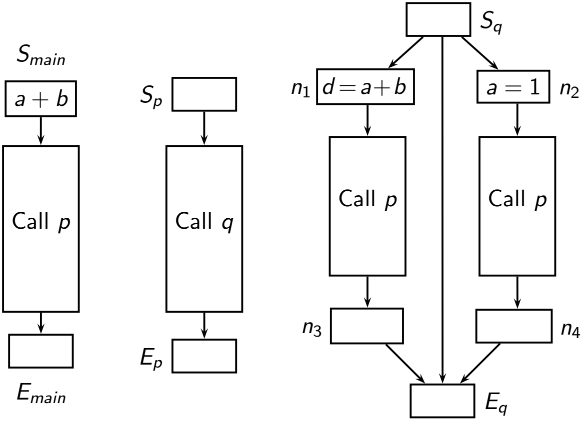
- Should achieve the effect of inlining callee procedures at their call sites

  Even in case of recursion

Why Interprocedural Analysis?

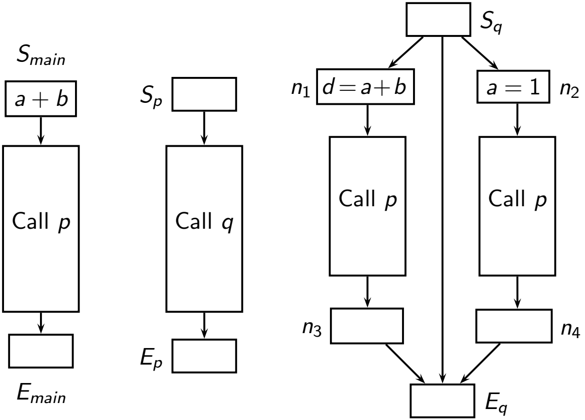
- Answering questions about formal parameters and global variables:
  - Which variables are constant?
  - Which variables aliased with each other?
  - Which locations can a pointer variable point to?
- Answering questions about side effects of a procedure call:
  - Which variables are defined or used 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

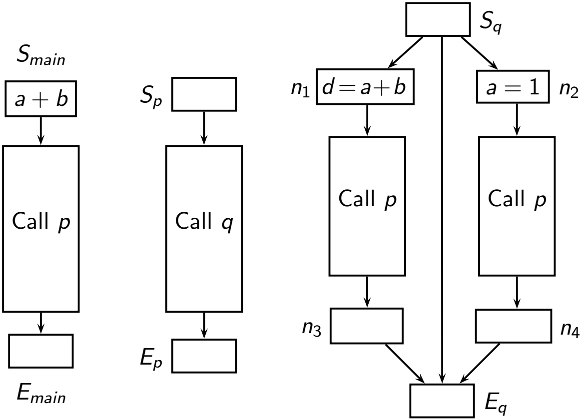
Program Representation for Interprocedural Data Flow Analysis: Call Multi-Graph



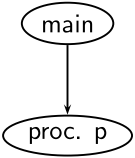
Supergraphs of procedures

Call multi-graph

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

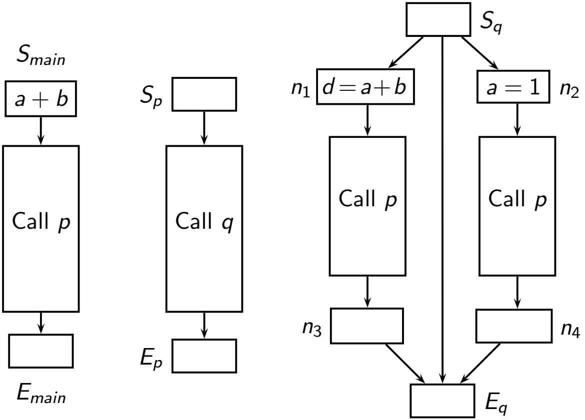


Supergraphs of procedures

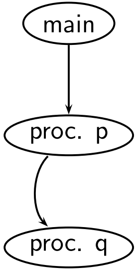


Call multi-graph

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

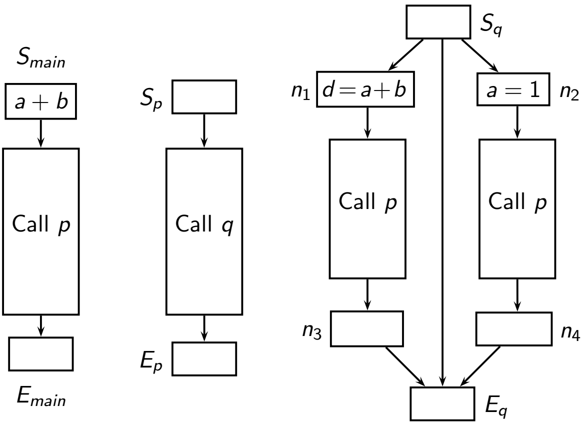


Supergraphs of procedures

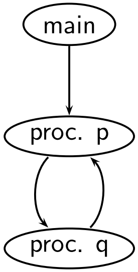


Call multi-graph

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

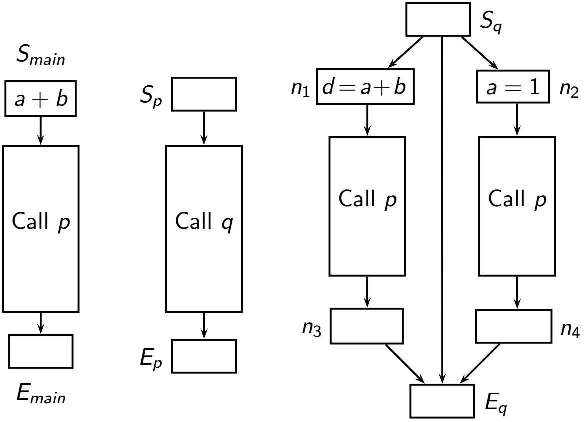


Supergraphs of procedures

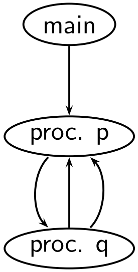


Call multi-graph

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

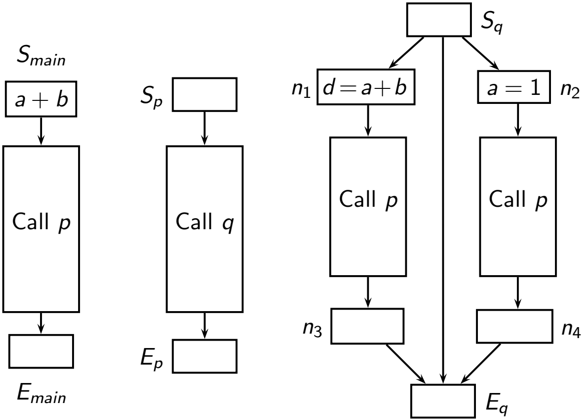


Supergraphs of procedures

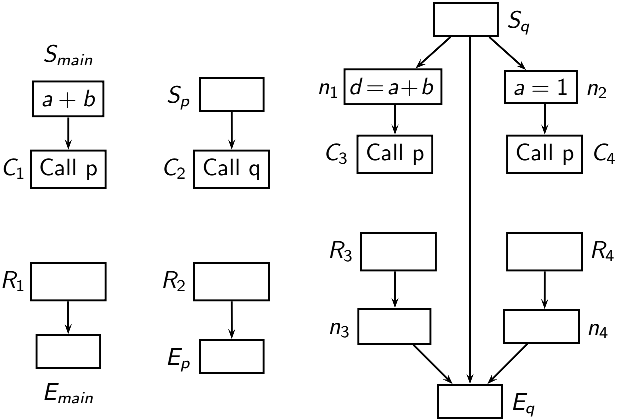


Call multi-graph

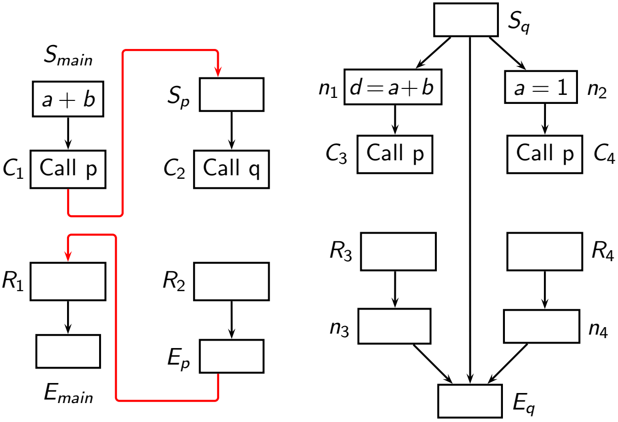
Program Representation for Interprocedural Data Flow Analysis: Supergraph



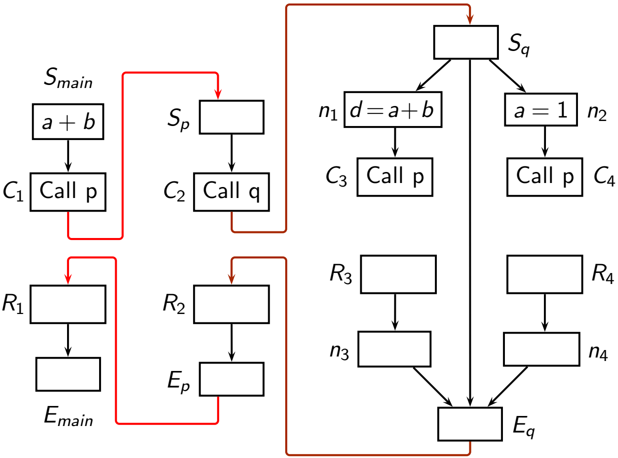
Program Representation for Interprocedural Data Flow Analysis: Supergraph



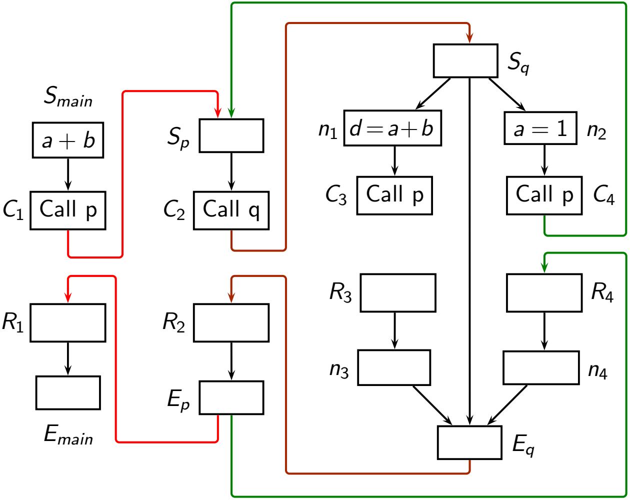
Program Representation for Interprocedural Data Flow Analysis: Supergraph



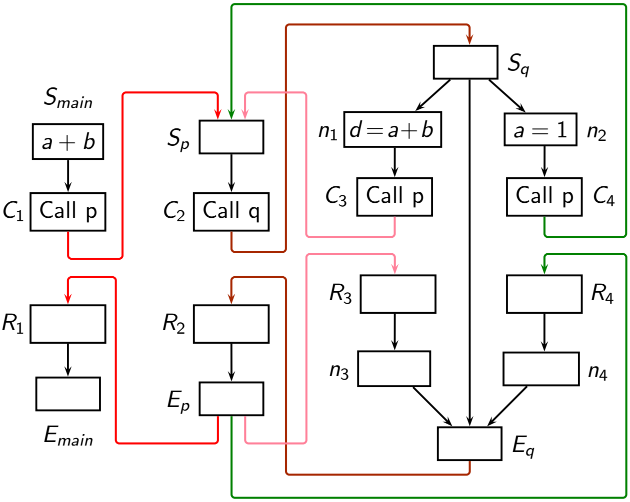
Program Representation for Interprocedural Data Flow Analysis: Supergraph



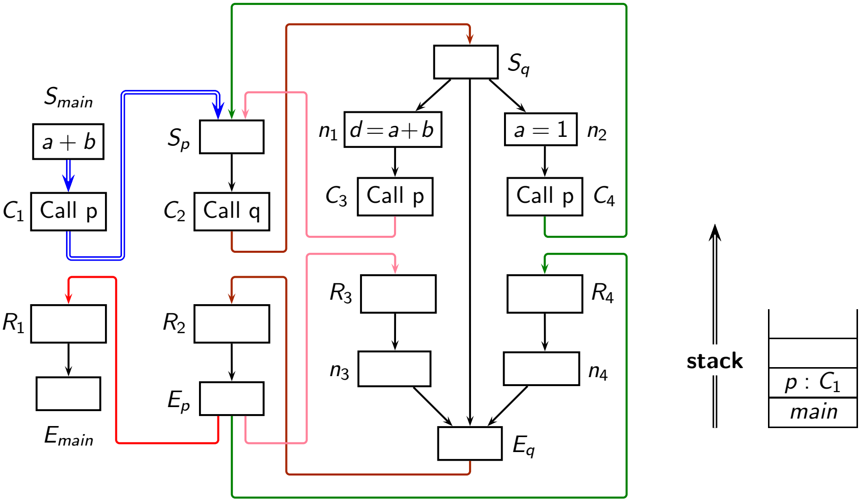
Program Representation for Interprocedural Data Flow Analysis: Supergraph



Program Representation for Interprocedural Data Flow Analysis: Supergraph

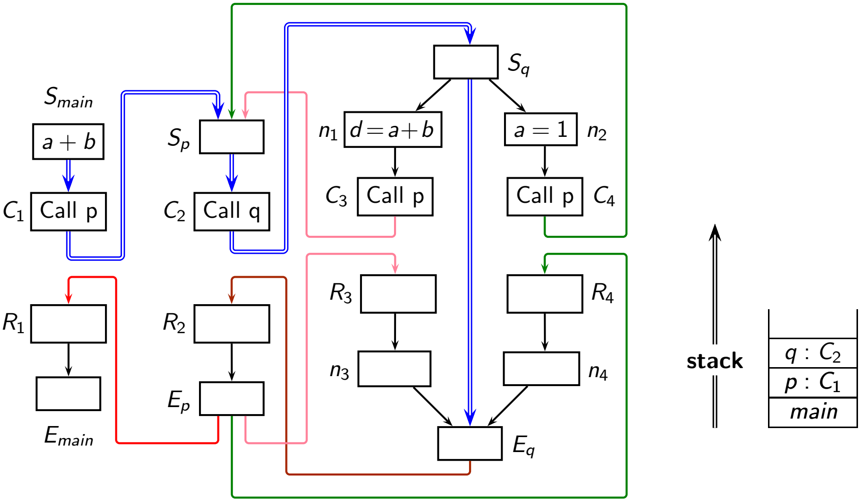


Validity of Interprocedural Control Flow Paths



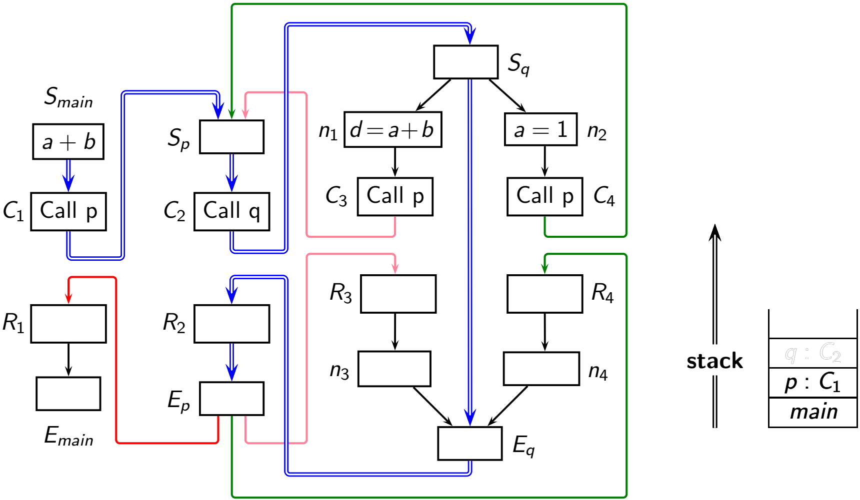
Interprocedurally valid control flow path

Validity of Interprocedural Control Flow Paths



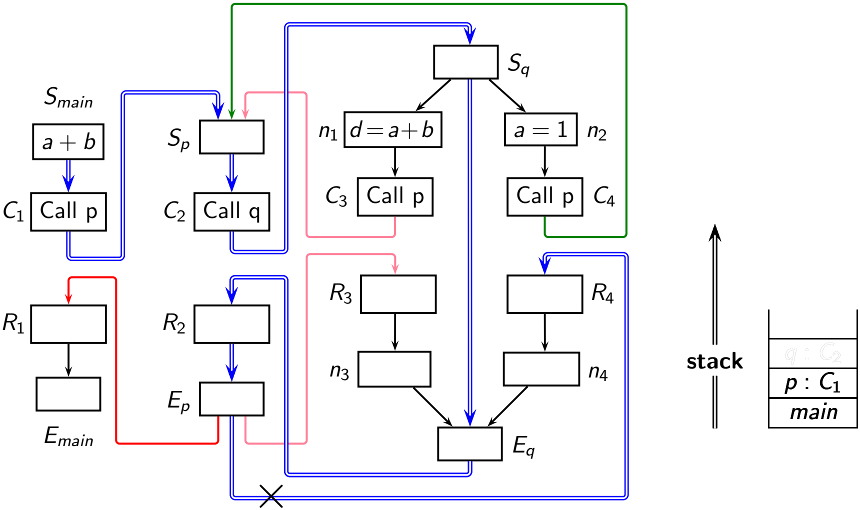
Interprocedurally valid control flow path

Validity of Interprocedural Control Flow Paths



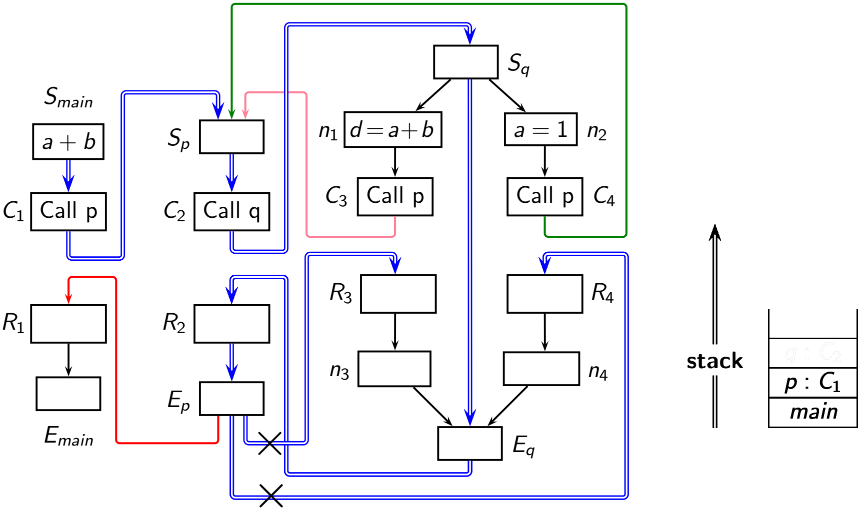
Interprocedurally valid control flow path

Validity of Interprocedural Control Flow Paths



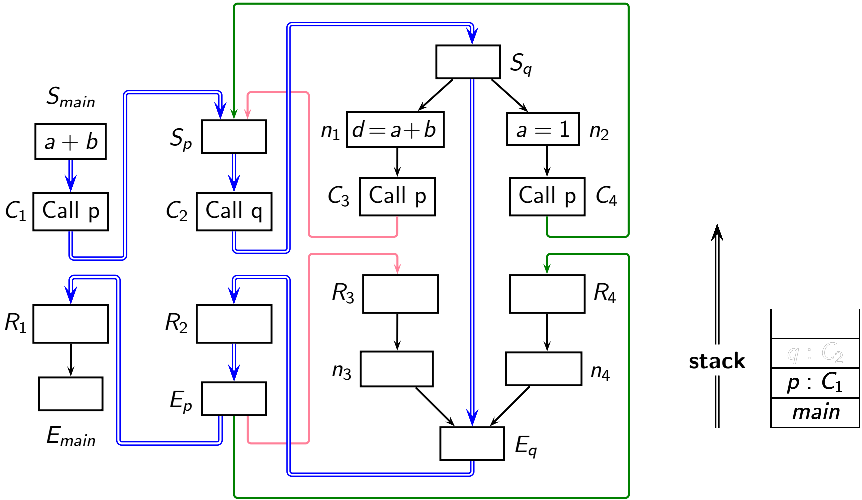
Interprocedurally invalid control flow path

Validity of Interprocedural Control Flow Paths



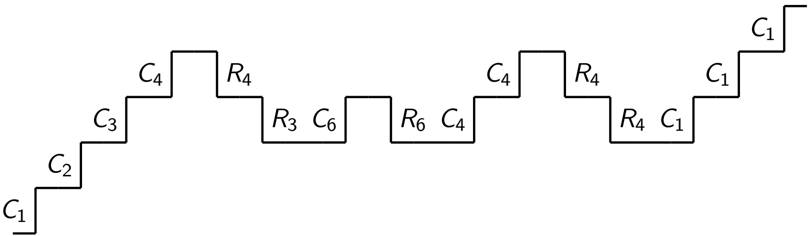
Interprocedurally invalid control flow path

Validity of Interprocedural Control Flow Paths

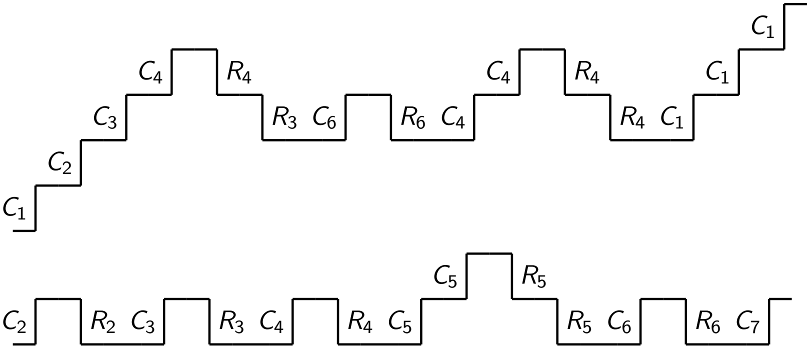


Interprocedurally valid control flow path

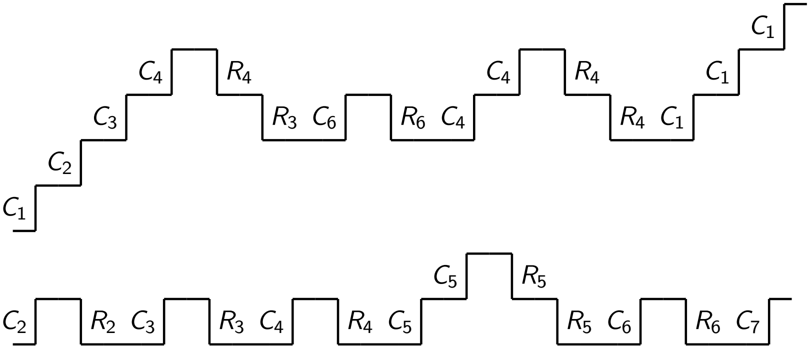
Staircase Diagrams of Interprocedurally Valid Paths



Staircase Diagrams of Interprocedurally Valid Paths

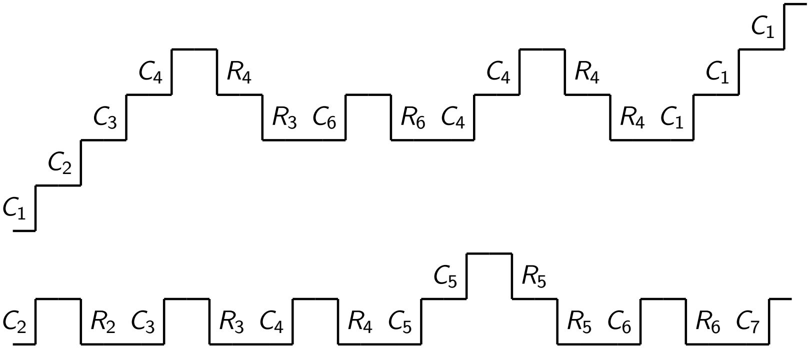


Staircase Diagrams of Interprocedurally Valid Paths



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

Staircase Diagrams of Interprocedurally Valid Paths

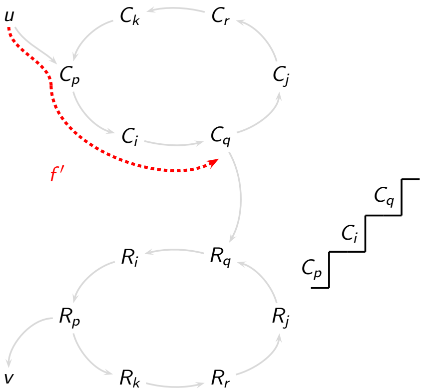


- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step

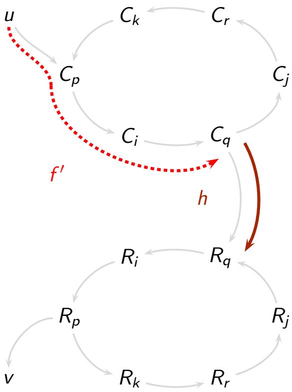
Staircase Diagrams in Presence of Recursion

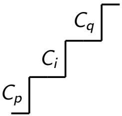
$$
\begin{tikzcd}
u \arrow[dr, bend left] & C_k \arrow[dl, bend right] & C_r \arrow[l, bend right] & \\
& C_p \arrow[dr, bend right] & & C_j \arrow[ul, bend right] \\
& & C_i \arrow[r, bend right] & C_q \arrow[u, bend right] \arrow[dd, bend left] \\
& & & \\
& R_i \arrow[dl, bend right] & R_q \arrow[l, bend right] & \\
v & R_p \arrow[l, bend right] \arrow[dr, bend right] & & R_j \arrow[ul, bend right] \\
& & R_k \arrow[r, bend right] & R_r \arrow[u, bend right]
\end{tikzcd}
$$

Staircase Diagrams in Presence of Recursion

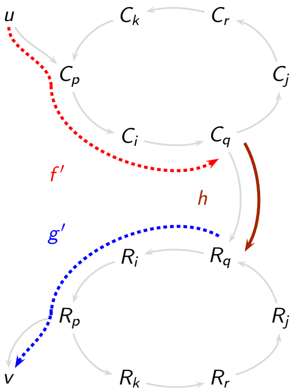


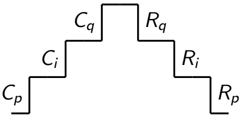
Staircase Diagrams in Presence of Recursion



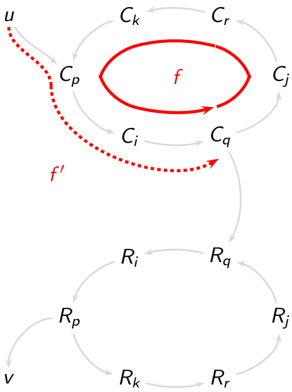


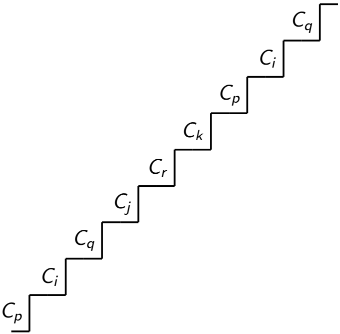
Staircase Diagrams in Presence of Recursion



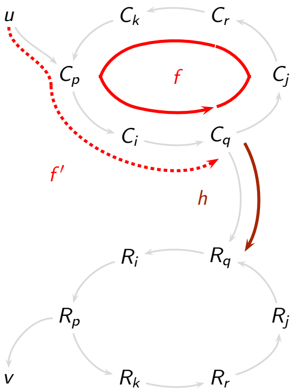


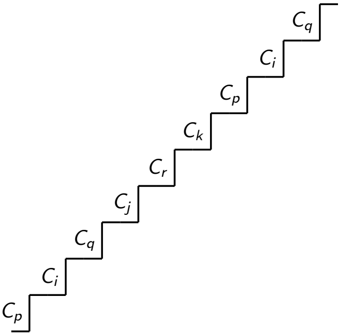
Staircase Diagrams in Presence of Recursion



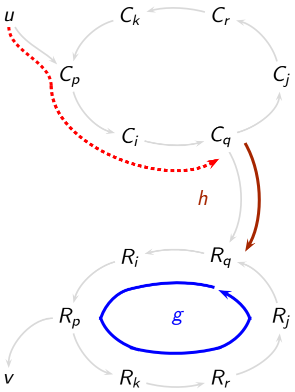


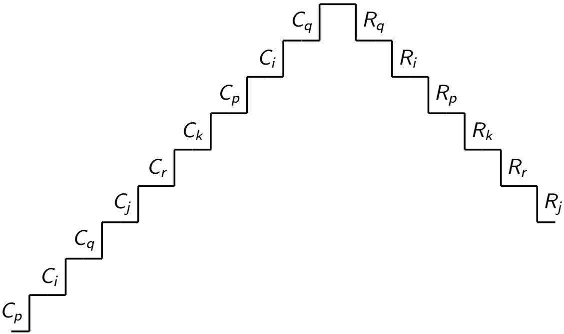
Staircase Diagrams in Presence of Recursion



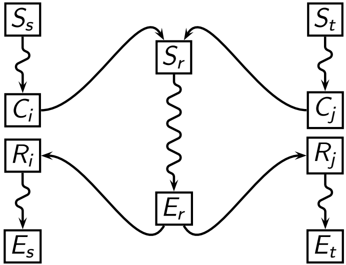


Staircase Diagrams in Presence of Recursion

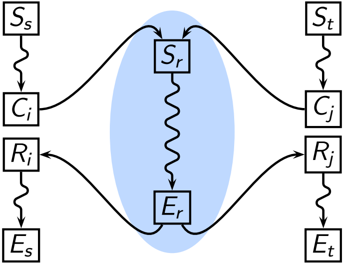




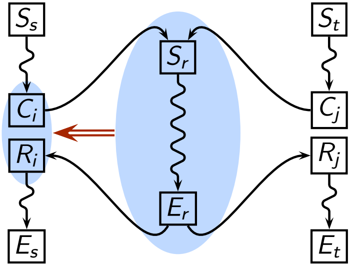
Understanding Context Sensitivity



Understanding Context Sensitivity

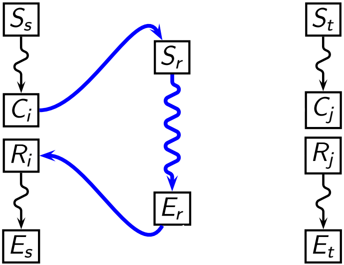


Understanding Context Sensitivity



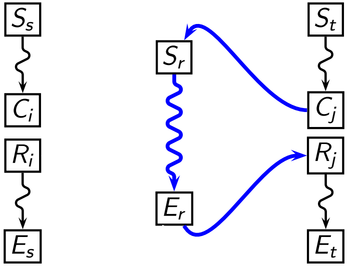
Precise interprocedural analysis aims to achieve the effect of inlining

Understanding Context Sensitivity



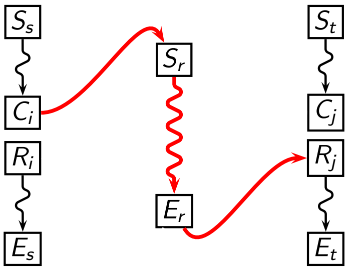
Interprocedurally valid path

Understanding Context Sensitivity



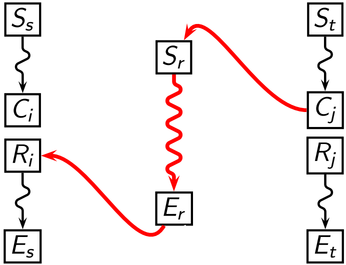
Interprocedurally valid path

Understanding Context Sensitivity



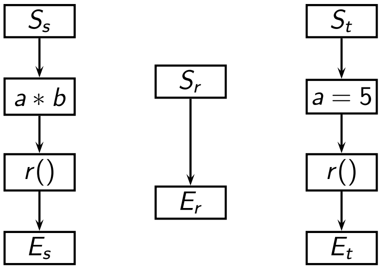
Interprocedurally invalid path

Understanding Context Sensitivity

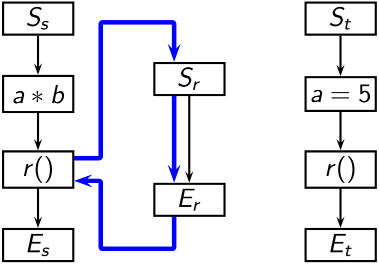


Interprocedurally invalid path

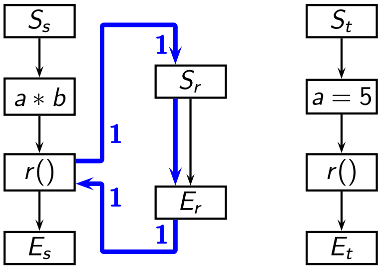
Context Sensitivity Vs. Context Insensitivity

Context-sensitive Analysis	Context-insensitive Analysis
	
Data flow values of distinct contexts are kept as separate values	Data flow values of all contexts are merged into a single value

Context Sensitivity Vs. Context Insensitivity

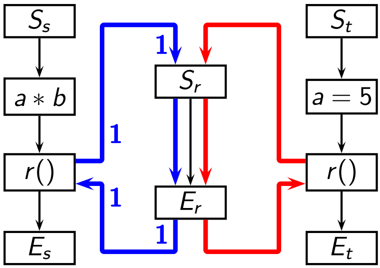
Context-sensitive Analysis	Context-insensitive Analysis
	
Data flow values of distinct contexts are kept as separate values	Data flow values of all contexts are merged into a single value

Context Sensitivity Vs. Context Insensitivity

Context-sensitive Analysis	Context-insensitive Analysis
	
Data flow values of distinct contexts are kept as separate values	Data flow values of all contexts are merged into a single value

Context Sensitivity Vs. Context Insensitivity

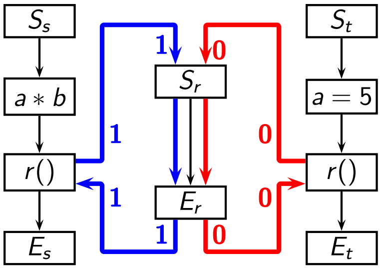
Context-sensitive Analysis	Context-insensitive Analysis



Data flow values of distinct contexts are kept as separate values

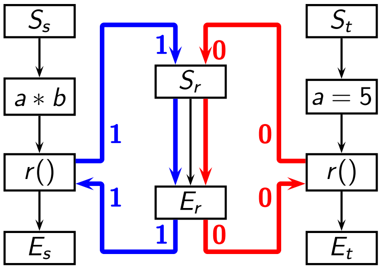
Data flow values of all contexts are merged into a single value

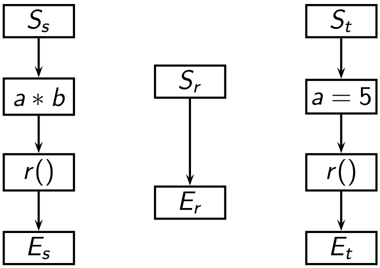
Context Sensitivity Vs. Context Insensitivity

Context-sensitive Analysis	Context-insensitive Analysis
	
Data flow values of distinct contexts are kept as separate values	Data flow values of all contexts are merged into a single value

Context Sensitivity Vs. Context Insensitivity

Context-sensitive Analysis	Context-insensitive Analysis

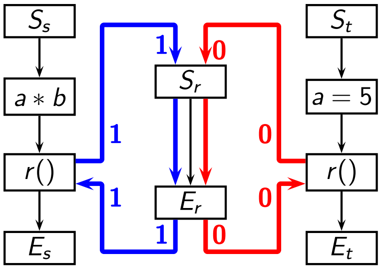
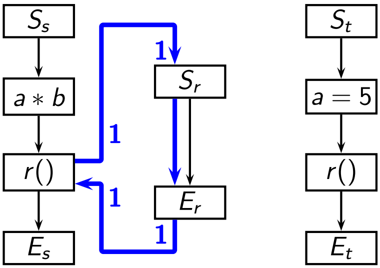




Data flow values of distinct contexts are kept as separate values

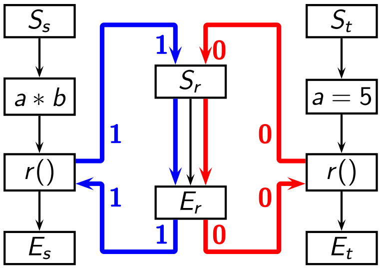
Data flow values of all contexts are merged into a single value

Context Sensitivity Vs. Context Insensitivity

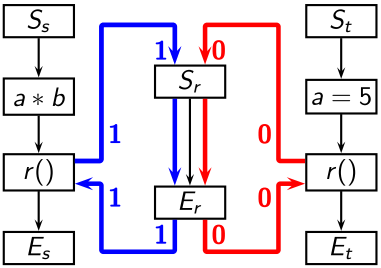
Context-sensitive Analysis	Context-insensitive Analysis
	
Data flow values of distinct contexts are kept as separate values	Data flow values of all contexts are merged into a single value

Context Sensitivity Vs. Context Insensitivity

Context-sensitive Analysis	Context-insensitive Analysis



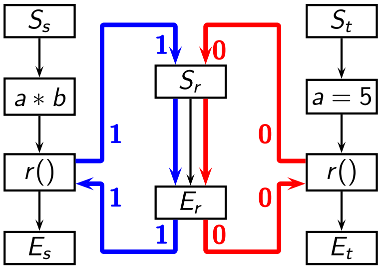
Data flow values of distinct contexts are kept as separate values



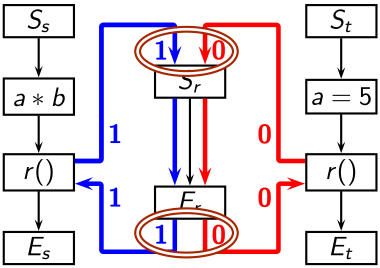
Data flow values of all contexts are merged into a single value

Context Sensitivity Vs. Context Insensitivity

Context-sensitive Analysis	Context-insensitive Analysis



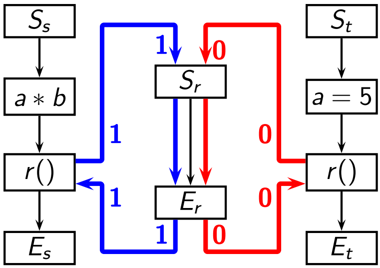
Data flow values of distinct contexts are kept as separate values



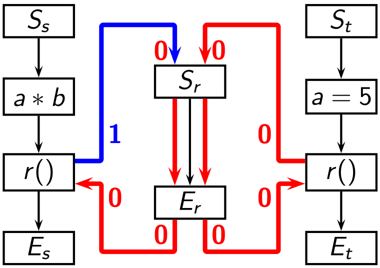
Data flow values of all contexts are merged into a single value

Context Sensitivity Vs. Context Insensitivity

Context-sensitive Analysis	Context-insensitive Analysis



Data flow values of distinct contexts are kept as separate values



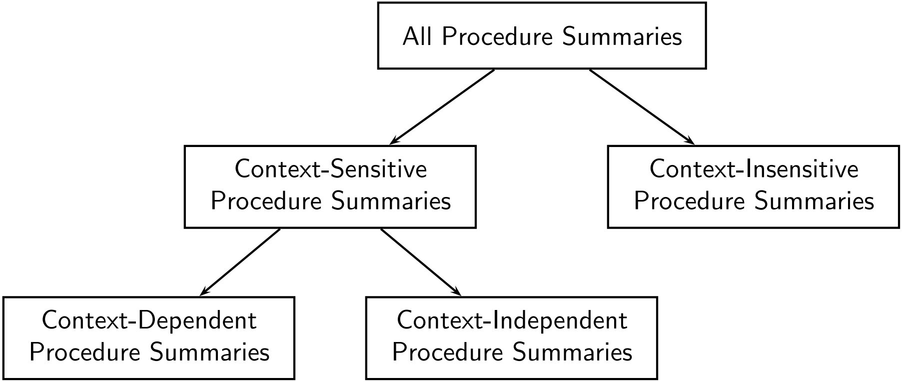
Data flow values of all contexts are merged into a single value

Understanding Context Sensitivity

The effect of inlining is achieved by

- call-return matching (call strings method),
- computing the summary of a procedure and incorporating it at the call point (functional method), or
- analysing a procedure for a particular data flow value and using the analysed result at the call point (graph reachability, value context method)

A Taxonomy of Terms



The Nature of Context-Sensitive Procedure Summaries

Context-sensitive methods summarize a procedure and use the prcocedure summaries in the callers at the call sites

A procedure summary is computed in terms of data flow values or flow functions

The Nature of Context-Sensitive Procedure Summaries

Context Dependent

Contexts are a part of the summary and are used to look up the data flow values

Context Independent

Contexts are not a part of a procedure summary

The summary is used at the call sites in the callers

This enables context sensitivity

The Nature of Context-Sensitive Procedure Summaries

Context Dependent	Context Independent
Extensional representation	Intensional representation

The Nature of Context-Sensitive Procedure Summaries

Context Dependent

Extensional representation

Enumeration of key-value pairs

Example

sq : Int $\rightarrow$ Int

$\{1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 9, \ldots\}$

Context Independent

Intensional representation

Parametrised expressions

Example

sq : Int $\rightarrow$ Int

$\lambda x.x^2$ or simply x^2

The Nature of Context-Sensitive Procedure Summaries

Context Dependent

Extensional representation

Example of constant propagation

For procedure p: $\boxed{\begin{array}{l} x = y + 1 \\ y = 2 \end{array}}$

Consider contexts σ_1 and σ_2 and let $y \mapsto 3$ in σ_1 and $y \mapsto 8$ in σ_2

Procedure summary $\mathbb{S}_p$ is

$$\left\{ (\sigma_1, \{x \mapsto 4, y \mapsto 2\}), \; (\sigma_2, \{x \mapsto 9, y \mapsto 2\}) \right\}$$

Context Independent

Intensional representation

The Nature of Context-Sensitive Procedure Summaries

Context Dependent

Extensional representation

Example of constant propagation

For procedure p: $\boxed{\begin{array}{l} x = y + 1 \\ y = 2 \end{array}}$

Consider contexts σ_1 and σ_2 and let $y \mapsto 3$ in σ_1 and $y \mapsto 8$ in σ_2

Procedure summary $\mathbb{S}_p$ is

$$\left\{ (\sigma_1, \{x \mapsto 4, y \mapsto 2\}), \; (\sigma_2, \{x \mapsto 9, y \mapsto 2\}) \right\}$$

Context Independent

Intensional representation

Example of constant propagation

For procedure p: $\boxed{\begin{array}{l} x = y + 1 \\ y = 2 \end{array}}$

Procedure summary $\mathbb{S}_p$ is

$$\langle x, y \rangle = \langle y + 1, 2 \rangle$$

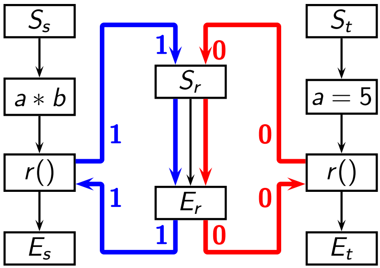
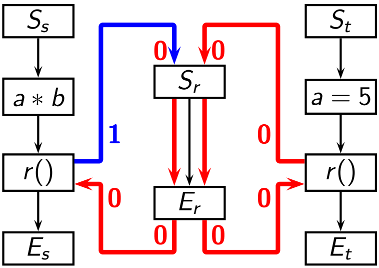
The Nature of Context-Sensitive Procedure Summaries

Context Dependent	Context Independent
Extensional representation	Intensional representation
Computed using a top-down traversal over the call graph	Computed using a bottom-up traversal over the call graph

The Nature of Context-Sensitive Procedure Summaries

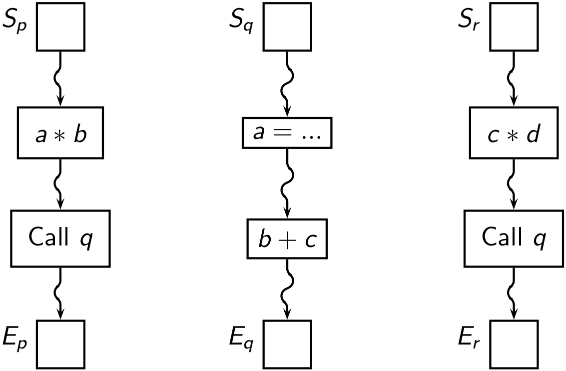
Context Dependent	Context Independent
Extensional representation	Intensional representation
Computed using a top-down traversal over the call graph	Computed using a bottom-up traversal over the call graph
Known methods Call strings, Value contexts, IFDS, IDE	Known method Functional approach

Context-Sensitive Vs. Context-Insensitive Procedure Summaries

Context-sensitive Analysis	Context-insensitive Analysis
	
Data flow values of distinct contexts are kept as separate values	Data flow values of all contexts are merged into a single value
$\{\, (\sigma_1, \{a*b \mapsto 1\}), (\sigma_2, \{a*b \mapsto 0\}) \,\}$	$\{a*b \mapsto 0\}\}$

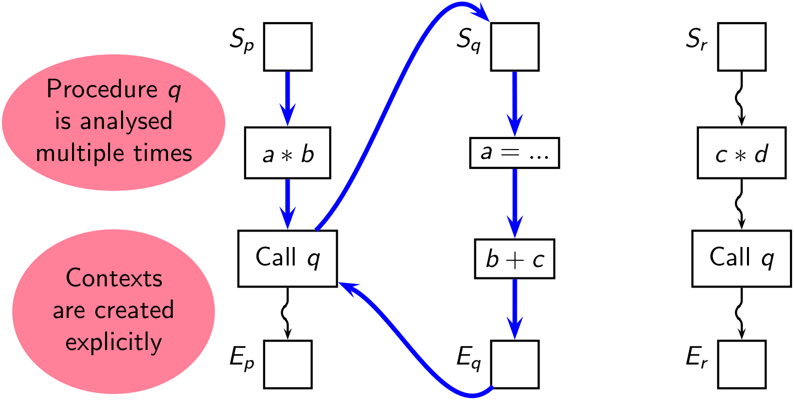
Top-down Vs. Bottom-up (Context-Sensitive) Procedure Summaries

Top-down Analysis for Available Expressions Analysis (Extensional Summary)



Top-down Vs. Bottom-up (Context-Sensitive) Procedure Summaries

Top-down Analysis for Available Expressions Analysis (Extensional Summary)



Context σ_1 Expression $b + c$ is available in procedure p

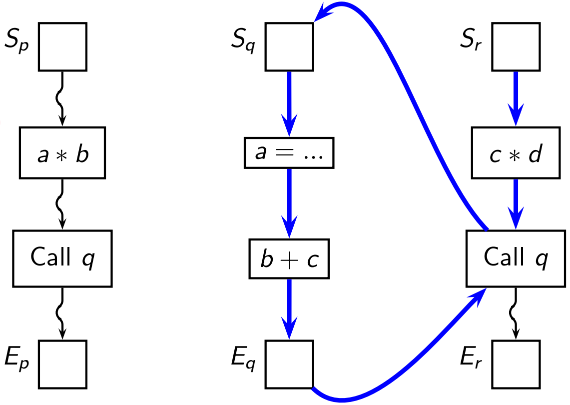
Expression $a * b$ is not available in procedure p

Top-down Vs. Bottom-up (Context-Sensitive) Procedure Summaries

Top-down Analysis for Available Expressions Analysis (Extensional Summary)

Procedure q is analysed multiple times

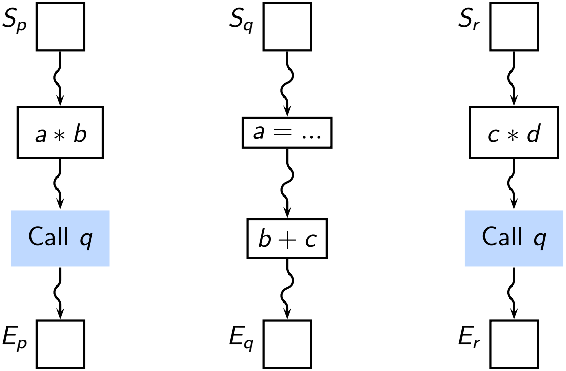
Contexts are created explicitly



Context σ_2 Expressions $b + c$ and $c * d$ are available in procedure r

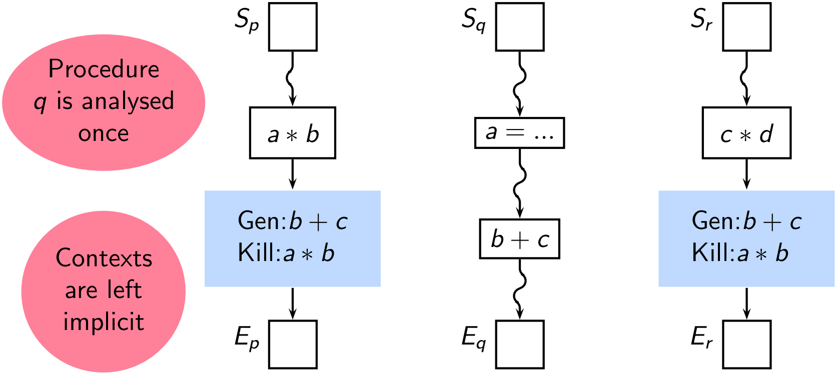
Top-down Vs. Bottom-up (Context-Sensitive) Procedure Summaries

Bottom-Up Analysis for Available Expressions Analysis (Intensional Summary)



Top-down Vs. Bottom-up (Context-Sensitive) Procedure Summaries

Bottom-Up Analysis for Available Expressions Analysis (Intensional Summary)



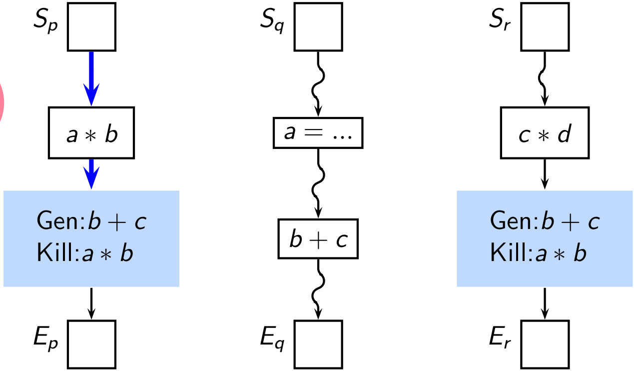
Using procedure summary of g at call sites

Top-down Vs. Bottom-up (Context-Sensitive) Procedure Summaries

Bottom-Up Analysis for Available Expressions Analysis (Intensional Summary)

Procedure q is analysed once

Contexts are left implicit

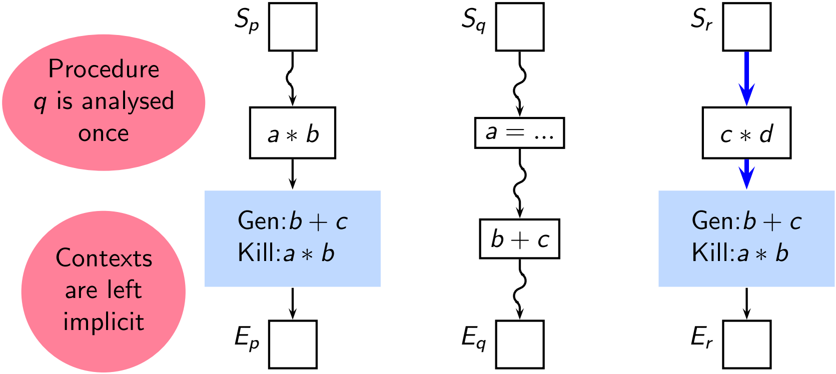


Expression $b + c$ is available in procedure p

Expression $a * b$ is not available in procedure p

Top-down Vs. Bottom-up (Context-Sensitive) Procedure Summaries

Bottom-Up Analysis for Available Expressions Analysis (Intensional Summary)



Expressions $b + c$ and $c * d$ are available in procedure r

Issues in Top-down Vs. Bottom-up Interprocedural Analysis

- Bottom-up approach
  - Compact representation
  - Information may depend on the calling context
- Top-down approach
  - Needs to visit a procedure separately for every calling context
  - Exponentially large number of calling contexts
  - Many contexts may have no effect on the procedure

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

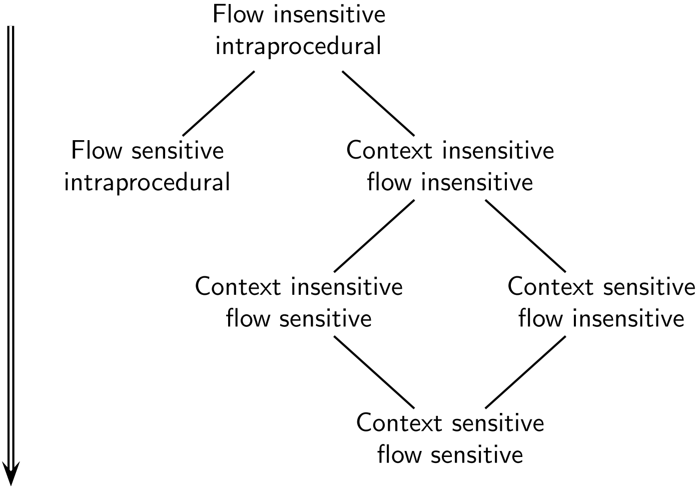
- Flow sensitive analysis:
  Considers intraprocedurally valid paths
- Context sensitive analysis:
  Considers interprocedurally valid paths
- For precision, analysis must be both flow- and context-sensitive

Flow and Context Sensitivity

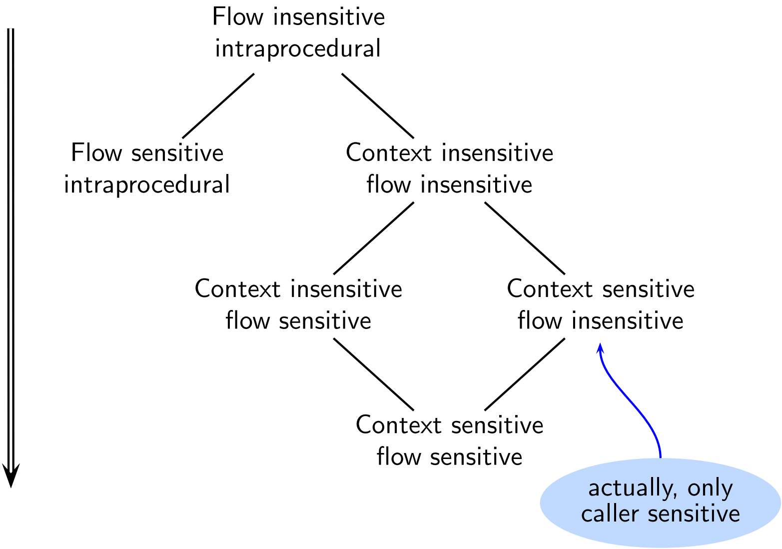
- Flow sensitive analysis:
  Considers intraprocedurally valid paths
- Context sensitive analysis:
  Considers interprocedurally valid paths
- For precision, analysis must be both flow- and context-sensitive

MFP computation restricted to valid paths only

Increasing Precision in Data Flow Analysis



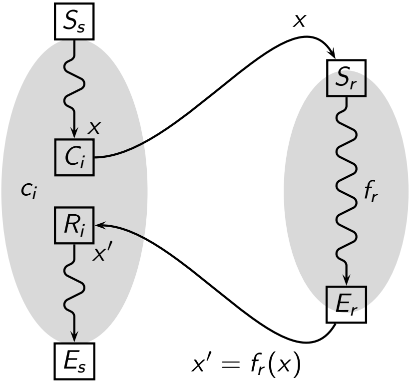
Increasing Precision in Data Flow Analysis



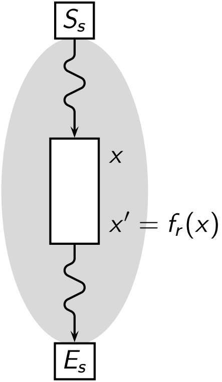
Part 3

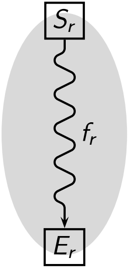
Classical Functional Approach

Functional Approach



Functional Approach



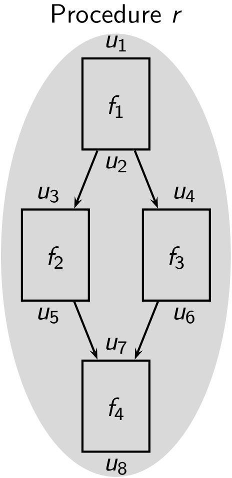


- Bottom-up Approach
- Compute summary flow functions for each procedure
- Use summary flow functions as the flow function for a call block
- Main challenge:

  Appropriate representation for summary flow functions

Notation for Summary Flow Function

Assuming forward flow

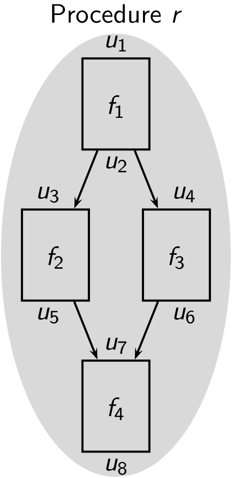


Notation for Summary Flow Function

Assuming forward flow

- u_i: Program points
- f_i: Node flow functions
- $\Phi_r(u_i)$: Summary flow function mapping data flow value from S_r to u_i

  $X(u_i) = \Phi_r(u_i)(BI)$

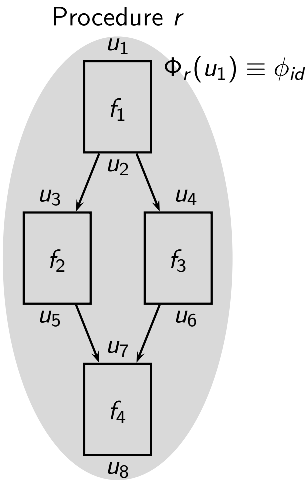


Notation for Summary Flow Function

Assuming forward flow

- u_i: Program points
- f_i: Node flow functions
- $\Phi_r(u_i)$: Summary flow function mapping data flow value from S_r to u_i

  $X(u_i) = \Phi_r(u_i)(BI)$

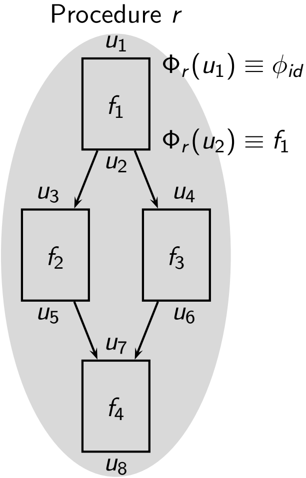


Notation for Summary Flow Function

Assuming forward flow

- u_i: Program points
- f_i: Node flow functions
- $\Phi_r(u_i)$: Summary flow function mapping data flow value from S_r to u_i

  $X(u_i) = \Phi_r(u_i)(BI)$

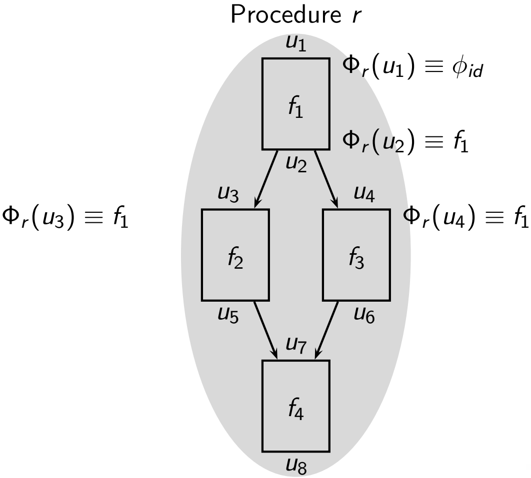


Notation for Summary Flow Function

Assuming forward flow

- u_i: Program points
- f_i: Node flow functions
- $\Phi_r(u_i)$: Summary flow function mapping data flow value from S_r to u_i

  $X(u_i) = \Phi_r(u_i)(BI)$

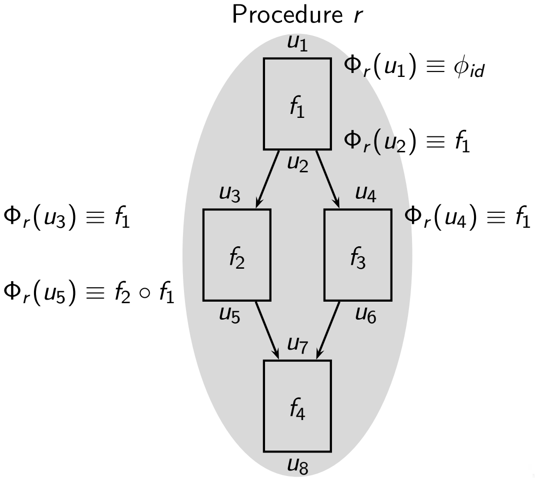


Notation for Summary Flow Function

Assuming forward flow

- u_i: Program points
- f_i: Node flow functions
- $\Phi_r(u_i)$: Summary flow function mapping data flow value from S_r to u_i

  $X(u_i) = \Phi_r(u_i)(BI)$

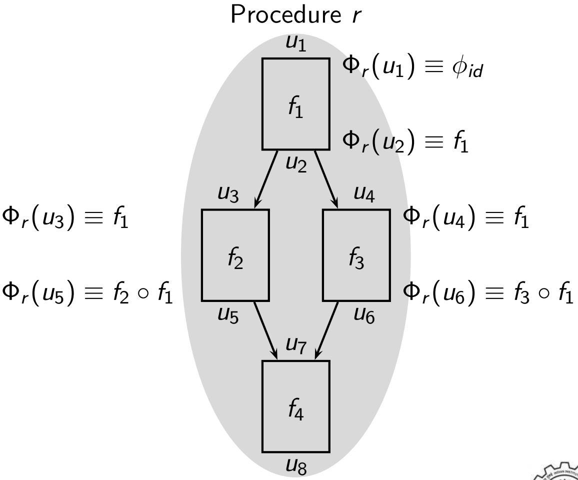


Notation for Summary Flow Function

Assuming forward flow

- u_i: Program points
- f_i: Node flow functions
- $\Phi_r(u_i)$: Summary flow function mapping data flow value from S_r to u_i

  $X(u_i) = \Phi_r(u_i)(BI)$

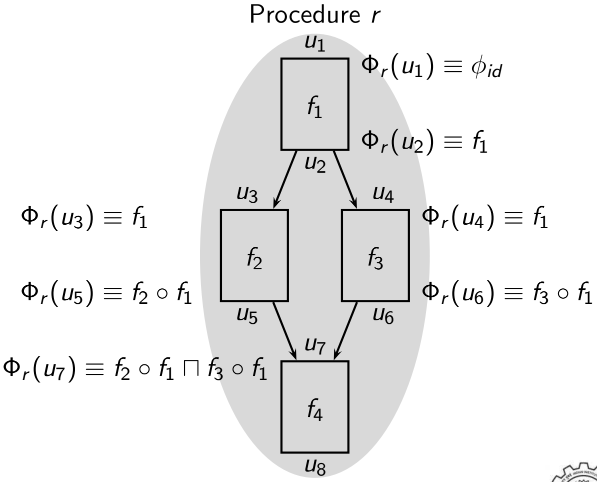


Notation for Summary Flow Function

Assuming forward flow

- u_i: Program points
- f_i: Node flow functions
- $\Phi_r(u_i)$: Summary flow function mapping data flow value from S_r to u_i

  $X(u_i) = \Phi_r(u_i)(BI)$

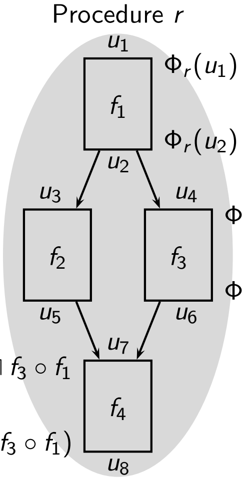


Notation for Summary Flow Function

Assuming forward flow

- u_i: Program points
- f_i: Node flow functions
- $\Phi_r(u_i)$: Summary flow function mapping data flow value from S_r to u_i

  $X(u_i) = \Phi_r(u_i)(BI)$



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

$\Phi_r(u_2) \equiv f_1$

$\Phi_r(u_3) \equiv f_1$

$\Phi_r(u_4) \equiv f_1$

$\Phi_r(u_5) \equiv f_2 \circ f_1$

$\Phi_r(u_6) \equiv f_3 \circ f_1$

$\Phi_r(u_7) \equiv f_2 \circ f_1 \sqcap f_3 \circ f_1$

$\Phi_r(u_8) \equiv f_4 \circ (f_2 \circ f_1 \sqcap f_3 \circ f_1)$

Equations for Constructing Summary Flow Functions

For simplicity forward flow is assumed. I_n is Entry of n, O_n is Exit of n

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

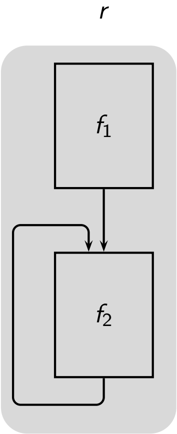
$$\Phi_r(O_n) = \begin{cases} \Phi_s(u) \circ \Phi_r(I_n) & \text{if } n \text{ calls procedure } s \text{ and } u \text{ is } O_{E_s} \\ f_n \circ \Phi_r(I_n) & \text{otherwise} \end{cases}$$

The summary flow function of a given procedure r

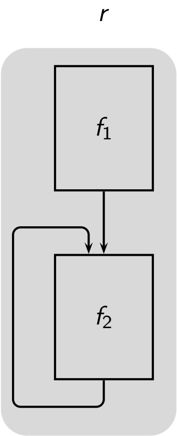
- is influenced by summary flow functions of the callees of r
- is not influenced by summary flow functions of the callers of r

Fixed point computation may be required in the presence of loops or recursion

Constructing Summary Flow Functions Iteratively



Constructing Summary Flow Functions Iteratively



Iteration #1

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

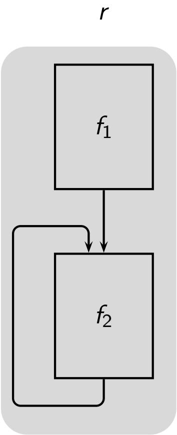
$\Phi_r(u_2) = f_1$

$\Phi_r(u_3) = f_1$

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

Constructing Summary Flow Functions Iteratively

Iteration #2



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

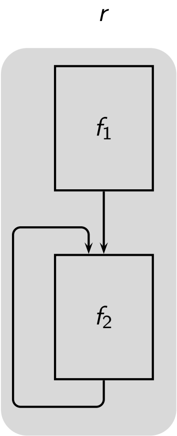
$\Phi_r(u_2) = f_1$

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

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

Constructing Summary Flow Functions Iteratively

Iteration #3



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

$\Phi_r(u_2) = f_1$

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

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

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><th>$\widehat{f}_n(x),\ \forall x \in \{\widehat{\top}, \widehat{\bot}\}$</th></tr>
<tr><td>$\emptyset$</td><td>$\emptyset$</td><td>$\widehat{\phi}_{id}$</td><td>x</td></tr>
<tr><td>$\emptyset$</td><td>$\{a\}$</td><td>$\widehat{\phi}_{\top}$</td><td>$\widehat{\top}$</td></tr>
<tr><td>$\{a\}$</td><td>$\emptyset$</td><td rowspan="2">$\widehat{\phi}_{\bot}$</td><td rowspan="2">$\widehat{\bot}$</td></tr>
<tr><td>$\{a\}$</td><td>$\{a\}$</td></tr>
</table>
</td>
<td>$\widehat{\phi}_{\top}$
$\downarrow$
$\widehat{\phi}_{id}$
$\downarrow$
$\widehat{\phi}_{\bot}$</td>
</tr>
</table>

Reducing Component Flow Functions for Live Variables Analysis

Let $\widehat{\phi} \in \{\widehat{\phi}_\top, \widehat{\phi}_{id}, \widehat{\phi}_\bot\}$ and $x \in \{1, 0\}$. Then,

- $\widehat{\phi}_\top \sqcap \widehat{\phi} = \widehat{\phi}$ (because $0 + x = x$)
- $\widehat{\phi}_\bot \sqcap \widehat{\phi} = \widehat{\phi}_\bot$ (because $1 + x = 1$)
- $\widehat{\phi}_\top \circ \widehat{\phi} = \widehat{\phi}_\top$ (because $\widehat{\phi}_\top$ is a constant function)
- $\widehat{\phi}_\bot \circ \widehat{\phi} = \widehat{\phi}_\bot$ (because $\widehat{\phi}_\bot$ is a constant function)
- $\widehat{\phi}_{id} \circ \widehat{\phi} = \widehat{\phi}$ (because $\widehat{\phi}_{id}$ is the identity function)

Reducing Function Compositions in Bit Vector Frameworks

Kill$_n$ denoted by K_n and Gen$_n$ denoted by G_n

$f_3(\mathrm{x}) = f_2(f_1(\mathrm{x}))$

Reducing Function Compositions in Bit Vector Frameworks

Kill_n denoted by K_n and Gen_n denoted by G_n

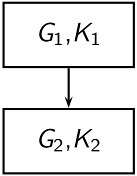
$$
\begin{aligned}
f_3(\mathsf{x}) &= f_2(f_1(\mathsf{x})) \\
&= f_2\big((\mathsf{x} - K_1) \cup G_1\big)
\end{aligned}
$$

$\boxed{G_1, K_1}$

Reducing Function Compositions in Bit Vector Frameworks

Kill_n denoted by K_n and Gen_n denoted by G_n

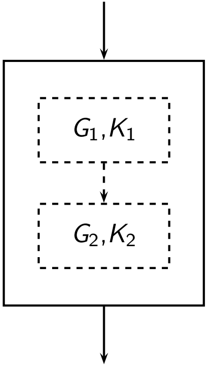
$$
\begin{aligned}
f_3(\mathsf{x}) &= f_2(f_1(\mathsf{x})) \\
&= f_2\big((\mathsf{x} - K_1) \cup G_1\big) \\
&= \Big(\big((\mathsf{x} - K_1) \cup G_1\big) - K_2\Big) \ \cup \ G_2
\end{aligned}
$$



Reducing Function Compositions in Bit Vector Frameworks

Kill$_n$ denoted by K_n and Gen$_n$ denoted by G_n

$$
\begin{aligned}
f_3(\mathsf{x}) &= f_2(f_1(\mathsf{x}))\\
&= f_2\big((\mathsf{x} - K_1) \cup G_1\big)\\
&= \Big(\big((\mathsf{x} - K_1) \cup G_1\big) - K_2\Big) \;\cup\; G_2\\
&= \big(\mathsf{x} - (K_1 \cup K_2)\big) \;\cup\; (G_1 - K_2) \;\cup\; G_2
\end{aligned}
$$



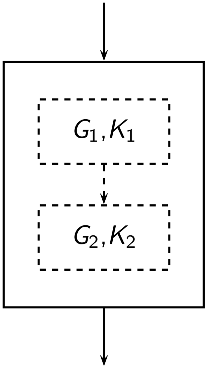
Reducing Function Compositions in Bit Vector Frameworks

Kill_n denoted by K_n and Gen_n denoted by G_n

$$\begin{aligned} f_3(x) &= f_2(f_1(x)) \\ &= f_2\big((x - K_1) \cup G_1\big) \\ &= \big(\big((x - K_1) \cup G_1\big) - K_2\big) \ \cup \ G_2 \\ &= \big(x - (K_1 \cup K_2)\big) \ \cup \ (G_1 - K_2) \ \cup \ G_2 \end{aligned}$$

Hence,

$$\begin{aligned} K_3 &= K_1 \cup K_2 \\ G_3 &= (G_1 - K_2) \ \cup \ G_2 \end{aligned}$$



Reducing Bit Vector Flow Function Confluences (1)

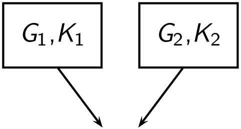
Kill_n denoted by K_n and Gen_n denoted by G_n

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

$$
\begin{aligned}
f_3(x) &= f_2(x) \cup f_1(x) \\
&= \big((x - K_2) \cup G_2\big) \;\cup\; \big((x - K_1) \cup G_1\big) \\
&= \big(x - (K_1 \cap K_2)\big) \;\cup\; \big(G_1 \cup G_2\big)
\end{aligned}
$$

Hence,

$$
\begin{aligned}
K_3 &= K_1 \cap K_2 \\
G_3 &= G_1 \cup G_2
\end{aligned}
$$



Reducing Bit Vector Flow Function Confluences (1)

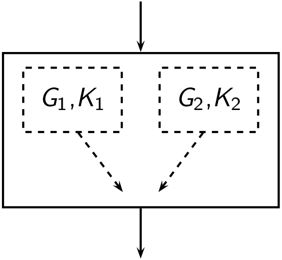
Kill_n denoted by K_n and Gen_n denoted by G_n

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

$$\begin{aligned} f_3(x) &= f_2(x) \cup f_1(x) \\ &= \big((x - K_2) \cup G_2\big) \ \cup \ \big((x - K_1) \cup G_1\big) \\ &= \big(x - (K_1 \cap K_2)\big) \ \cup \ \big(G_1 \cup G_2\big) \end{aligned}$$

Hence,

$$\begin{aligned} K_3 &= K_1 \cap K_2 \\ G_3 &= G_1 \cup G_2 \end{aligned}$$



Reducing Bit Vector Flow Function Confluences (2)

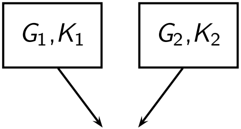
Kill_n denoted by K_n and Gen_n denoted by G_n

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

$$
\begin{aligned}
f_3(\mathsf{x}) &= f_2(\mathsf{x}) \cap f_1(\mathsf{x}) \\
&= \big((\mathsf{x} - K_2) \cup G_2\big) \;\cap\; \big((\mathsf{x} - K_1) \cup G_1\big) \\
&= \big(\mathsf{x} - (K_1 \cup K_2)\big) \;\cup\; \big(G_1 \cap G_2\big)
\end{aligned}
$$

Hence,

$$
\begin{aligned}
K_3 &= K_1 \cup K_2 \\
G_3 &= G_1 \cap G_2
\end{aligned}
$$



Reducing Bit Vector Flow Function Confluences (2)

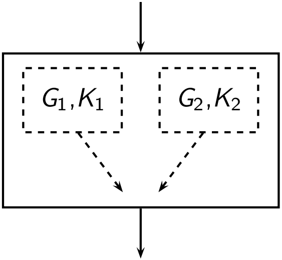
Kill_n denoted by K_n and Gen_n denoted by G_n

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

$$\begin{aligned} f_3(x) &= f_2(x) \cap f_1(x) \\ &= \big((x - K_2) \cup G_2\big) \;\cap\; \big((x - K_1) \cup G_1\big) \\ &= \big(x - (K_1 \cup K_2)\big) \;\cup\; \big(G_1 \cap G_2\big) \end{aligned}$$

Hence,

$$\begin{aligned} K_3 &= K_1 \cup K_2 \\ G_3 &= G_1 \cap G_2 \end{aligned}$$

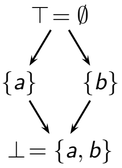


Lattice of Flow Functions for Live Variables Analysis

Flow functions for two variables

- Product of lattices for independent variables (because of separability)

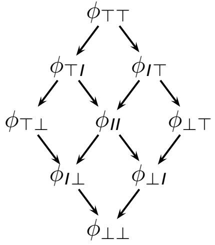
Lattice of data flow values



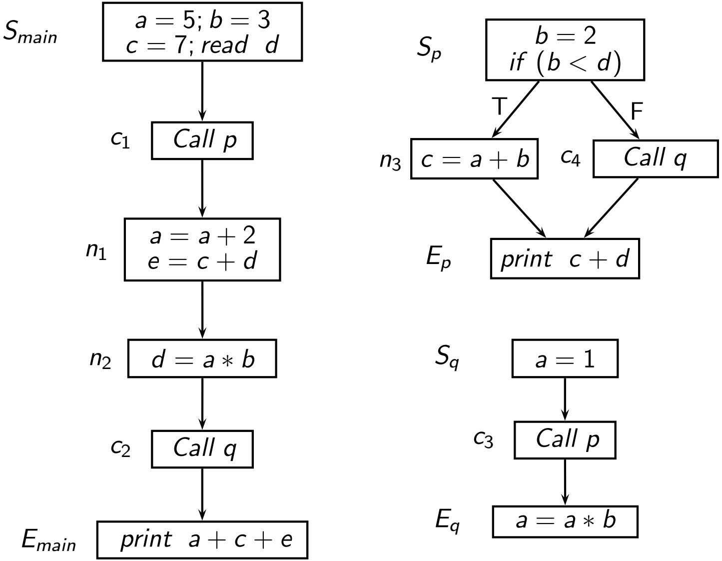
All possible flow functions

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



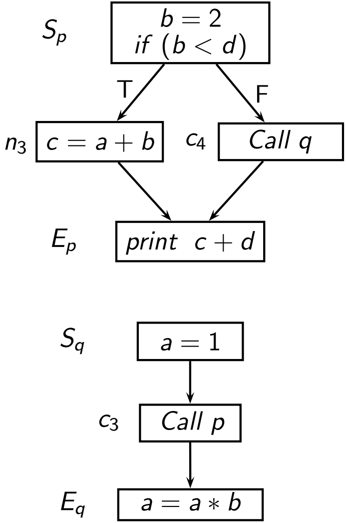
An Example of Interprocedural Liveness Analysis



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}	$\{c, d\}$	$\emptyset$		
	$\Phi_p(n_3)$	$f_{n_3} \circ \Phi_p(E_p)$	$\{a, b, d\}$	$\{c\}$		
	$\Phi_p(c_4)$	$f_q \circ \Phi_p(E_p) = \phi_\top$	$\emptyset$	$\{a, b, c, d, e\}$	$\{d\}$	$\{a, b, c\}$
	$\Phi_p(S_p)$	$f_{S_p} \circ \left(\Phi_p(n_3) \sqcap \Phi_p(c_4)\right)$	$\{a, d\}$	$\{b, c\}$		
	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 Functions



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

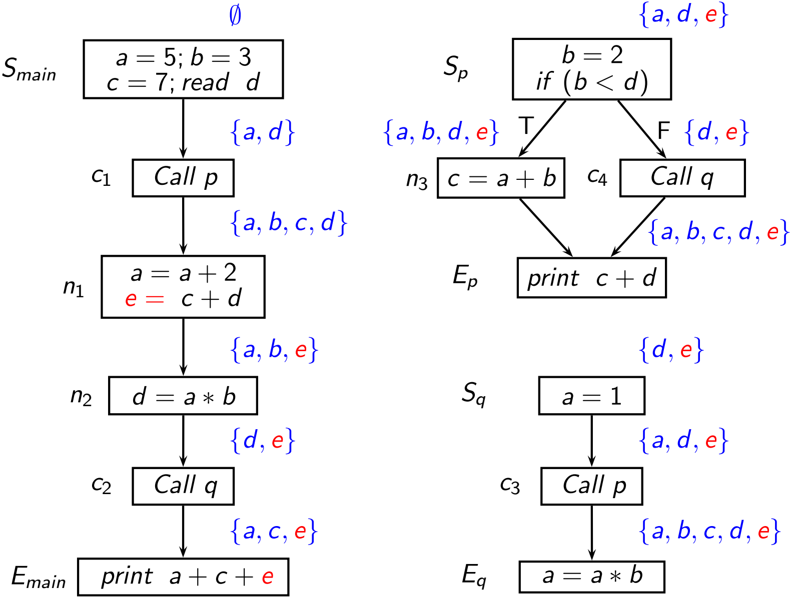
Result of Interprocedural Liveness Analysis

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

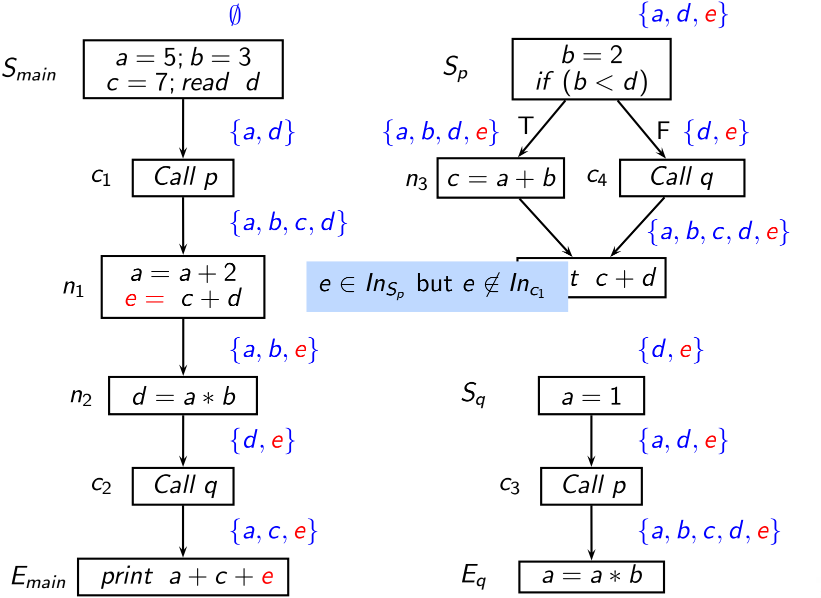
Result of Interprocedural Liveness Analysis

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

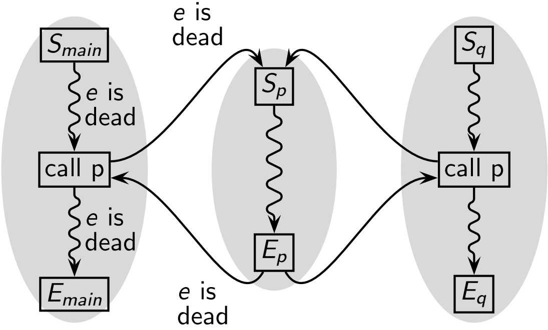
Context Sensitivity of Interprocedural Liveness Analysis



Context Sensitivity of Interprocedural Liveness Analysis

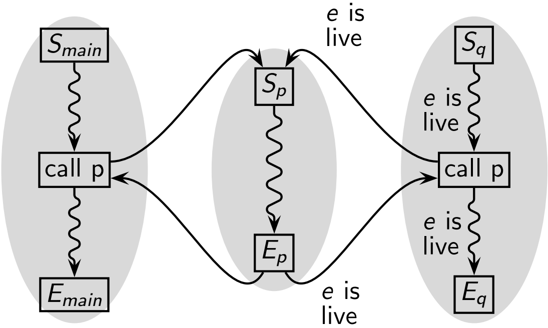


Explaining Context Sensitivity



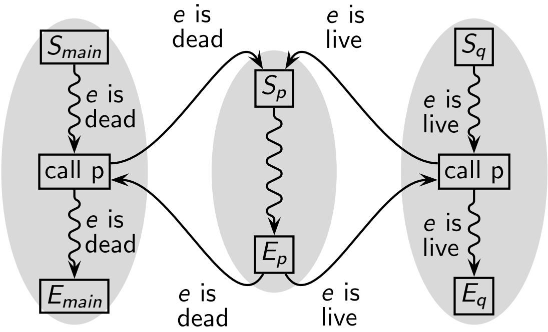
- Flow function of procedure p is identity with respect to variable e

Explaining Context Sensitivity



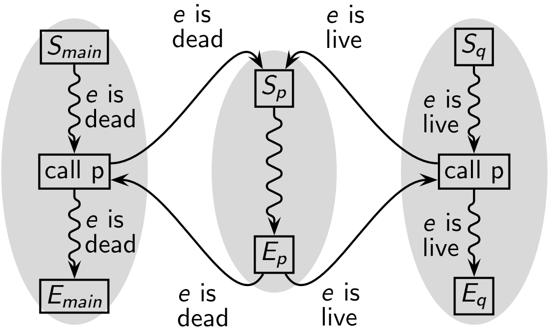
- Flow function of procedure p is identity with respect to variable e

Explaining Context Sensitivity



- Flow function of procedure p is identity with respect to variable e
- Is e live in the body of procedure p?
  - During the analysis: Depends on the calling context
  - After the analysis: Yes (static approximation across all executions)

Explaining Context Sensitivity



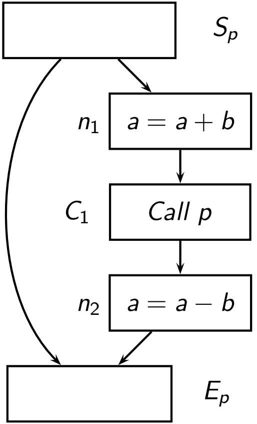
- Flow function of procedure p is identity with respect to variable e
- Is e live in the body of procedure p?
  - During the analysis: Depends on the calling context
  - After the analysis: Yes (static approximation across all executions)
- Distinction between caller's effect on callee and callee's effect on caller

Tutorial Problem #1

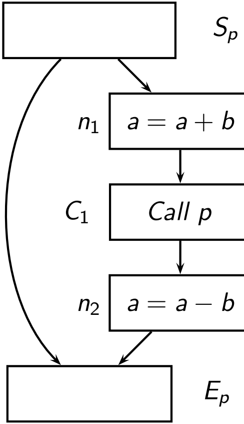
Perform interprocedural live variables analysis for the following program

<pre>main()
{
    p();
}</pre>
	<pre>p()
	{ while (c < 10)
	   { p();
	      a = a*b;
	   }
	}</pre>

Tutorial Problem #2: Summary Flow Function for Constant Propagation

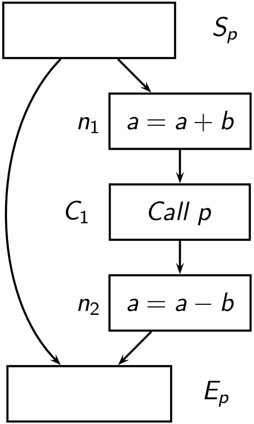


Tutorial Problem #2: Summary Flow Function for Constant Propagation



	Iter. #1	Iter. #2
$[\Phi_p(S_p)](\langle v_a, v_b\rangle)$	$\langle v_a, v_b\rangle$	$\langle v_a, v_b\rangle$
$[\Phi_p(n_1)](\langle v_a, v_b\rangle)$	$\langle v_a + v_b, v_b\rangle$	$\langle v_a + v_b, v_b\rangle$
$[\Phi_p(C_1)](\langle v_a, v_b\rangle)$	$\langle \widehat{\top}, \widehat{\top}\rangle$	$\langle v_a + v_b, v_b\rangle$
$[\Phi_p(n_2)](\langle v_a, v_b\rangle)$	$\langle \widehat{\top}, \widehat{\top}\rangle$	$\langle v_a, v_b\rangle$
$[\Phi_p(E_p)](\langle v_a, v_b\rangle)$	$\langle v_a, v_b\rangle$	$\langle v_a, v_b\rangle$
$f_p(\langle v_a, v_b\rangle)$	$\langle v_a, v_b\rangle$	$\langle v_a, v_b\rangle$

Tutorial Problem #2: Summary Flow Function for Constant Propagation



	Iter. #1	Iter. #2
$[\Phi_p(S_p)](\langle v_a, v_b \rangle)$	$\langle v_a, v_b \rangle$	$\langle v_a, v_b \rangle$
$[\Phi_p(n_1)](\langle v_a, v_b \rangle)$	$\langle v_a + v_b, v_b \rangle$	$\langle v_a + v_b, v_b \rangle$
$[\Phi_p(C_1)](\langle v_a, v_b \rangle)$	$\langle \widehat{\top}, \widehat{\top} \rangle$	$\langle v_a + v_b, v_b \rangle$
$[\Phi_p(n_2)](\langle v_a, v_b \rangle)$	$\langle \widehat{\top}, \widehat{\top} \rangle$	$\langle v_a, v_b \rangle$
$[\Phi_p(E_p)](\langle v_a, v_b \rangle)$	$\langle v_a, v_b \rangle$	$\langle v_a, v_b \rangle$
$f_p(\langle v_a, v_b \rangle)$	$\langle v_a, v_b \rangle$	$\langle v_a, v_b \rangle$

Will this work always?

Tutorial Problem #3

- Is `a*b` available on line 18? Line 6?
- Perform available expressions analysis by constructing the summary flow function for procedure `p`

```
1.   main()
2.   {
3.          c = a*b;
4.          p();
5.          a = a*b;
6.   }
```

```
7.   p()
8.   {    if (...)
9.        {    a = a*b;
10.            p();
11.       }
12.       else if (...)
13.       {    c = a * b;
14.            p();
15.            c = a;
16.       }
17.       else
18.            ; /* ignore */
19.  }
```

Enumeration Based Functional 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

Enumeration Based Functional 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 4

Bottom-up Summaries for Points-to Analysis

Limitations of Functional Approach to Interprocedural Data Flow Analysis

- Problems with constructing summary flow functions
  - Reducing expressions defining flow functions may not be possible in the presence of dependent parts
  - May work for some instances of some problems but not for all
- Hence, $\text{Gen}_n/\text{Kill}_n$ for pointer analysis and constant propagation are defined for individual statements instead of basic blocks of multiple statements

Summarizing a Procedure for Points-to Analysis

A flow-sensitive analysis requires control flow to be recorded between memory updates that share data dependence

Summarizing a Procedure for Points-to Analysis

A flow-sensitive analysis requires control flow to be recorded between memory updates that share data dependence

- Data dependence exists $\Rightarrow$

  Can be eliminated and the

  Control flow between the updates becomes redundant

1. $x = \&a;$
2. $y = x;$

Summarizing a Procedure for Points-to Analysis

A flow-sensitive analysis requires control flow to be recorded between memory updates that share data dependence

- Data dependence exists $\Rightarrow$
  Can be eliminated and the
  Control flow between the updates becomes redundant

1. $x = \&a;$
2. $y = x;$

$\Downarrow$

$x = \&a;$
$y = \&a;$

Summarizing a Procedure for Points-to Analysis

A flow-sensitive analysis requires control flow to be recorded between memory updates that share data dependence

- Data dependence exists ⇒
  Can be eliminated and the
  Control flow between the updates becomes redundant
- Data dependence does not exist ⇒
  Redundant memory updates can be eliminated
  Control flow between the updates is redundant

1. $x = \&a;$
2. $y = \&b;$
3. $x = \&b;$

Summarizing a Procedure for Points-to Analysis

A flow-sensitive analysis requires control flow to be recorded between memory updates that share data dependence

- Data dependence exists $\Rightarrow$
  Can be eliminated and the
  Control flow between the updates becomes redundant
- Data dependence does not exist $\Rightarrow$
  Redundant memory updates can be eliminated
  Control flow between the updates is redundant

$$\boxed{\begin{array}{l} 1.\ x = \&a; \\ 2.\ y = \&b; \\ 3.\ x = \&b; \end{array}}$$

$$\Downarrow$$

$$\boxed{\begin{array}{l} y = \&b; \\ x = \&b; \end{array}}$$

Summarizing a Procedure for Points-to Analysis

A flow-sensitive analysis requires control flow to be recorded between memory updates that share data dependence

- Data dependence exists $\Rightarrow$
  Can be eliminated and the
  Control flow between the updates becomes redundant
- Data dependence does not exist $\Rightarrow$
  Redundant memory updates can be eliminated
  Control flow between the updates is redundant
- Data dependence is unknown $\Rightarrow$
  More information is required (available after inlining in callers)

1. $y = \&b;$
2. $*x = \&a;$
3. $z = y;$

Summarizing a Procedure for Points-to Analysis

A flow-sensitive analysis requires control flow to be recorded between memory updates that share data dependence

1. $y = \&b;$
2. $*x = \&a;$
3. $z = y;$

- Data dependence exists ⇒
  Can be eliminated and the
  Control flow between the updates becomes redundant
- Data dependence does not exist ⇒
  Redundant memory updates can be eliminated
  Control flow between the updates is redundant
- Data dependence is unknown ⇒
  More information is required (available after inlining in callers)
  - Control flow between the updates required because some pointers have definitions in the callers

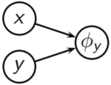
Bottom-up Procedure Summaries for Points-to Analysis

Accesses of pointees defined in the callers are represented using placeholders

Bottom-up Procedure Summaries for Points-to Analysis

Accesses of pointees defined in the callers are represented using placeholders

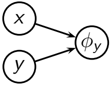
e.g., $x = y \Rightarrow$



ϕ_y is a placeholder

Bottom-up Procedure Summaries for Points-to Analysis

Accesses of pointees defined in the callers are represented using placeholders

e.g., $x = y \Rightarrow$ 

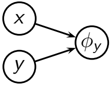
ϕ_y is a placeholder

- Context-based summaries [Zhang-PLDI-14, Wilson-PLDI-95]
  - Use aliases present in the caller
  - Construct a collection of partial transfer functions (PTFs)

Bottom-up Procedure Summaries for Points-to Analysis

Accesses of pointees defined in the callers are represented using placeholders

e.g., $x = y \Rightarrow$



ϕ_y is a placeholder

- Context-based summaries [Zhang-PLDI-14, Wilson-PLDI-95]
  - Use aliases present in the caller
  - Construct a collection of partial transfer functions (PTFs)
- Context-independent summaries [Sălcianu-VMCAI-05, Madhavan-SAS-12]
  - No aliases assumed in the calling contexts
  - Construct a single procedure summary

Placeholder-based Bottom-up Procedure Summaries for Points-to Analysis

Procedure

1. $y = \&b;$
2. $*x = \&a;$
3. $z = y;$

Placeholder-based Bottom-up Procedure Summaries for Points-to Analysis

Procedure

1. $y = \&b;$
2. $*x = \&a;$
3. $z = y;$

Context-based summary

Aliasing pattern	Corresponding partial transfer function
$x \stackrel{\circ}{=} \&y \wedge x \stackrel{\circ}{\neq} \&z$	$y = \&a \;\|\|\; z = \&a$
$x \stackrel{\circ}{\neq} \&y \wedge x \stackrel{\circ}{=} \&z$	$y = \&b \;\|\|\; z = \&b$
$x \stackrel{\circ}{=} \&y \wedge x \stackrel{\circ}{=} \&z$	$y = \&b \;\|\|\; y = \&a \;\|\|\; z = \&a \;\|\|\; z = \&b$
$x \stackrel{\circ}{\neq} \&y \wedge x \stackrel{\circ}{\neq} \&z$	$y = \&b \;\|\|\; \phi_1 = \&a \;\|\|\; z = \&b$

ϕ_1 is a placeholder for a pointee of x whereas ϕ_2 is a placeholder for a pointee of y

Parallel assignments are separated by "$||$" and the placeholders used in them are instantiated before any write

Sequential assignments are separated by ";" and the placeholders used in them are instantiated separately for each assignment

Placeholder-based Bottom-up Procedure Summaries for Points-to Analysis

Procedure	Context-independent summary
1. $y = \&b;$ 2. $*x = \&a;$ 3. $z = y;$	**Flow-sensitive** $y = \&b\ ;\ \phi_1 = \&a\ ;\ z = \&\phi_2$ **Flow-insensitive** $y = \&b \mid\mid \phi_1 = \&a \mid\mid z = \&a \mid\mid z = \&b$

ϕ_1 is a placeholder for a pointee of x whereas ϕ_2 is a placeholder for a pointee of y

Parallel assignments are separated by "$||$" and the placeholders used in them are instantiated before any write

Sequential assignments are separated by ";" and the placeholders used in them are instantiated separately for each assignment

Limitations of Placeholders

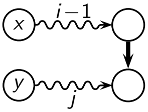
- Placeholders explicate the pointees defined in callers
- Placeholders create a low level abstraction of memory
  (Every assignment in the summary is of the form "$u = \&v$")
- This results in
  - either multiple call-specific procedure summaries, or
  - large summaries

Representing Basic Pointer Assignments using the Generalized Points-to Updates

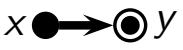
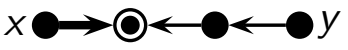
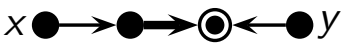
General Case	Specific Examples

General Case

GPU $x \xrightarrow[s]{i|j} y$



Specific Examples

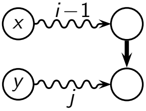
Pointer assignment	GPU	Relevant memory graph after the assignment
$s: x = \&y$	$x \xrightarrow[s]{1\|0} y$	
$s: x = y$	$x \xrightarrow[s]{1\|1} y$	
$s: x = *y$	$x \xrightarrow[s]{1\|2} y$	
$s: *x = y$	$x \xrightarrow[s]{2\|1} y$	

Representing Basic Pointer Assignments using the Generalized Points-to Updates

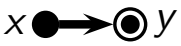
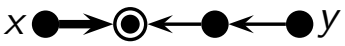
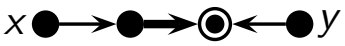
General Case	Specific Examples

General Case

GPU $x \xrightarrow[s]{i|j} y$



Specific Examples

Pointer assignment	GPU	Relevant memory graph after the assignment
$s: x = \&y$	$x \xrightarrow[s]{1\|0} y$	
$s: x = y$	$x \xrightarrow[s]{1\|1} y$	
$s: x = *y$	$x \xrightarrow[s]{1\|2} y$	
$s: *x = y$	$x \xrightarrow[s]{2\|1} y$	

- The direction in a GPU is to distinguish between what is being defined to what is being read
- For pointer analysis, case $i = 0$ does not exist
- classical points-to update is a special case of generalized points-to update with $i = 1$ and $j = 0$

Comparing Bottom-up Procedure Summaries

Procedure

1. $y = \&b;$
2. $*x = \&a;$
3. $z = y;$

Comparing Bottom-up Procedure Summaries

Procedure

1. $y = \&b;$
2. $*x = \&a;$
3. $z = y;$

Context-based

Aliasing pattern	Corresponding partial transfer function
$x \stackrel{\circ}{=} \&y \wedge x \stackrel{\circ}{\neq} \&z$	$y = \&a \mid\mid z = \&a$
$x \stackrel{\circ}{\neq} \&y \wedge x \stackrel{\circ}{=} \&z$	$y = \&b \mid\mid z = \&b$
$x \stackrel{\circ}{=} \&y \wedge x \stackrel{\circ}{=} \&z$	$y = \&b \mid\mid y = \&a \mid\mid z = \&a \mid\mid z = \&b$
$x \stackrel{\circ}{\neq} \&y \wedge x \stackrel{\circ}{\neq} \&z$	$y = \&b \mid\mid \phi_1 = \&a \mid\mid z = \&b$

Context-independent

Placeholder-based flow-sensitive

$y = \&b \; ; \; \phi_1 = \&a \; ; \; z = \&\phi_2$

Placeholder-based flow-insensitive

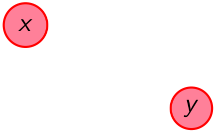
$y = \&b \mid\mid \phi_1 = \&a \mid\mid z = \&a \mid\mid z = \&b$

GPU-based

$y \xrightarrow[1]{1|0} b \, ; x \xrightarrow[2]{2|0} a \, ; z \xrightarrow[3]{1|1} y$

Classical Points-to Updates: A Low Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```

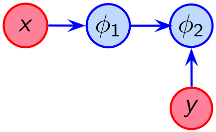


All variables are global

Red nodes are known named locations

Classical Points-to Updates: A Low Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```



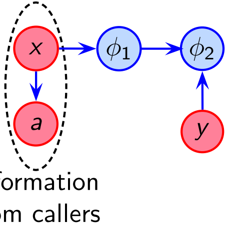
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

Classical Points-to Updates: A Low Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```



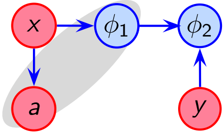
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

Classical Points-to Updates: A Low Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```



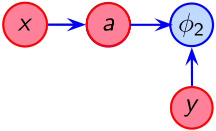
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

Classical Points-to Updates: A Low Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```



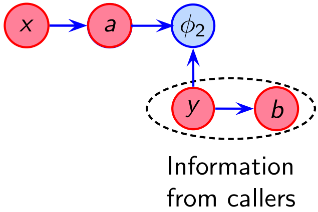
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

Classical Points-to Updates: A Low Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```



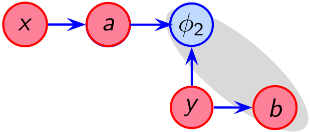
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

Classical Points-to Updates: A Low Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```



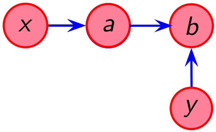
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

Classical Points-to Updates: A Low Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```



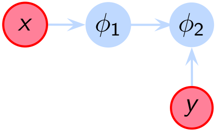
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

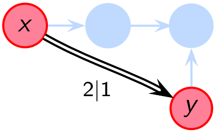
```
f()
{
  *x = y
}
```



Blue arrows are low level view of memory in terms of classical points-to facts

Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```

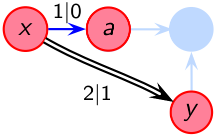


Blue arrows are low level view of memory in terms of classical points-to facts

Black arrows are high level view of memory in terms of generalized points-to facts

Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

```
f()
{
   *x = y
}
```

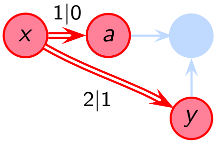


Blue arrows are low level view of memory in terms of classical points-to facts

Black arrows are high level view of memory in terms of generalized points-to facts

Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```

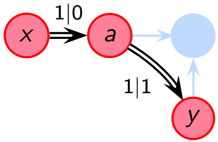


Blue arrows are low level view of memory in terms of classical points-to facts

Black arrows are high level view of memory in terms of generalized points-to facts

Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```

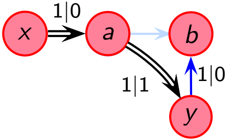


Blue arrows are low level view of memory in terms of classical points-to facts

Black arrows are high level view of memory in terms of generalized points-to facts

Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```

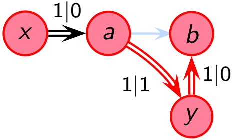


Blue arrows are low level view of memory in terms of classical points-to facts

Black arrows are high level view of memory in terms of generalized points-to facts

Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```

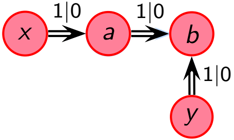


Blue arrows are low level view of memory in terms of classical points-to facts

Black arrows are high level view of memory in terms of generalized points-to facts

Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```

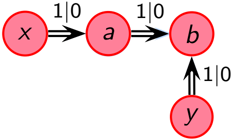


Blue arrows are low level view of memory in terms of classical points-to facts

Black arrows are high level view of memory in terms of generalized points-to facts

Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```



Blue arrows are low level view of memory in terms of classical points-to facts

Black arrows are high level view of memory in terms of generalized points-to facts

This abstraction does not introduce any imprecision over the classical points-to graph

Generalized Points-to Graphs (GPGs) I

A GPG is a graph with

- Nodes are generalized points-to blocks (GPBs)
  - A GPB contains a set of GPUs
- Edges represent control flow between GPBs

Generalized Points-to Graphs (GPGs) I

A GPG is a graph with

- Nodes are generalized points-to blocks (GPBs)
  - A GPB contains a set of GPUs
- Edges represent control flow between GPBs

A GPG is analogous to a CFG of a procedure

$$\begin{tikzcd}
\text{GPG} \arrow[r, Leftrightarrow] \arrow[d] & \text{CFG} \arrow[d] \\
\text{GPB} \arrow[r, Leftrightarrow] \arrow[d] & \text{BB} \arrow[d] \\
\text{GPU} \arrow[r, Leftrightarrow] & \text{Ptr. Assgn.}
\end{tikzcd}$$

Generalized Points-to Graphs (GPGs) I

A GPG

- N
- E

First difference:

- GPUs in a GPB represent parallel assignments
- Assignments in a basic block are sequential

A GPG is analogous to a CFG of a procedure

$$
\begin{tikzcd}
\text{GPG} \arrow[r, Leftrightarrow] \arrow[d] & \text{CFG} \arrow[d] \\
\text{GPB} \arrow[r, Leftrightarrow] \arrow[d] & \text{BB} \arrow[d] \\
\text{GPU} \arrow[r, Leftrightarrow] & \text{Ptr. Assgn.}
\end{tikzcd}
$$

Generalized Points-to Graphs (GPGs) I

A GPG

- N
- E

Second difference:

- CFGs contain call basic blocks
- GPGs do not have call GPBs

A GPG is analogous to a CFG of a procedure

$$\begin{tikzcd}
\text{GPG} \arrow[r, Leftrightarrow] \arrow[d] & \text{CFG} \arrow[d] \\
\text{GPB} \arrow[r, Leftrightarrow] \arrow[d] & \text{BB} \arrow[d] \\
\text{GPU} \arrow[r, Leftrightarrow] & \text{Ptr. Assgn.}
\end{tikzcd}$$

Generalized Points-to Graphs (GPGs) II

Construction of Initial GPGs:

- Non-pointer assignments and condition tests are removed
- Each pointer assignment *s* is transliterated to its GPU
- A separate GPB is created for assignment in the CFG
- GPG edges are induced from the control flow of the CFG
- GPGs contain only variables that are shared across procedures

GPGs then undergo extensive optimizations

The Road Ahead

- Before devising control flow optimizations, we found GPGs with 742 nodes, 377 calls, 59747 edges containing ONLY 2 GPUs!!
- Our explorations in both top-down and bottom-up approaches of interprocedural analysis lead us to observe that

The Road Ahead

- Before devising control flow optimizations, we found GPGs with 742 nodes, 377 calls, 59747 edges containing ONLY 2 GPUs!!
- Our explorations in both top-down and bottom-up approaches of interprocedural analysis lead us to observe that

The real killer of scalability in program analysis is not

- the **data that needs to be computed** but
- the **control flow that it is subjected to** in search of precision

The Road Ahead

- Before devising control flow optimizations, we found GPGs with 742 nodes, 377 calls, 59747 edges containing ONLY 2 GPUs!!
- Our explorations in both top-down and bottom-up approaches of interprocedural analysis lead us to observe that

> The real killer of scalability in program analysis is not
>
> - the **data that needs to be computed** but
> - the **control flow that it is subjected to** in search of precision

- For scaling program analysis, we need to optimize away the part of the control flow that does not contribute to data flow

The Next Holy Grail in Search of Scalability?

Under-approximated Control Flow	Original Control Flow	Over-approximated Control Flow

The Next Holy Grail in Search of Scalability?

Under-approximated Control Flow

Original Control Flow

Over-approximated Control Flow

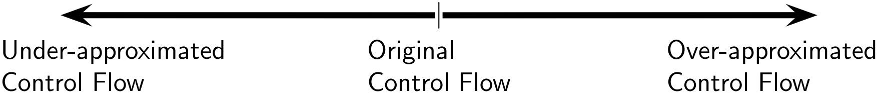
Missing control flow paths

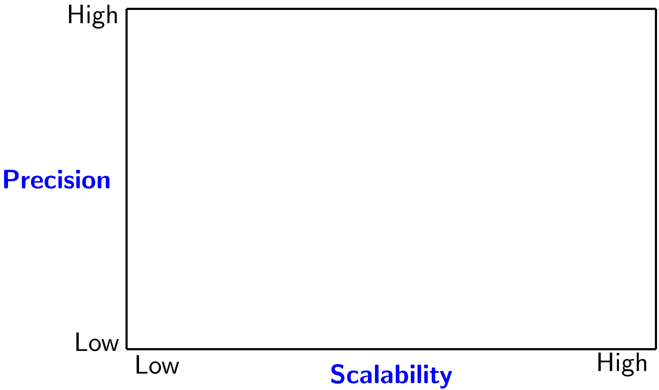
Flow insensitivity
Context insensitivity

The Next Holy Grail in Search of Scalability?

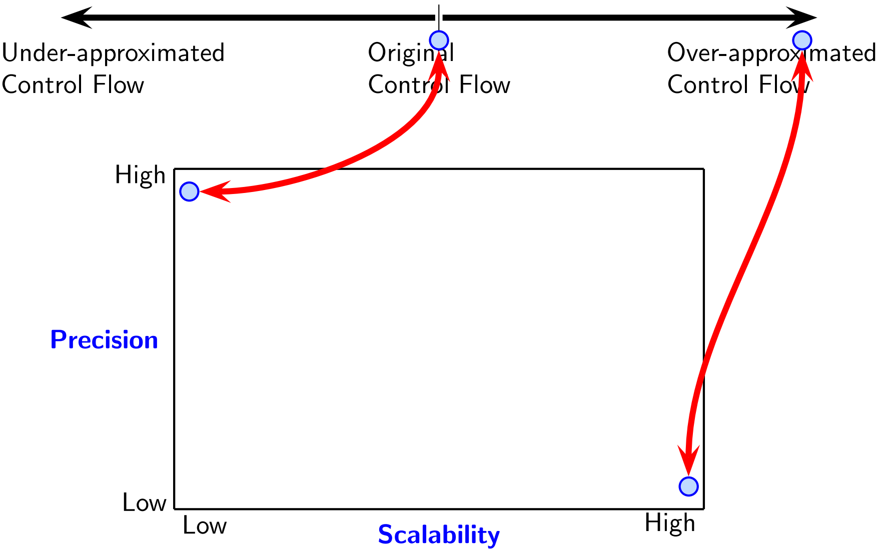
Under-approximated Control Flow	Original Control Flow	Over-approximated Control Flow
Missing control flow paths		Flow insensitivity Context insensitivity
Unsound	Sound and Precise	Sound and Imprecise

The Next Holy Grail in Search of Scalability?

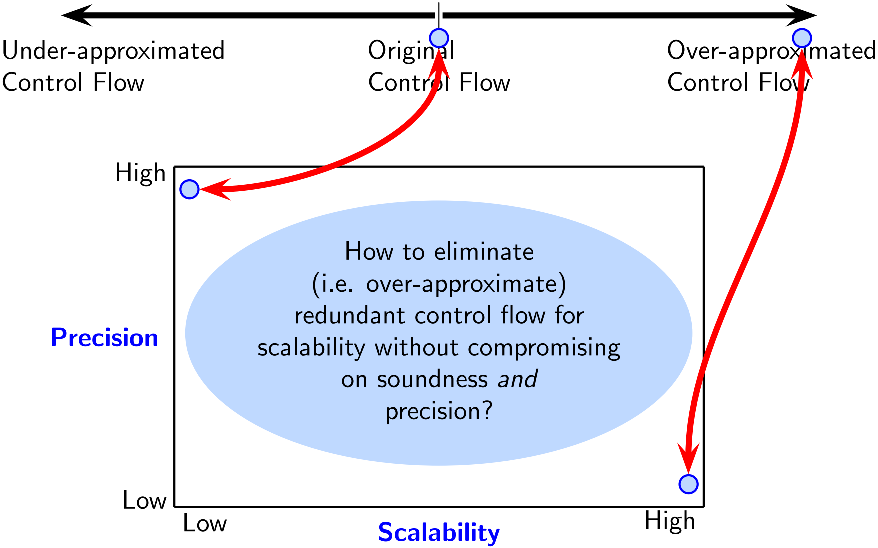




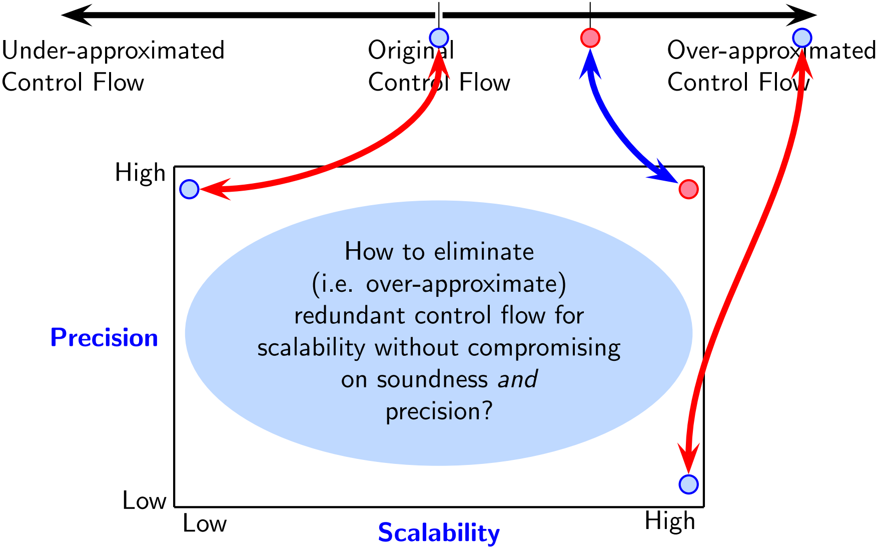
The Next Holy Grail in Search of Scalability?



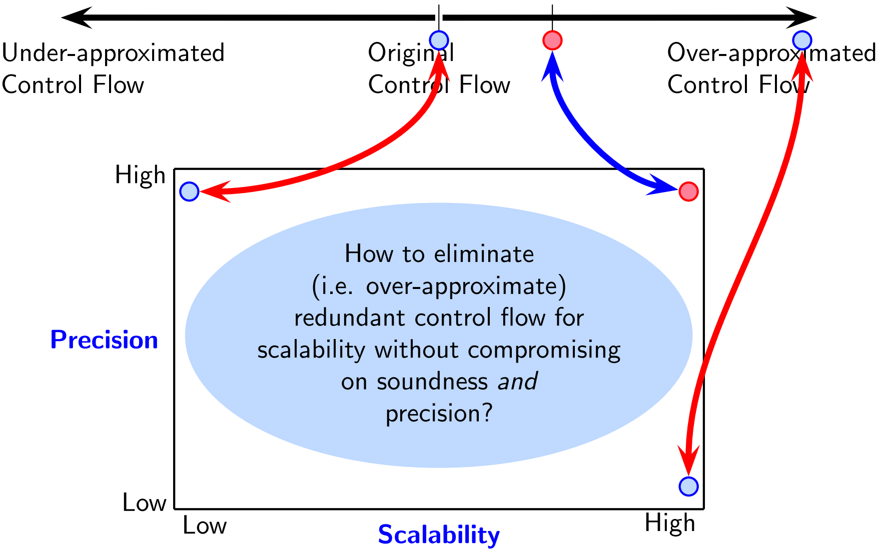
The Next Holy Grail in Search of Scalability?



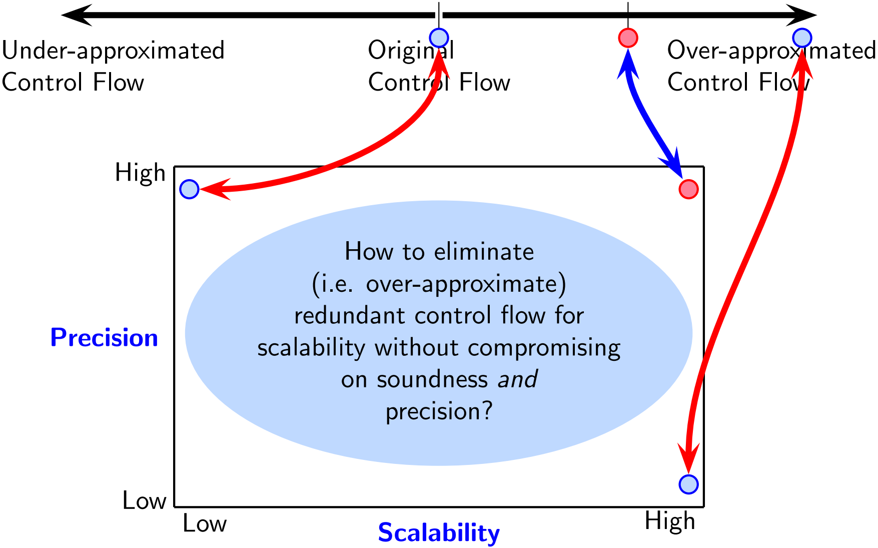
The Next Holy Grail in Search of Scalability?



The Next Holy Grail in Search of Scalability?



The Next Holy Grail in Search of Scalability?



Part 5

Top-Down Interprocedural Analysis

Formalizing Context-Sensitive Analyses

- Context-sensitive information at a program point is a pair (σ, x) where σ is the context and x is the data flow value
  - A separate value is computed for each context reaching a procedure
  - We leave the context σ and the data flow value x undefined

    σ is defined by the method of performing analysis

    x is defined by the analysis to be performed
- Examples of contexts
  - A call chain (i.e. a sequence of unfinished calls) reaching a procedure
  - A data flow reaching the start of a procedure

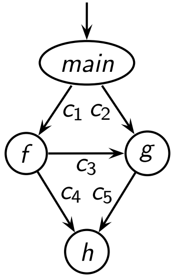
Context-Sensitive Call Graph for Context-Sensitive Methods

Context-sensitive call graph

- Derived out of a call graph
- Each procedure is cloned for each context reaching it (requires the number of contexts to be finite)
- Semantically equivalent to the original call graph
- A useful tool for explaining context-sensitive methods

Context-Sensitive Call Graph

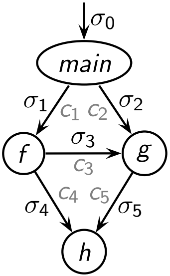
Example call graph	Corresponding context-sensitive call graph



Call	Call site	Context
main calls *f*	c_1	
main calls *g*	c_2	
f calls *g*	c_3	
f calls *h*	c_4	
g calls *h*	c_5	

Context-Sensitive Call Graph

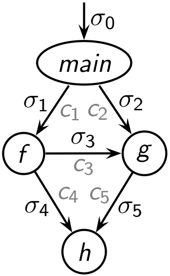
Example call graph	Corresponding context-sensitive call graph

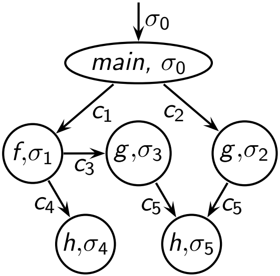


Call	Call site	Context
main calls *f*	c_1	σ_1
main calls *g*	c_2	σ_2
f calls *g*	c_3	σ_3
f calls *h*	c_4	σ_4
g calls *h*	c_5	σ_5

Context-Sensitive Call Graph

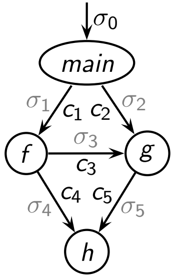
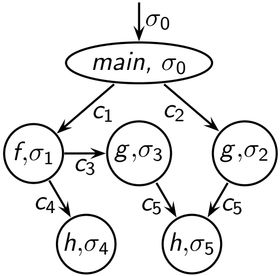
Example call graph	Corresponding context-sensitive call graph





Call	Call site	Context
main calls *f*	c_1	σ_1
main calls *g*	c_2	σ_2
f calls *g*	c_3	σ_3
f calls *h*	c_4	σ_4
g calls *h*	c_5	σ_5

Context-Sensitive Call Graph

Example call graph	Corresponding context-sensitive call graph
	

$\mathbb{C}$: set of call nodes

Nodes: $\mathbb{N}_{CG}$
Edges: $\mathbb{E}_{CG} = \mathbb{N}_{CG} \times \mathbb{N}_{CG} \times \mathbb{C}$

Σ: set of contexts

Nodes: $\mathbb{N}_{CG} = \mathbb{N}_{CSG} \times \Sigma$
Edges: $\mathbb{E}_{CSG} = \mathbb{N}_{CSG} \times \mathbb{N}_{CSG} \times \mathbb{C}$

A Unified Model of Context-Sensitive Interprocedural Analysis

We formalize context-sensitive interprocedural analysis by defining

- how to compute a context-sensitive call graph
  (can be instantiated for different methods)
- how to compute context-sensitive data flow information using a context-sensitive call graph

Computing Context-Sensitive Call Graph

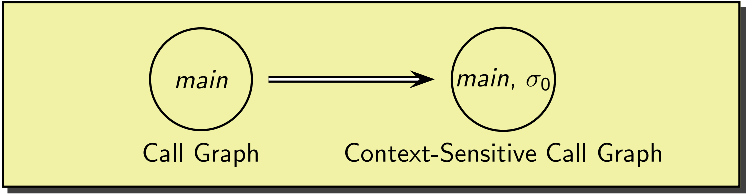
$$\frac{main \in \mathbb{N}_{CG},\ \ \sigma_0 \text{ is initial context}}{(main, \sigma_0) \in \mathbb{N}_{CSG}}$$

$$\frac{p \xrightarrow{C_i} q \in \mathbb{E}_{CG},\ \ (p, \sigma) \in \mathbb{N}_{CSG},\ \ \sigma' \in \textsc{context}(p, C_i, \sigma)}{(q, \sigma') \in \mathbb{N}_{CSG},\ \ (p, \sigma) \xrightarrow{C_i} (q, \sigma') \in \mathbb{E}_{CSG}}$$

Function CONTEXT computes context for a callee procedure q using the information from call node C_i in the caller procedure p in context σ

Computing Context-Sensitive Call Graph

$$\frac{main \in \mathbb{N}_{CG}, \;\; \sigma_0 \text{ is initial context}}{(main, \sigma_0) \in \mathbb{N}_{CSG}}$$



Function CONTEXT computes context for a callee procedure q using the information from call node C_i in the caller procedure p in context σ

Computing Context-Sensitive Call Graph

$$\frac{main \in \mathbb{N}_{CG}, \;\; \sigma_0 \text{ is initial context}}{(main, \sigma_0) \in \mathbb{N}_{CSG}}$$

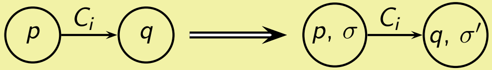
$$\frac{p \xrightarrow{C_i} q \in \mathbb{E}_{CG}, \;\; (p, \sigma) \in \mathbb{N}_{CSG}, \;\; \sigma' \in \text{CONTEXT}(p, C_i, \sigma)}{(q, \sigma') \in \mathbb{N}_{CSG}, \;\; (p, \sigma) \xrightarrow{C_i} (q, \sigma') \in \mathbb{E}_{CSG}}$$

Function CONTEXT computes context for a callee procedure q using the information from call node C_i in the caller procedure p in context σ

Computing Context-Sensitive Call Graph

$$\frac{main \in \mathbb{N}_{CG},\ \ \sigma_0 \text{ is initial context}}{(main, \sigma_0) \in \mathbb{N}_{CSG}}$$

$$\frac{p \xrightarrow{C_i} q \in \mathbb{E}_{CG},\ \ (p,\sigma) \in \mathbb{N}_{CSG},\ \ \sigma' \in \text{CONTEXT}(p, C_i, \sigma)}{(q,\sigma') \in \mathbb{N}_{CSG},\ \ (p,\sigma) \xrightarrow{C_i} (q,\sigma') \in \mathbb{E}_{CSG}}$$



Call Graph   Context-Sensitive Call Graph

Function CONTEXT computes context for a callee procedure q using the information from call node C_i in the caller procedure p in context σ

Computing Context-Sensitive Call Graph

$$\frac{main \in \mathbb{N}_{CG},\ \ \sigma_0 \text{ is initial context}}{(main, \sigma_0) \in \mathbb{N}_{CSG}}$$

$$\frac{p \xrightarrow{C_i} q \in \mathbb{E}_{CG},\ \ (p, \sigma) \in \mathbb{N}_{CSG},\ \ \sigma' \in \text{CONTEXT}(p, C_i, \sigma)}{(q, \sigma') \in \mathbb{N}_{CSG},\ \ (p, \sigma) \xrightarrow{C_i} (q, \sigma') \in \mathbb{E}_{CSG}}$$

Function CONTEXT computes context for a callee procedure q using the information from call node C_i in the caller procedure p in context σ

Computing Data Flow Information Using CSG (1)

Data flow value		Effect
In_n	n is S_{main}	boundary information for the entire program
	n is S_p for a callee p	interprocedural effect of call edge $c_i \rightarrow (S_p \equiv n)$
	n is any other node	intraprocedural effect
Out_n	n is a call node	interprocedural effect of return edge $E_p \rightarrow (c_i \equiv n)$
	n is any other node	intraprocedural effect

Computing Data Flow Information Using CSG (2)

$$In_n^p = \begin{cases} \left\{ (\sigma_0, BI) \right\} & n \text{ is } S_{main} \\ \left\{ (\sigma, x) \;\middle|\; (q, \sigma') \xrightarrow{C_i} (p, \sigma) \in \mathbb{E}_{CSG},\ x = In^q_{C_i}(\sigma') \right\} & n \text{ is } S_p \\ \left\{ (\sigma, x) \;\middle|\; (p, \sigma) \in \mathbb{N}_{CSG},\ x = \displaystyle\bigsqcap_{(p,m) \in pred(p,n)} Out^p_m(\sigma) \right\} & \text{otherwise} \end{cases}$$

$$Out_n^p = \begin{cases} \left\{ (\sigma, x) \;\middle|\; (p, \sigma) \xrightarrow{n} (q, \sigma') \in \mathbb{E}_{CSG},\ x = Out^q_{E_q}(\sigma') \right\} & n \text{ is } C_i \\ \left\{ (\sigma, x) \;\middle|\; (p, \sigma) \in \mathbb{N}_{CSG},\ x = \gamma_n^p\left(In_n^p(\sigma) \right) \right\} & \text{otherwise} \end{cases}$$

Computing Data Flow Information Using CSG (2)

$In_n^p = \begin{cases} \{(\sigma_0, B \\ \{(\sigma, x) \\ \{(\sigma, x) \end{cases}$ ain ... ise

$Out_n^p = \begin{cases} \{(\sigma, x) \\ \{(\sigma, x) \end{cases}$ ... ise

- In_n^p and Out_n^p represent the data flow values for node n in procedure p
- σ represents a context

  Its rules of computation are defined by a specific method (will be defined later)
- x represents a data flow value
- $In_n^p(\sigma)$ and $Out_n^p(\sigma)$ select the data flow value for a particular context σ

  They return $\top$ if no value for σ is found
- γ_n^p is the forward node flow function for node n of procedure p

Computing Data Flow Information Using CSG (2)

$$In_n^p = \begin{cases} \boxed{\{(\sigma_0, BI)\} \qquad n \text{ is } S_{main}} \\ \{(\sigma, x) \mid (q, \sigma' \ldots & n \text{ is } S_p \\ \{(\sigma, x) \mid (p, \sigma) \ldots & \text{otherwise} \end{cases}$$

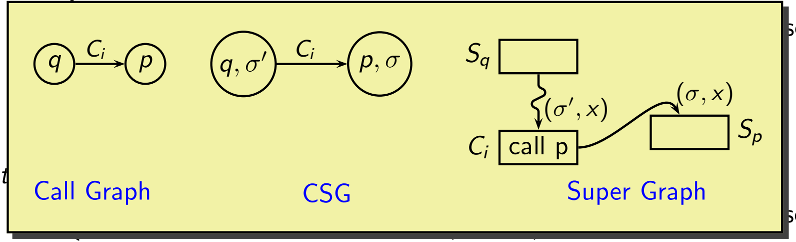
- σ_0 represents the initial context reaching *main*
- BI represents the boundary information of *main*

$$Out_n^p = \begin{cases} \{(\sigma, x) \mid (p, \sigma) \ldots \in E_{CSG},\ x = Out_{E_q}(\sigma') \} & n \text{ is } C_i \\ \{(\sigma, x) \mid (p, \sigma) \in \mathbb{N}_{CSG},\ x = \gamma_n^p\left(In_n^p(\sigma)\right) \} & \text{otherwise} \end{cases}$$

Computing Data Flow Information Using CSG (2)

$$In_n^p = \begin{cases} \left\{ (\sigma_0, BI) \right\} & n \text{ is } S_{main} \\ \boxed{\left\{ (\sigma, x) \mid (q, \sigma') \xrightarrow{C_i} (p, \sigma) \in \mathbb{E}_{CSG},\ x = In^q_{C_i}(\sigma') \right\} \quad n \text{ is } S_p} \\ \ldots & \ldots\text{se} \end{cases}$$

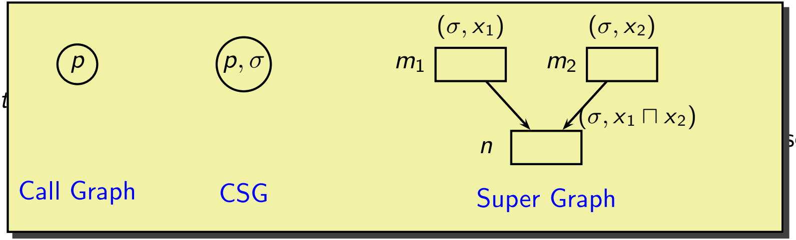
Out ... se



Computing Data Flow Information Using CSG (2)

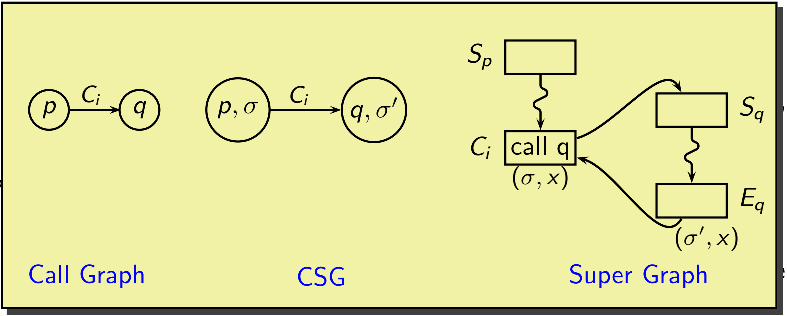
$$
In_n^p = \begin{cases}
\left\{ (\sigma_0, BI) \right\} & n \text{ is } S_{main} \\[2ex]
\left\{ (\sigma, x) \;\middle|\; (q, \sigma') \xrightarrow{C_i} (p, \sigma) \in \mathbb{E}_{CSG},\ x = In^q_{C_i}(\sigma') \right\} & n \text{ is } S_p \\[2ex]
\boxed{\left\{ (\sigma, x) \;\middle|\; (p, \sigma) \in \mathbb{N}_{CSG},\ x = \displaystyle\bigsqcap_{(p,m) \in pred(p,n)} Out^p_m(\sigma) \right\} \quad \text{otherwise}}
\end{cases}
$$

Out ... se



Computing Data Flow Information Using CSG (2)

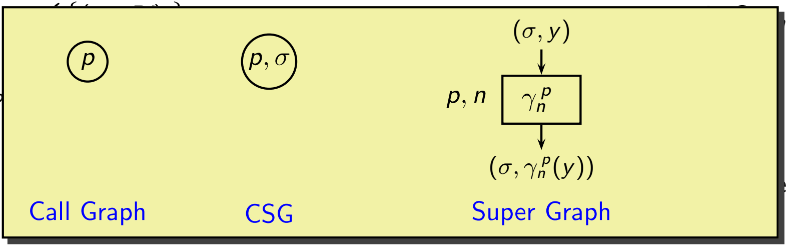
In_n^p



$$Out_n^p = \begin{cases} \boxed{\left\{ (\sigma, x) \;\middle|\; (p,\sigma) \xrightarrow{n} (q, \sigma') \in \mathbb{E}_{CSG},\ x = Out_{E_q}^q(\sigma') \right\} \qquad n \text{ is } C_i} \\ \left\{ (\sigma, x) \;\middle|\; (p,\sigma) \in \mathbb{N}_{CSG},\ x = \gamma_n^p\left(In_n^p(\sigma) \right) \right\} & \text{otherwise} \end{cases}$$

Computing Data Flow Information Using CSG (2)

In_n^p



$$Out_n^p = \begin{cases} \left\{ (\sigma, x) \;\middle|\; (p, \sigma) \xrightarrow{n} (q, \sigma') \in \mathbb{E}_{CSG},\; x = Out_{E_q}^{q}(\sigma') \right\} & n \text{ is } C_i \\ \boxed{\left\{ (\sigma, x) \;\middle|\; (p, \sigma) \in \mathbb{N}_{CSG},\; x = \gamma_n^p\left(In_n^p(\sigma) \right) \right\} \quad \text{otherwise}} \end{cases}$$

Computing Data Flow Information Using CSG (2)

$$In_n^p = \begin{cases} \left\{ (\sigma_0, BI) \right\} & n \text{ is } S_{main} \\ \left\{ (\sigma, x) \mid (q, \sigma') \xrightarrow{C_i} (p, \sigma) \in \mathbb{E}_{CSG},\ x = In_{C_i}^q(\sigma') \right\} & n \text{ is } S_p \\ \left\{ (\sigma, x) \mid (p, \sigma) \in \mathbb{N}_{CSG},\ x = \displaystyle\bigsqcap_{(p,m) \in pred(p,n)} Out_m^p(\sigma) \right\} & \text{otherwise} \end{cases}$$

$$Out_n^p = \begin{cases} \left\{ (\sigma, x) \mid (p, \sigma) \xrightarrow{n} (q, \sigma') \in \mathbb{E}_{CSG},\ x = Out_{E_q}^q(\sigma') \right\} & n \text{ is } C_i \\ \left\{ (\sigma, x) \mid (p, \sigma) \in \mathbb{N}_{CSG},\ x = \gamma_n^p\left(In_n^p(\sigma)\right) \right\} & \text{otherwise} \end{cases}$$

Part 6

Classical Call-Strings Approach

Classical Full Call-Strings Approach

Most general, flow and context sensitive method

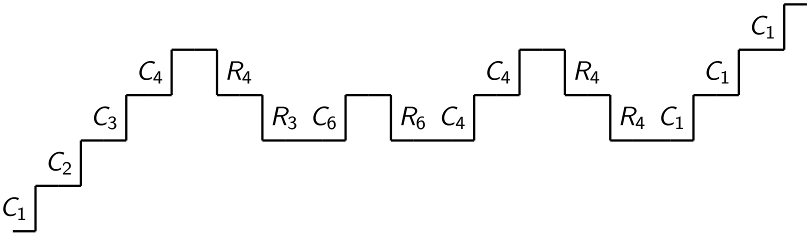
- Remember call history

  Information should be propagated *back* to the correct point

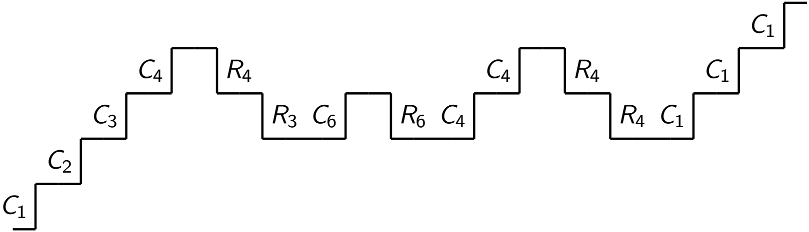
- Call string at a program point:
  - Sequence of *unfinished calls* reaching that point
  - Starting from the S_{main}

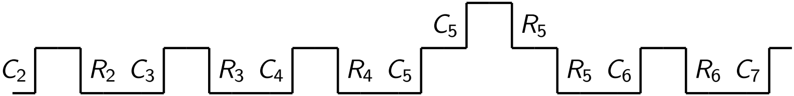
  A snap-shot of call stack in terms of call sites

Interprocedural Validity and Calling Contexts

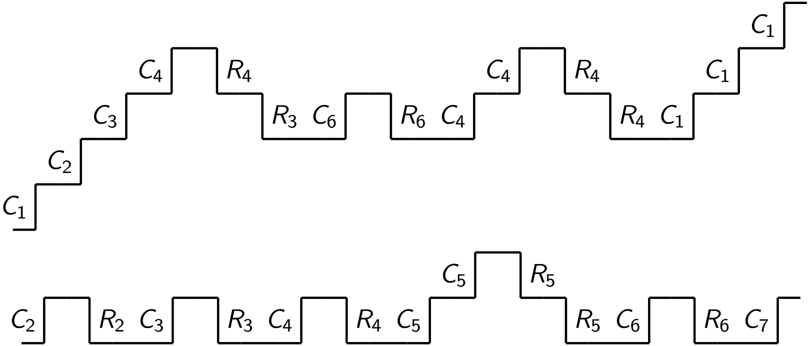


Interprocedural Validity and Calling Contexts



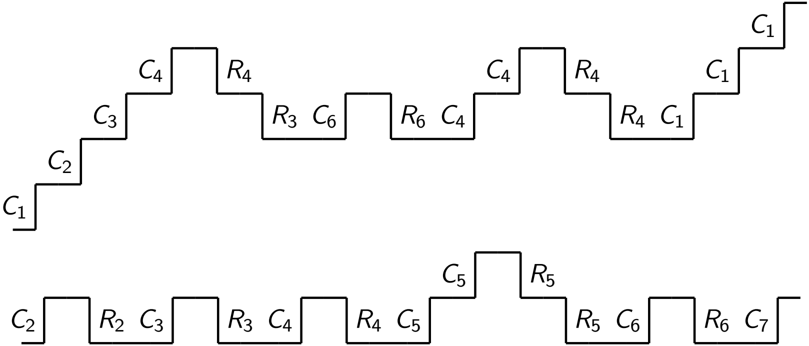


Interprocedural Validity and Calling Contexts



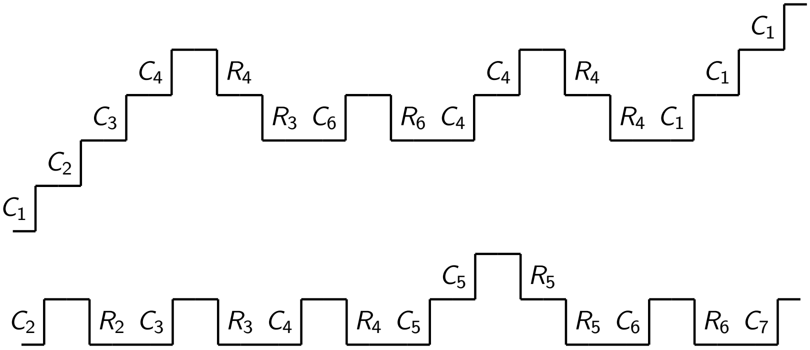
- "You can descend only as much as you have ascended!"

Interprocedural Validity and Calling Contexts



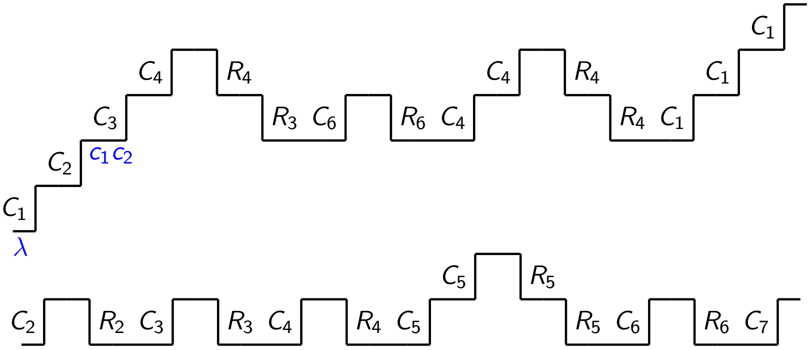
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step

Interprocedural Validity and Calling Contexts



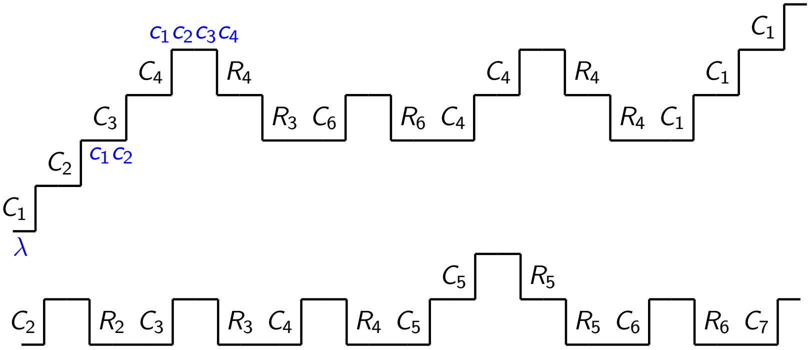
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps

Interprocedural Validity and Calling Contexts



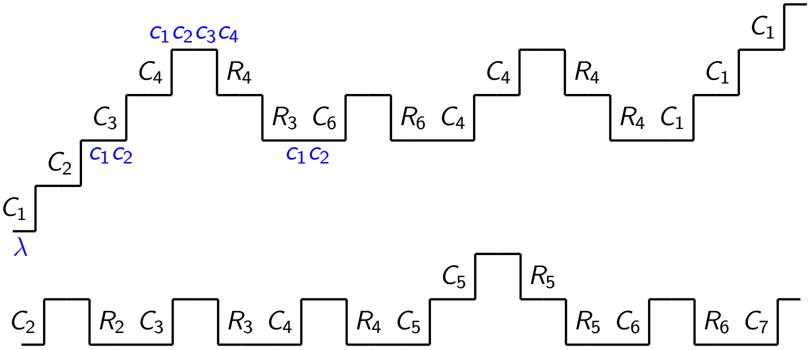
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps

Interprocedural Validity and Calling Contexts



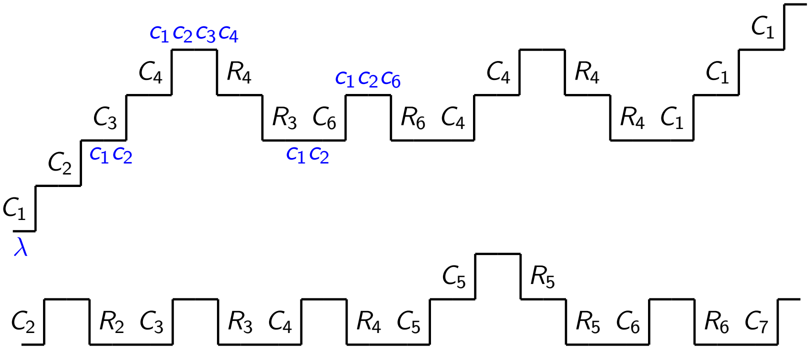
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps

Interprocedural Validity and Calling Contexts



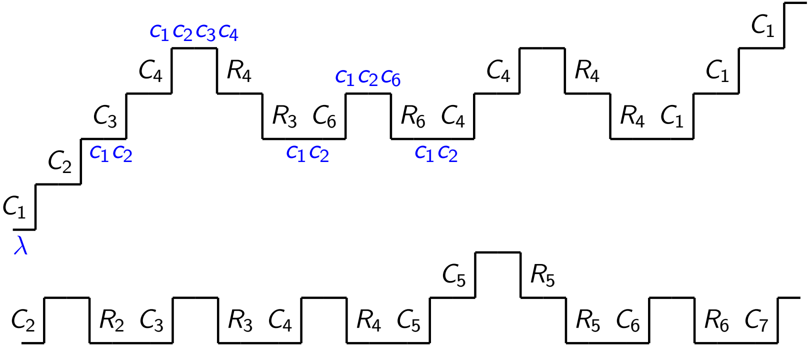
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps

Interprocedural Validity and Calling Contexts



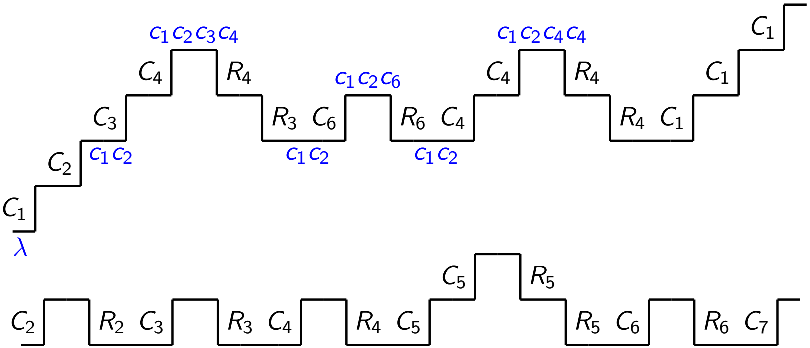
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps

Interprocedural Validity and Calling Contexts



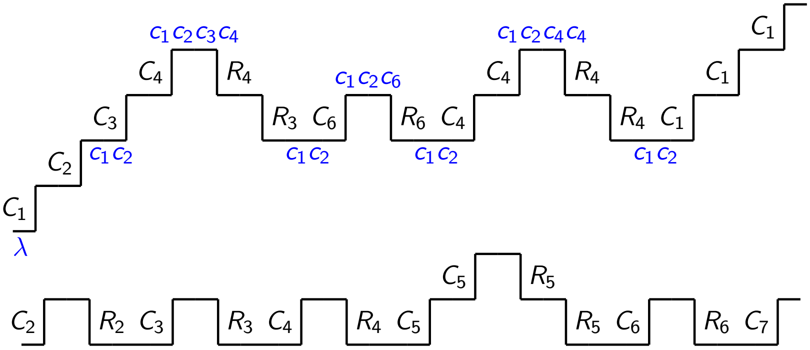
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps

Interprocedural Validity and Calling Contexts



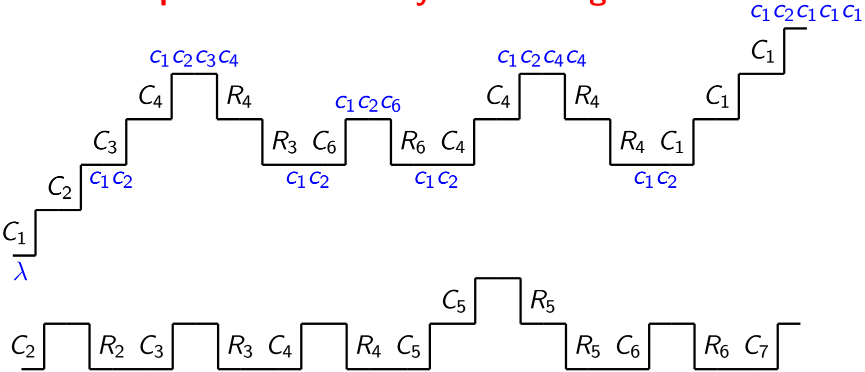
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps

Interprocedural Validity and Calling Contexts



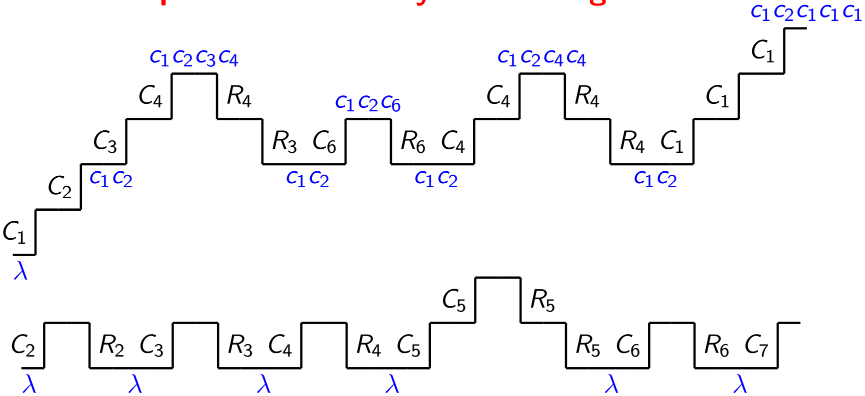
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps

Interprocedural Validity and Calling Contexts



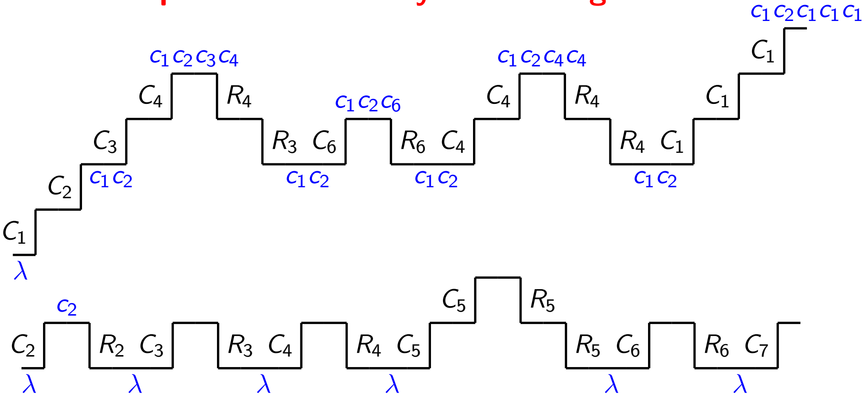
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps

Interprocedural Validity and Calling Contexts



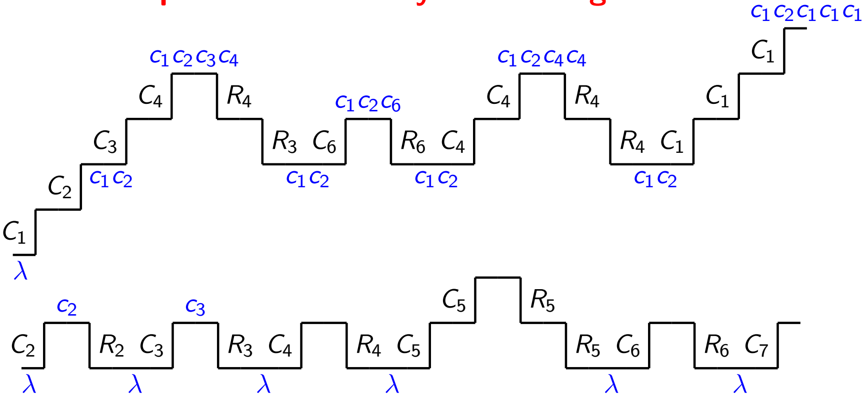
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps

Interprocedural Validity and Calling Contexts



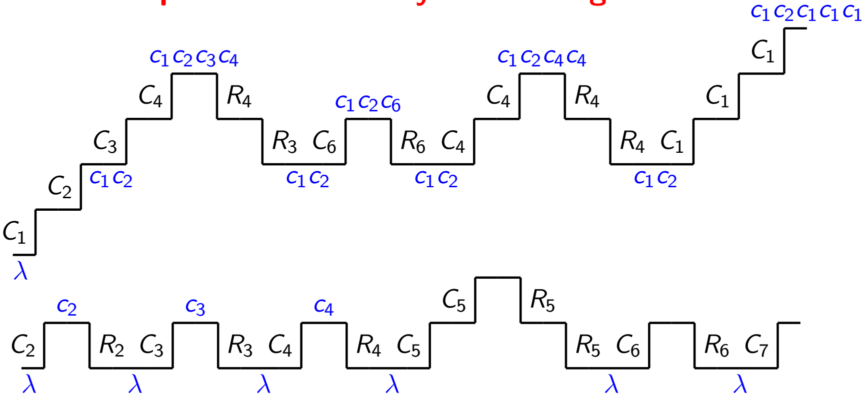
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps

Interprocedural Validity and Calling Contexts



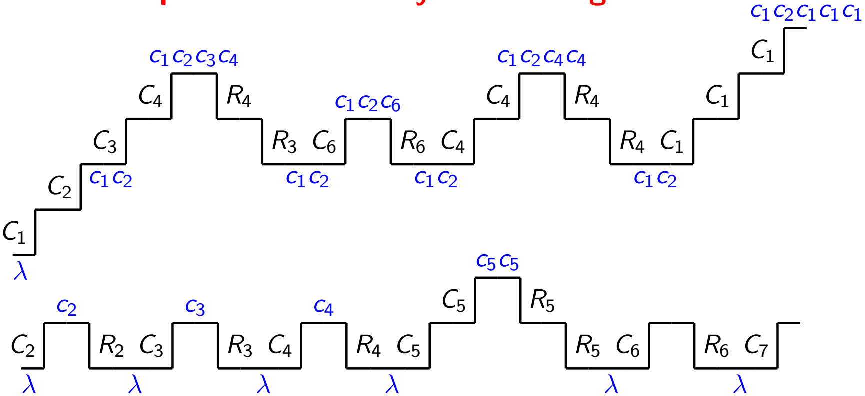
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps

Interprocedural Validity and Calling Contexts



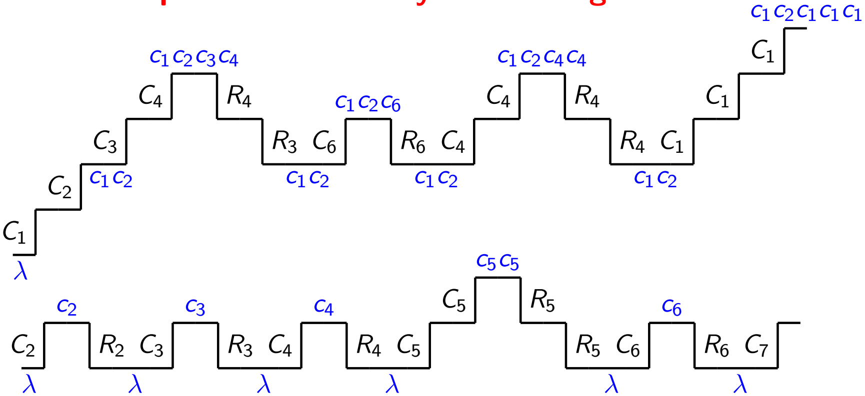
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps

Interprocedural Validity and Calling Contexts



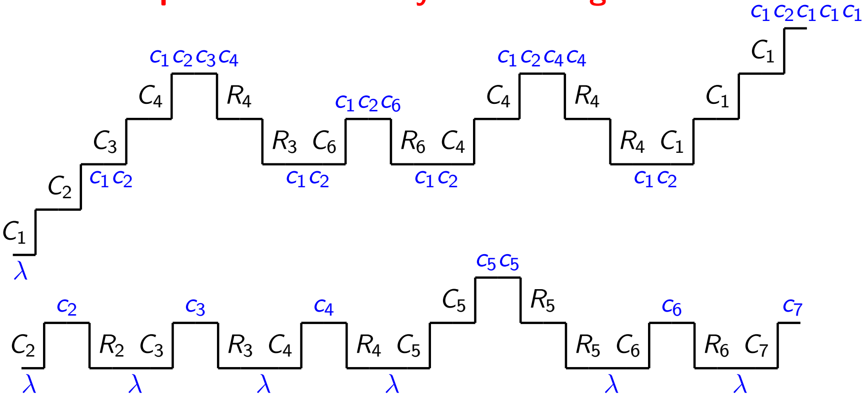
- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps

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

Computing and Using Call Strings

- Call node C_i
  - Append c_i to every σ
  - Propagate the data flow values unchanged
    (modulo parameter mappings)

Computing and Using Call Strings

- Call node C_i
  - Append c_i to every σ
  - Propagate the data flow values unchanged (modulo parameter mappings)
- Return node R_i
  - If the last call site is c_i, remove it and propagate the data flow value unchanged
  - Block other data flow values (this filters out interproceedurally invalid paths)

Computing and Using Call Strings

- Call node C_i
  - Append c_i to every σ
  - Propagate the data flow values unchanged (modulo parameter mappings)

Ascend

- Return node R_i
  - If the last call site is c_i, remove it and propagate the data flow value unchanged
  - Block other data flow values (this filters out interproceedurally invalid paths)

Descend

Instantiating the Unified Model to Call Strings

We need to define three things:

- The initial context σ_0
- We need to define the $\textsc{context}(p, C_i, \sigma)$ function
- We need to ensure that the set of context Σ is finite

Instantiating the Unified Model to Call Strings

We need to define three things:

- The initial context σ_0

  σ_0 is the empty call string λ

- We need to define the $\textsc{context}(p, C_i, \sigma)$ function

  $\textsc{context}(p, C_i, \sigma) = \sigma \cdot c_i$

- We need to ensure that the set of context Σ is finite

  We construct call-strings that contain at most three occurrences of a call site for bit vector frameworks

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

$\Rightarrow$ 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

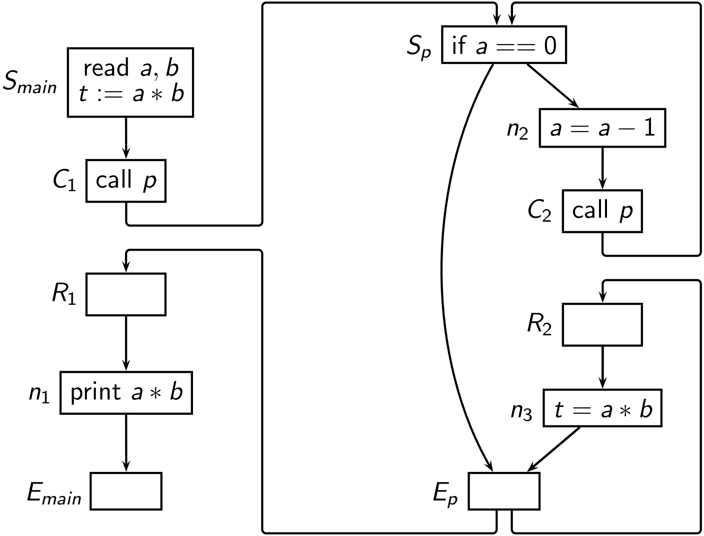
Terminating Call String Construction

- 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

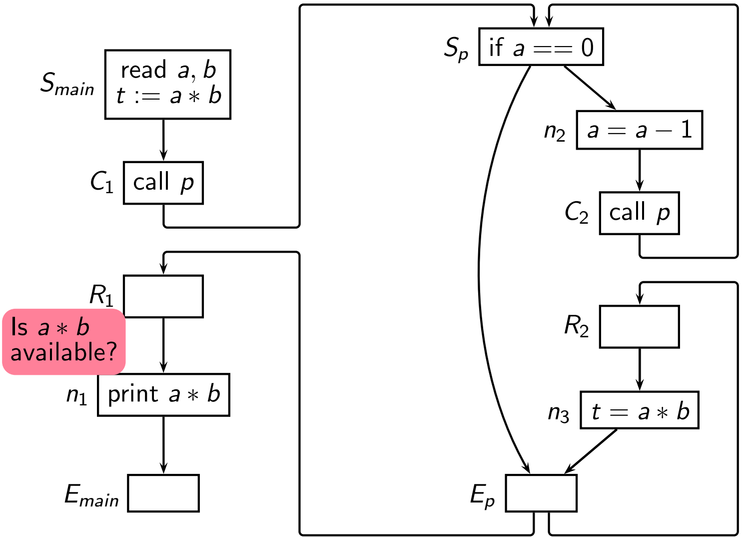
  $\Rightarrow$ Not a bound but prescribed necessary length

  $\Rightarrow$ Large number of long call strings
- A procedure needs to be renalyzed for a call string even if the data flow is same as any other call string

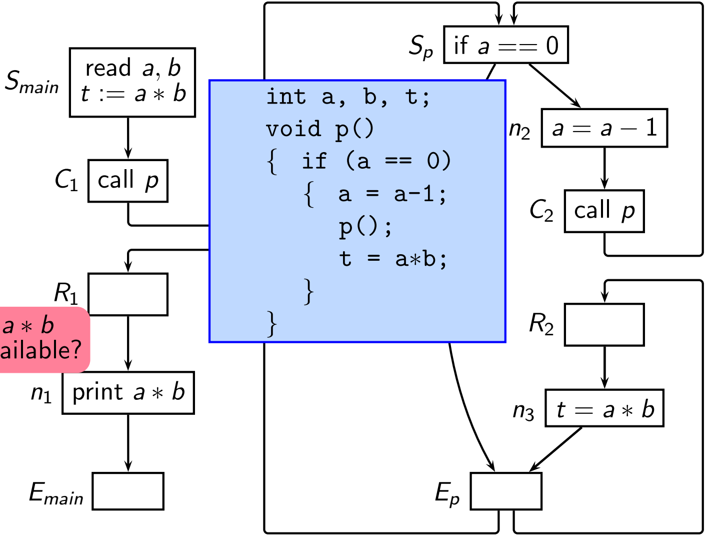
Available Expressions Analysis Using Call Strings (1)



Available Expressions Analysis Using Call Strings (1)

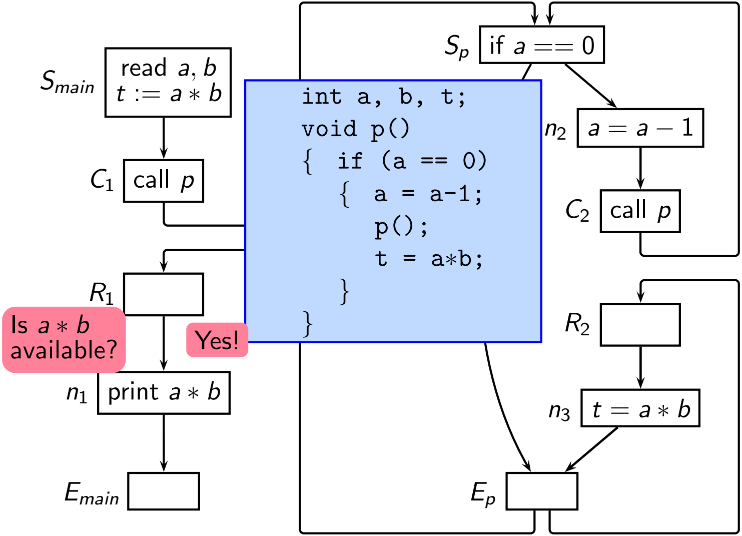


Available Expressions Analysis Using Call Strings (1)



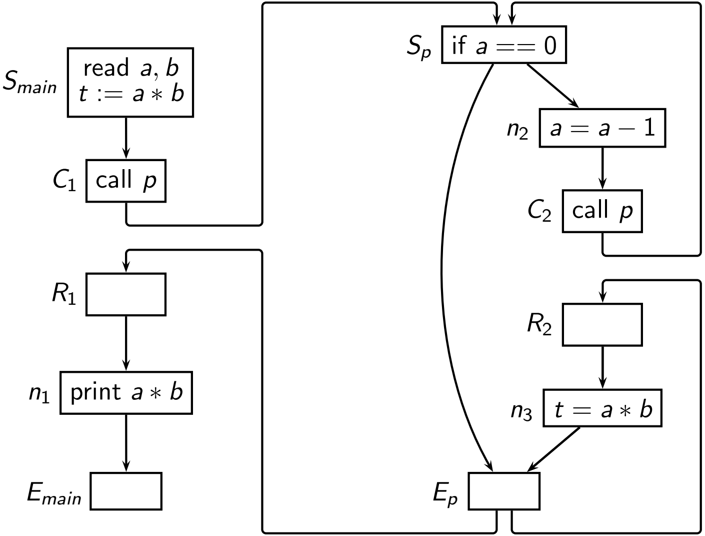

```
int a, b, t;
void p()
{  if (a == 0)
   {  a = a-1;
      p();
      t = a*b;
   }
}
```

Available Expressions Analysis Using Call Strings (1)

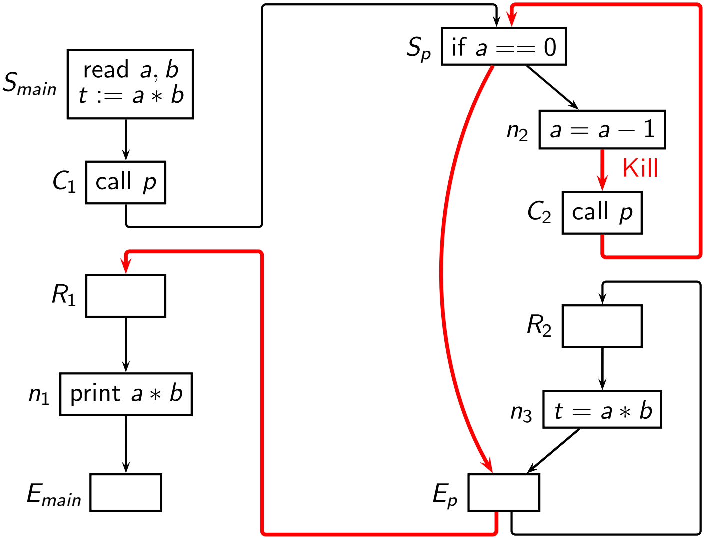



```
int a, b, t;
void p()
{  if (a == 0)
   {  a = a-1;
      p();
      t = a*b;
   }
}
```

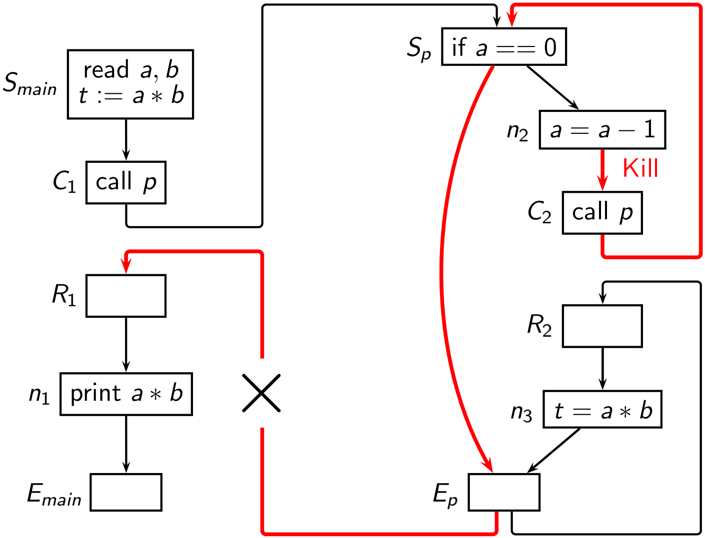
Available Expressions Analysis Using Call Strings (1)



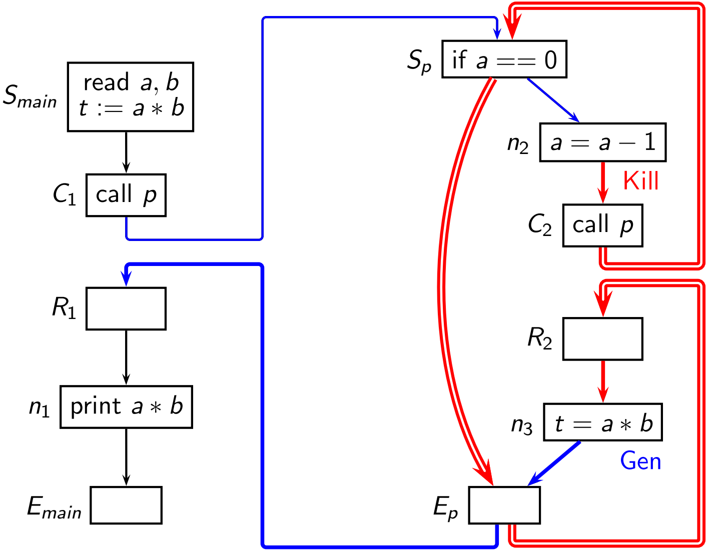
Available Expressions Analysis Using Call Strings (1)



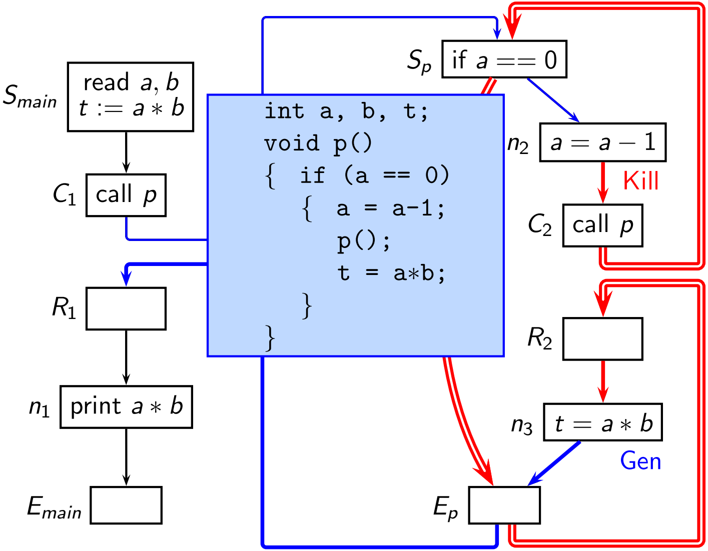
Available Expressions Analysis Using Call Strings (1)



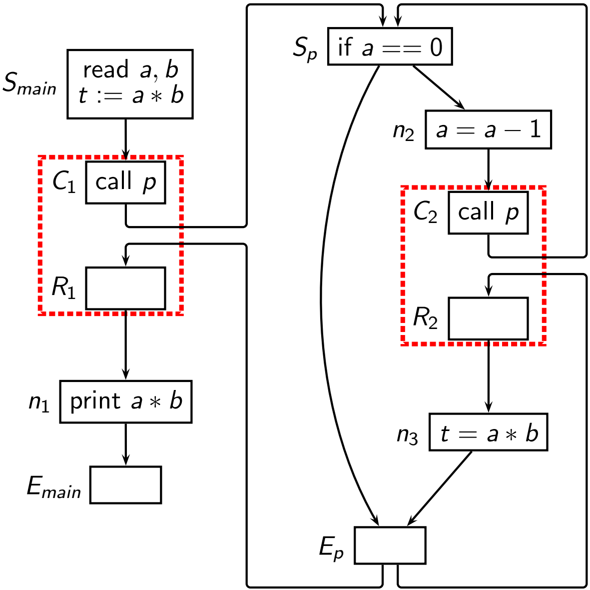
Available Expressions Analysis Using Call Strings (1)



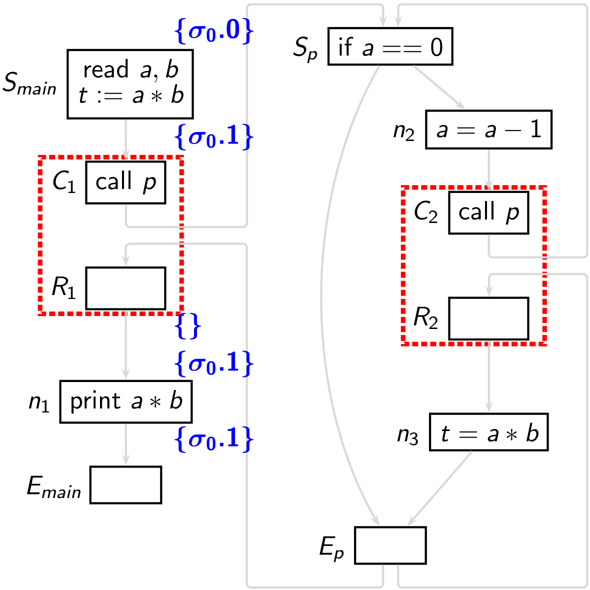
Available Expressions Analysis Using Call Strings (1)



Available Expressions Analysis Using Call Strings (2)



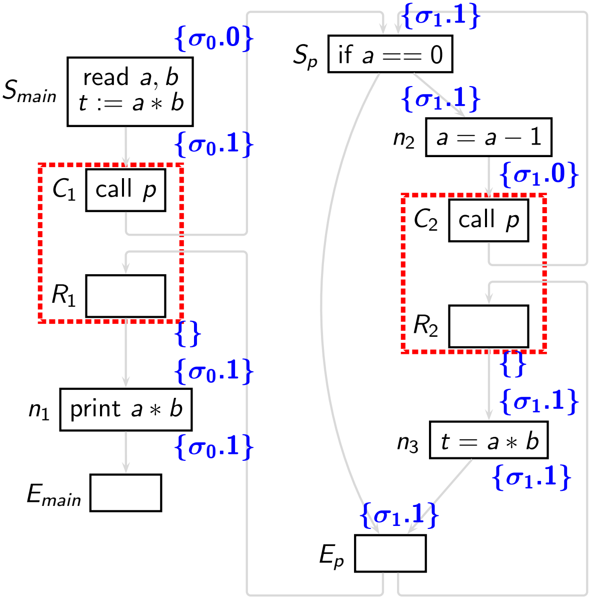
Available Expressions Analysis Using Call Strings (2)

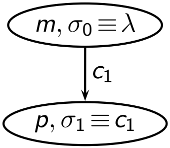


$m, \sigma_0 \equiv \lambda$

Computing the data flow values for S_{main}, C_1, and n_1 for the context λ

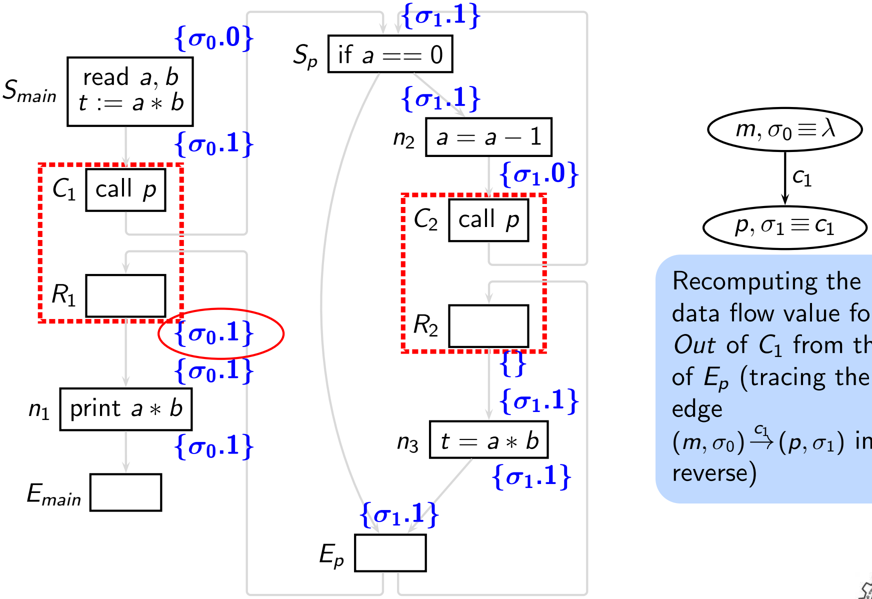
Available Expressions Analysis Using Call Strings (2)





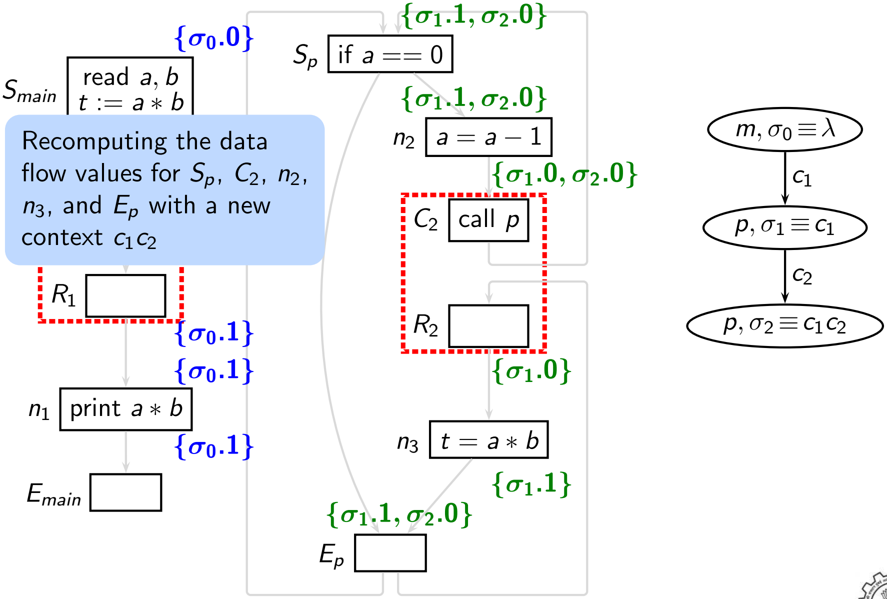
Computing the data flow values for S_p, C_2, n_2, n_3, and E_p for the context c_1

Available Expressions Analysis Using Call Strings (2)



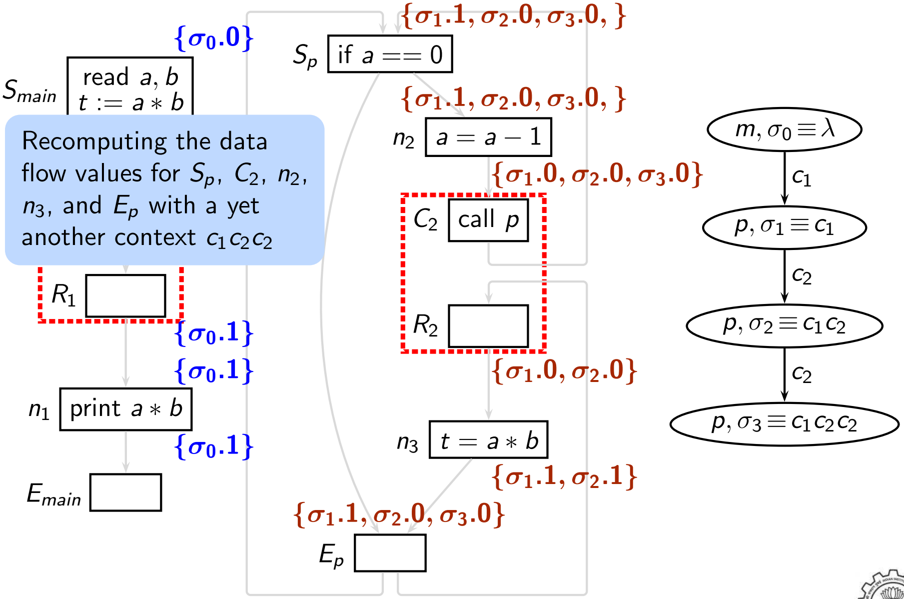
Recomputing the data flow value for *Out* of C_1 from that of E_p (tracing the edge $(m, \sigma_0) \xrightarrow{c_1} (p, \sigma_1)$ in reverse)

Available Expressions Analysis Using Call Strings (2)



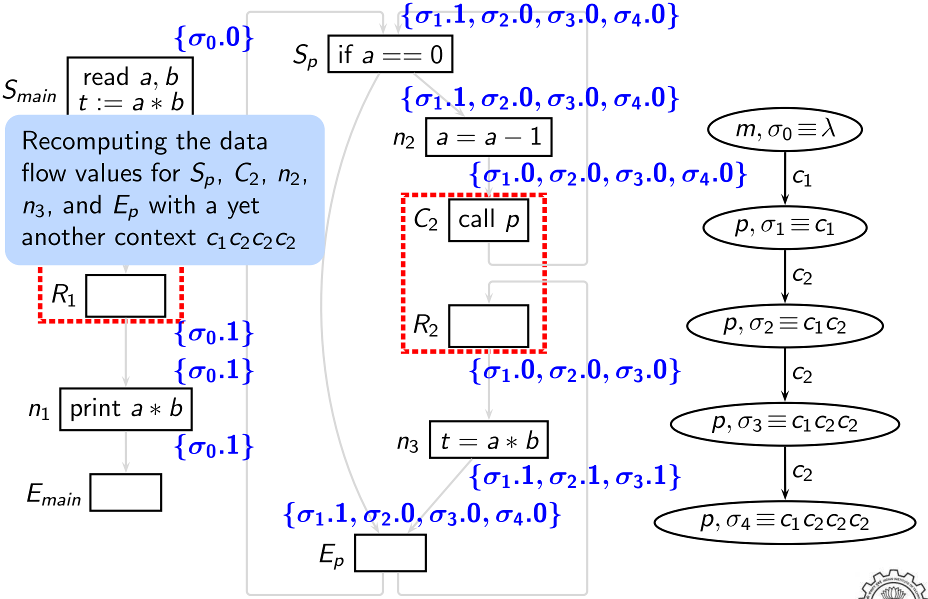
Available Expressions Analysis Using Call Strings (2)

Recomputing the data flow values for S_p, C_2, n_2, n_3, and E_p with a yet another context $c_1 c_2 c_2$

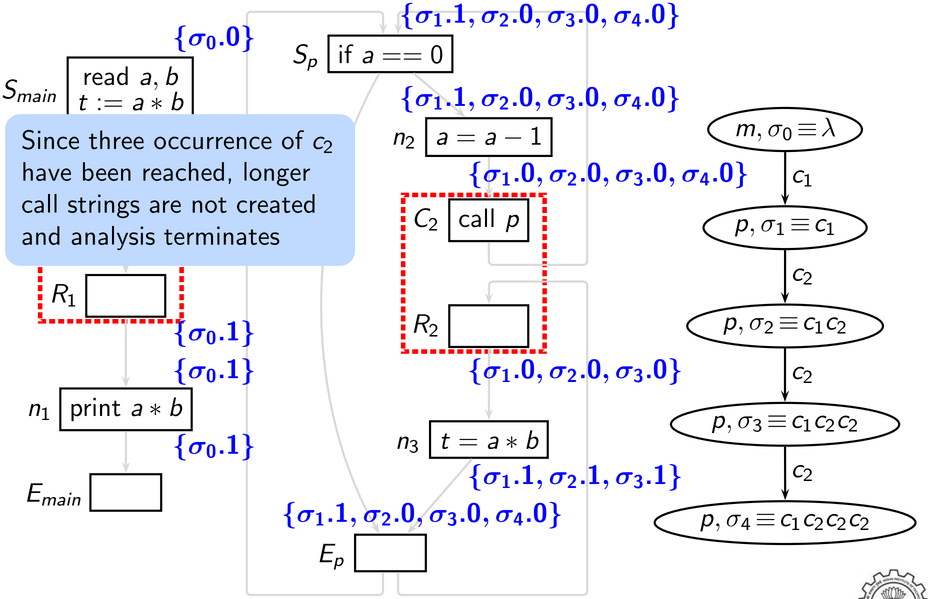


Available Expressions Analysis Using Call Strings (2)

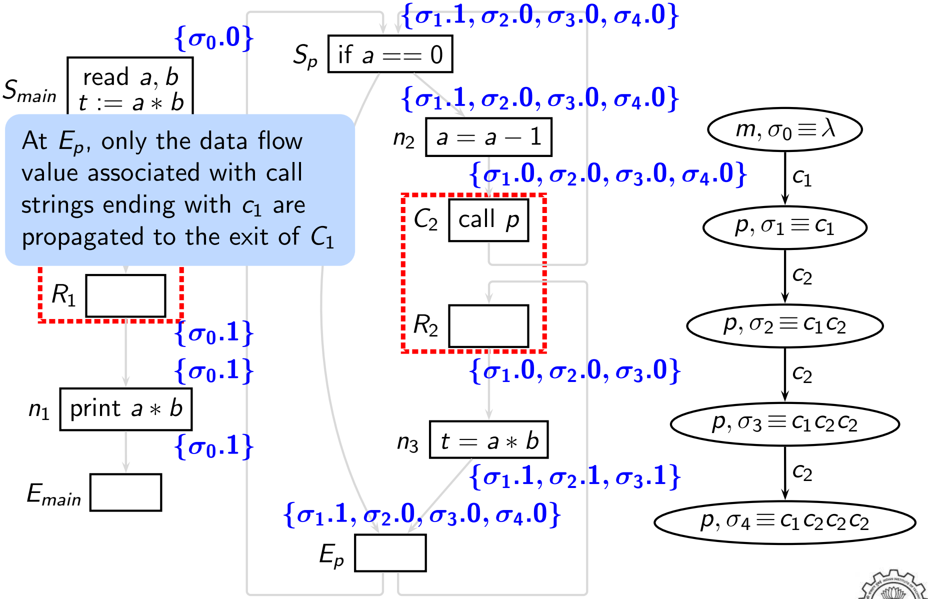
Recomputing the data flow values for S_p, C_2, n_2, n_3, and E_p with a yet another context $c_1 c_2 c_2 c_2$



Available Expressions Analysis Using Call Strings (2)

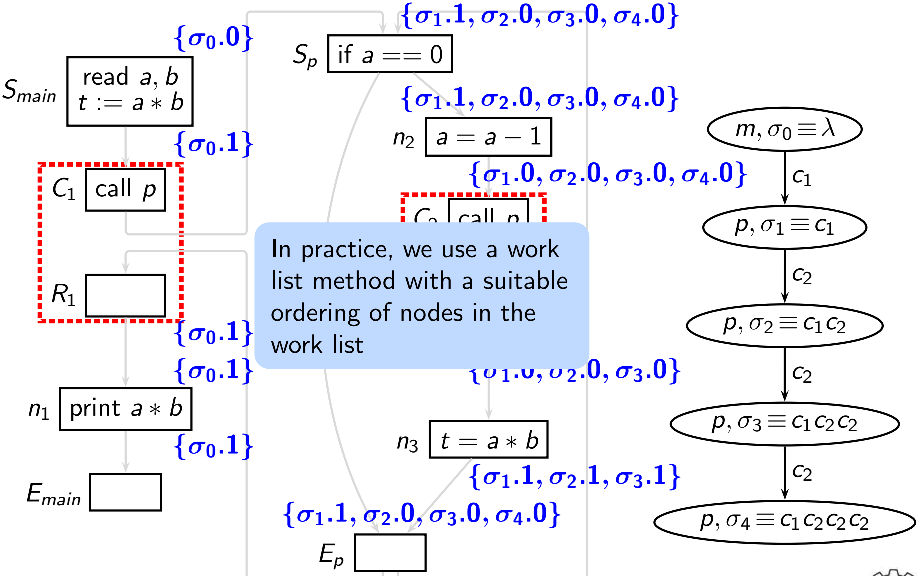


Available Expressions Analysis Using Call Strings (2)



At E_p, only the data flow value associated with call strings ending with c_1 are propagated to the exit of C_1

Available Expressions Analysis Using Call Strings (2)

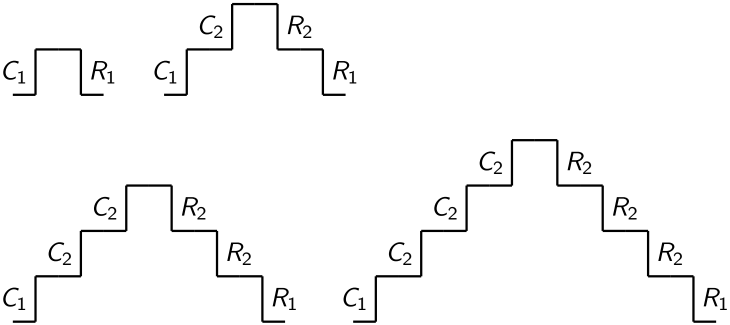


In practice, we use a work list method with a suitable ordering of nodes in the work list

Explaining the Result of Available Expressions Analysis

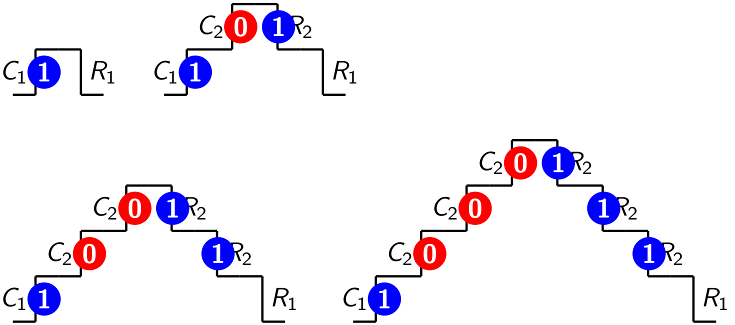
We consider only the paths containing at most three occurrence of any call site

The pattern remains same for longer paths too



Explaining the Result of Available Expressions Analysis

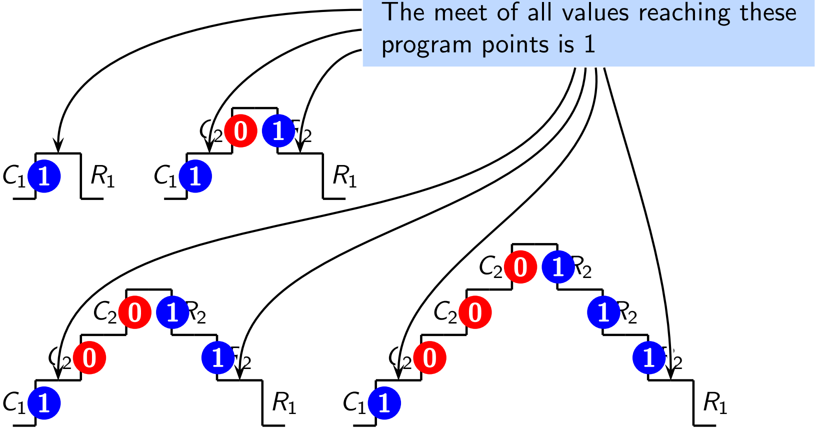
- Generation. Before call site C_1 and after return site R_2
- Killing. Before call site C_2



Explaining the Result of Available Expressions Analysis

Program points for call string c_1

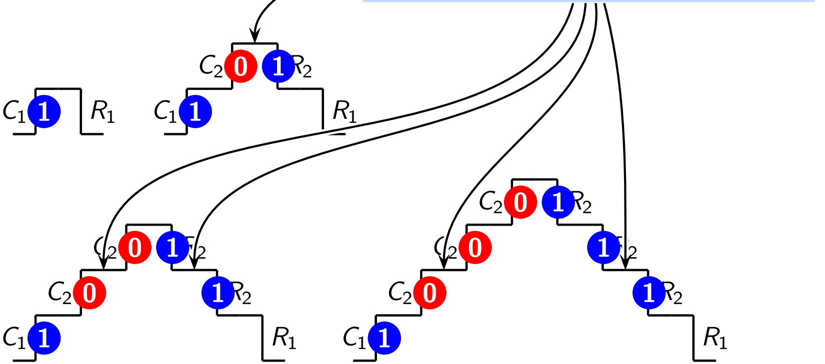
The meet of all values reaching these program points is 1



Explaining the Result of Available Expressions Analysis

Program points for call string $c_1 c_2$

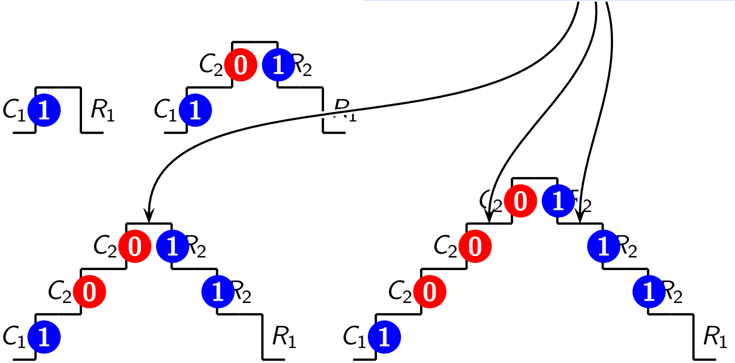
The meet of all values reaching these program points is 0



Explaining the Result of Available Expressions Analysis

Program points for call string $c_1c_2c_2$

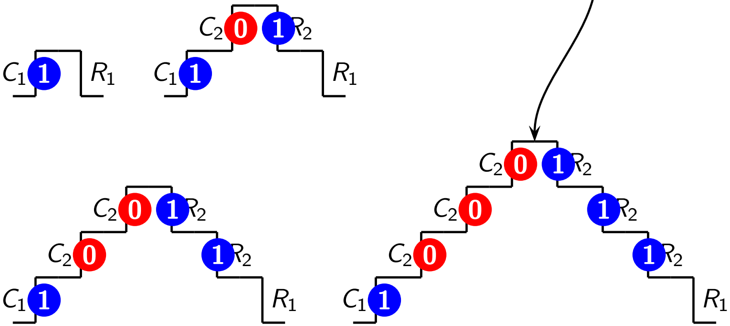
The meet of all values reaching these program points is 0



Explaining the Result of Available Expressions Analysis

Program point for call string $c_1c_2c_2c_2$

The meet of all values reaching these program points is 0



Tutorial Problem #1

Perform available expressions analysis for the following program

```
main()
{
    a = b*c;

    p();        /* C1 */

    d = b*c;    /* avail b*c? */

    q();        /* C2 */
}
```

```
p()
{
}

q()
{
    b = 5;

    p();    /* C3 */
}
```

Tutorial Problem #2

Is a*b available on line 18 in the following program? On line 15? Construct its supergraph and argue in terms of interprocedurally valid paths

```
1.  main()
2.  {
3.        c = a*b;
4.        p();
5.        a = a*b;
6.  }
```

```
 7.  p()
 8.  {    if (...)
 9.       {    a = a*b;
10.            p();
11.       }
12.       else if (...)
13.       {    c = a * b;
14.            p();
15.            c = a;
16.       }
17.       else
18.            ; /* ignore */
19.  }
```

Part 7

IPDFA Using Value Contexts

Value Contexts: Key Ideas

Consider call chains σ_1 and σ_2 reaching S_p

- Data flow value invariant:

  If the data flow reaching S_p along σ_1 and σ_2 are identical, then

Value Contexts: Key Ideas

Consider call chains σ_1 and σ_2 reaching S_p

- Data flow value invariant:

  If the data flow reaching S_p along σ_1 and σ_2 are identical, then

  - the data flow values reaching E_p for the two contexts will also be identical

Value Contexts: Key Ideas

Consider call chains σ_1 and σ_2 reaching S_p

- Data flow value invariant:

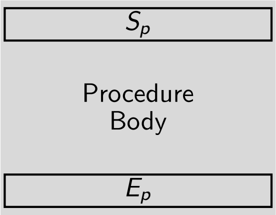
  If the data flow reaching S_p along σ_1 and σ_2 are identical, then

  - the data flow values reaching E_p for the two contexts will also be identical

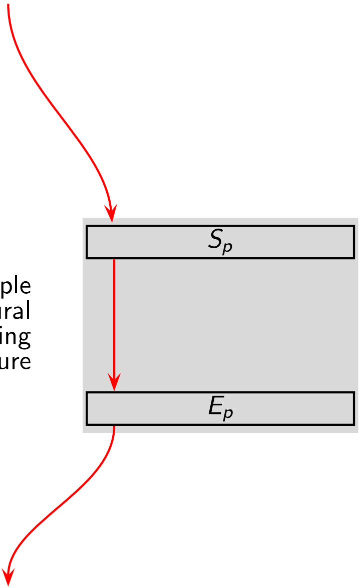
- We can reduce the amount of effort by using

  - Data flow values at S_p as value contexts
  - Maintaining distinct data flow values in p for each value context

Understanding Value Contexts



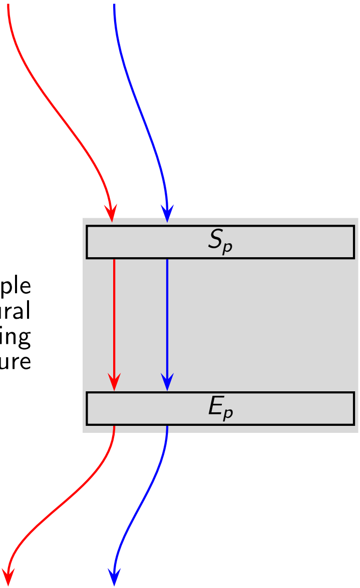
Understanding Value Contexts



Multiple interprocedural paths reaching the procedure

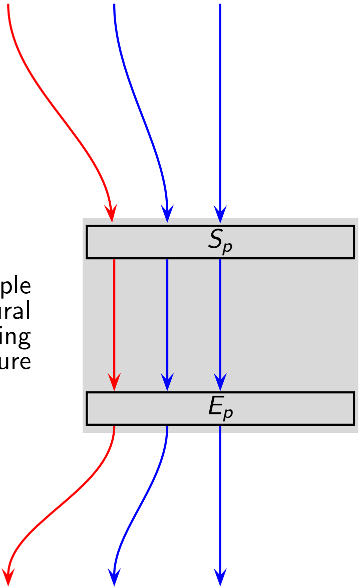
Understanding Value Contexts

Multiple interprocedural paths reaching the procedure

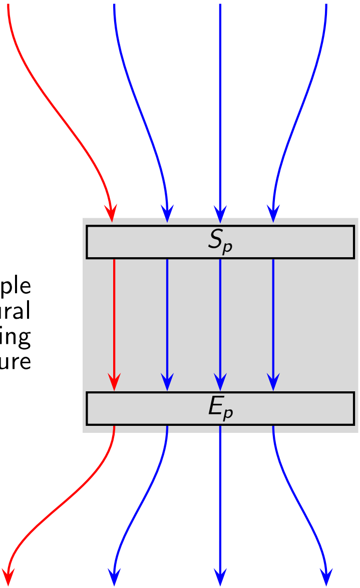


Understanding Value Contexts

Multiple interprocedural paths reaching the procedure



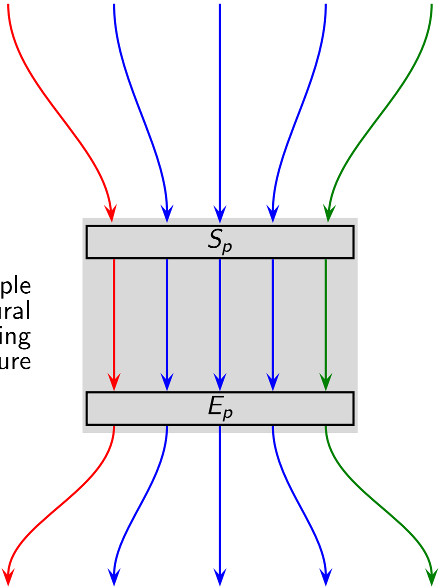
Understanding Value Contexts



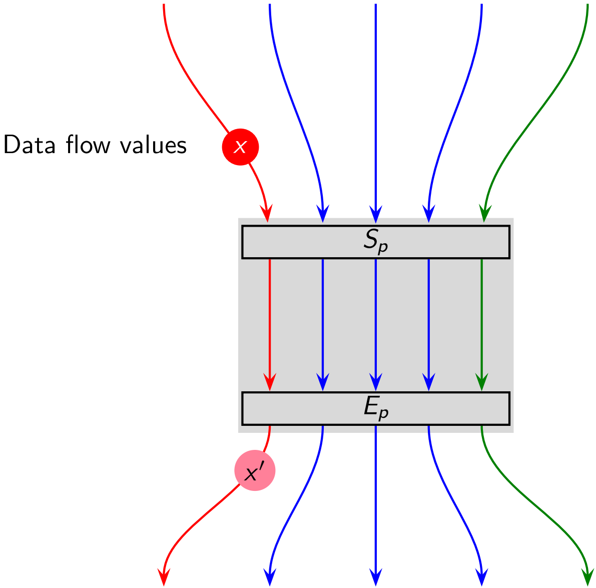
Multiple interprocedural paths reaching the procedure

Understanding Value Contexts

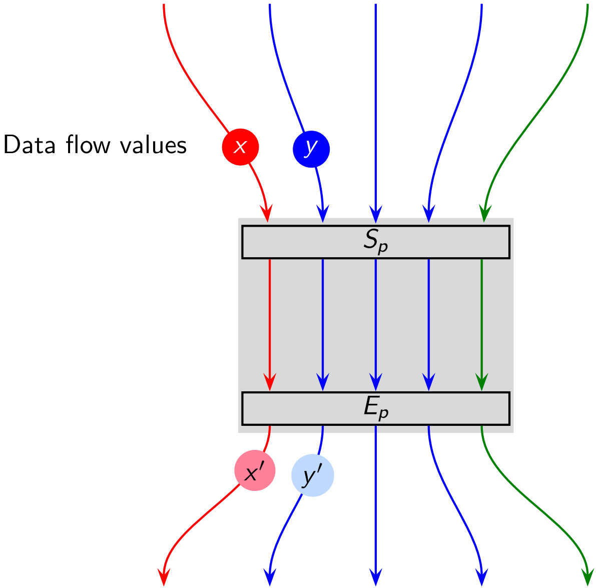
Multiple interprocedural paths reaching the procedure



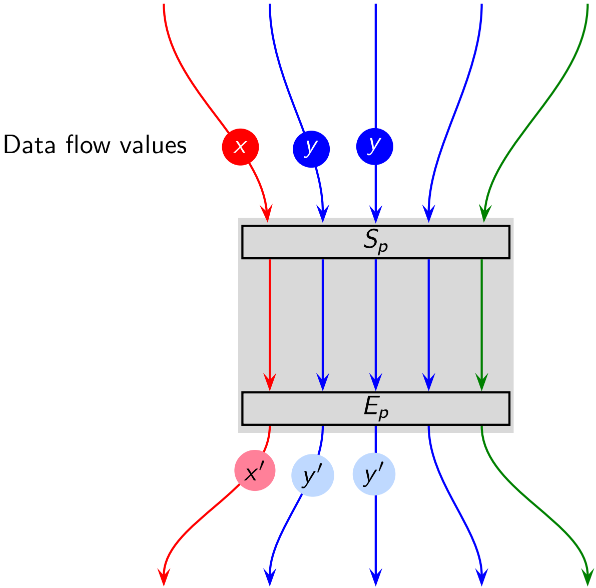
Understanding Value Contexts



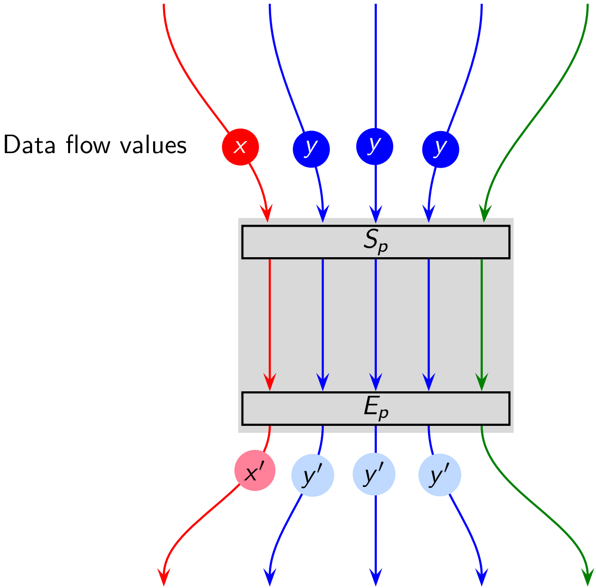
Understanding Value Contexts



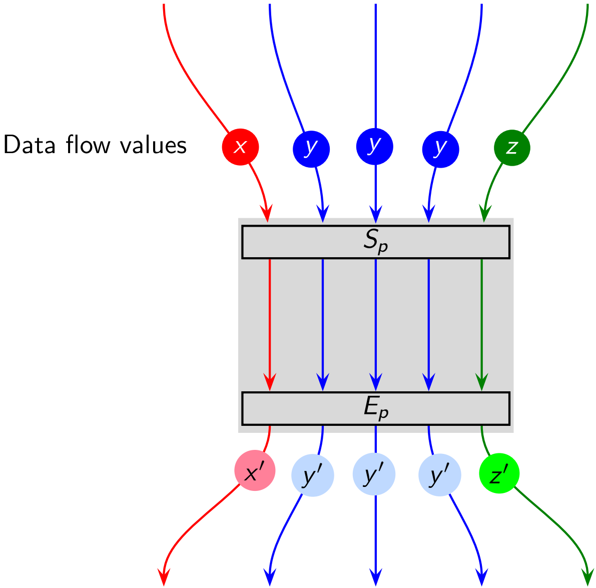
Understanding Value Contexts



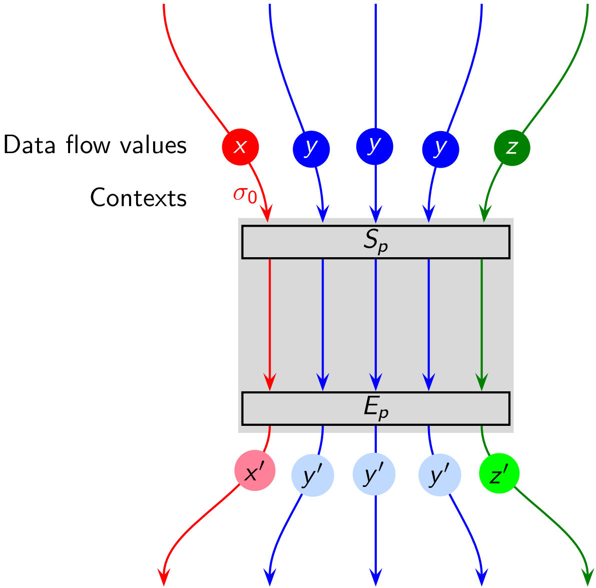
Understanding Value Contexts



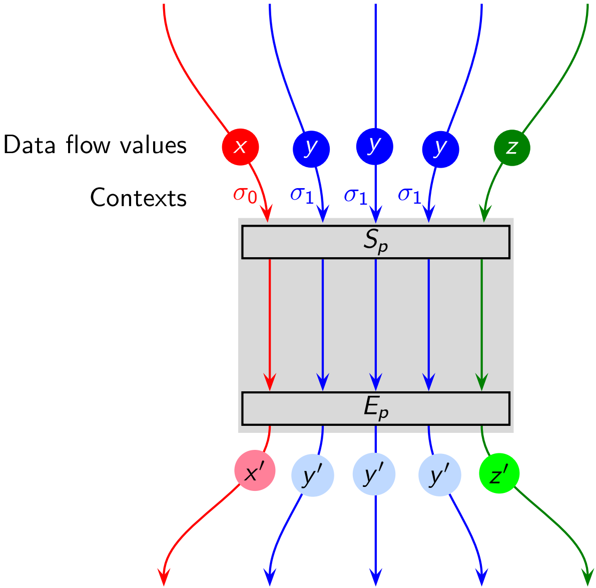
Understanding Value Contexts



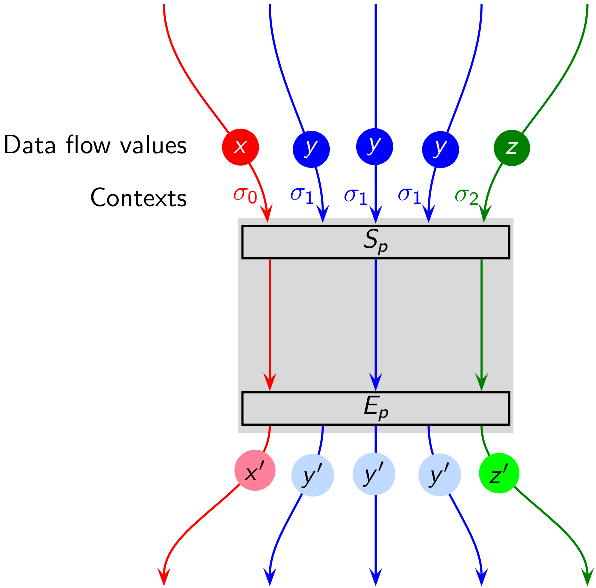
Understanding Value Contexts



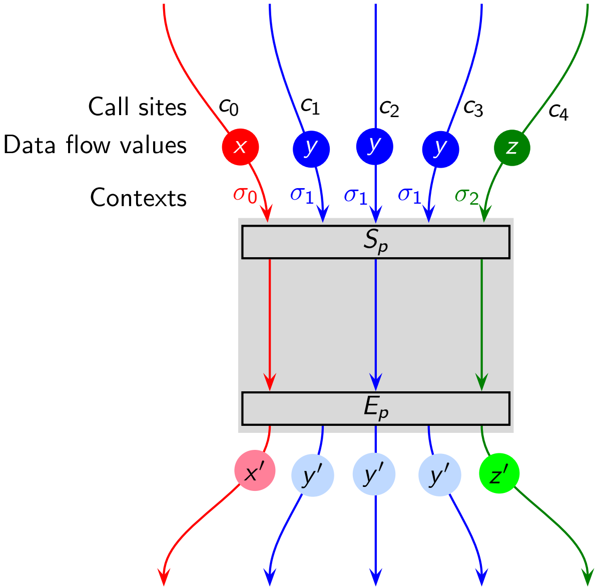
Understanding Value Contexts



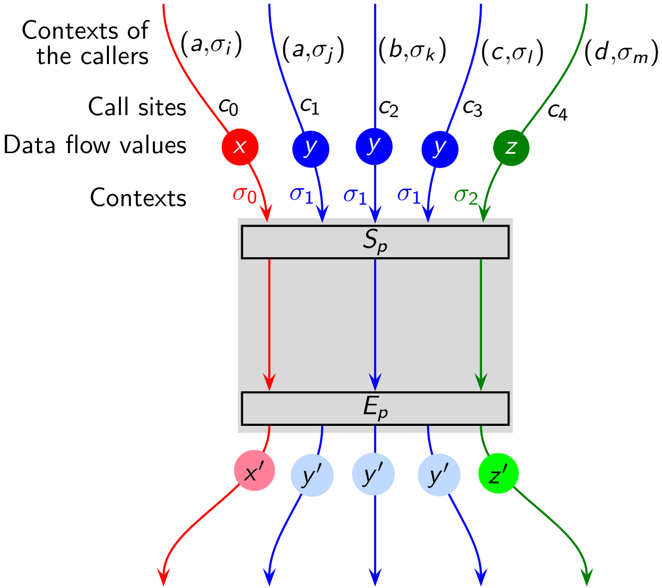
Understanding Value Contexts



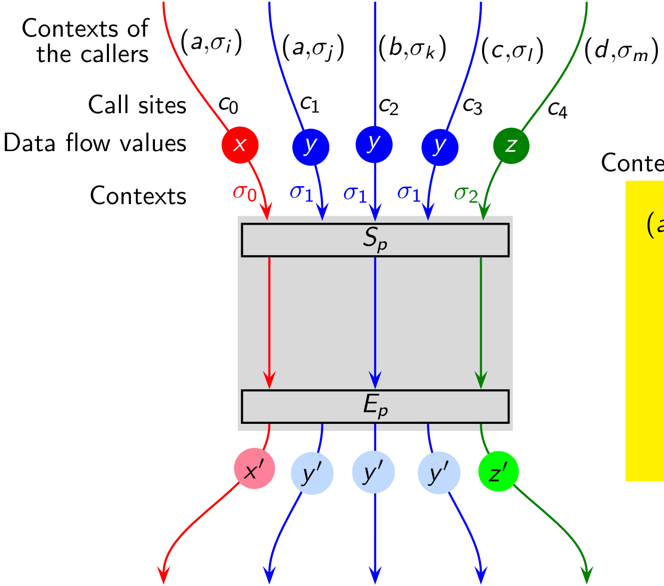
Understanding Value Contexts



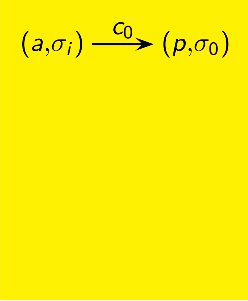
Understanding Value Contexts



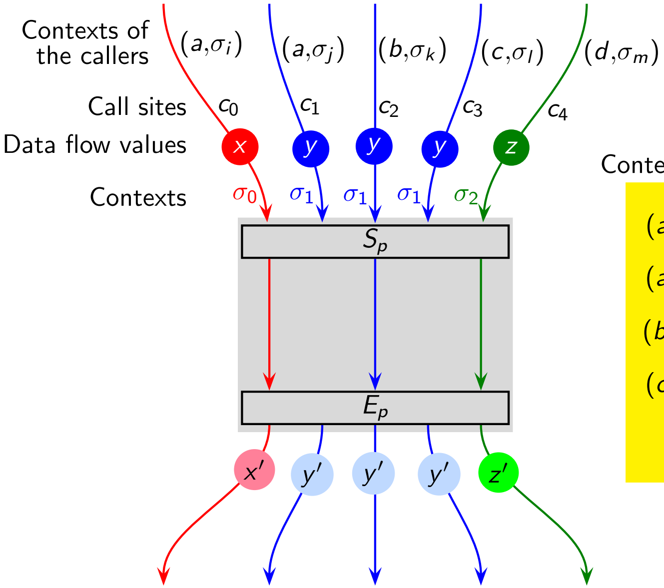
Understanding Value Contexts



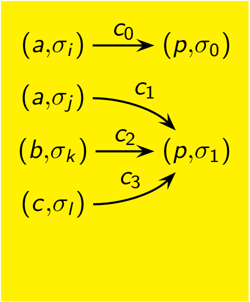
Context-sensitive call graph



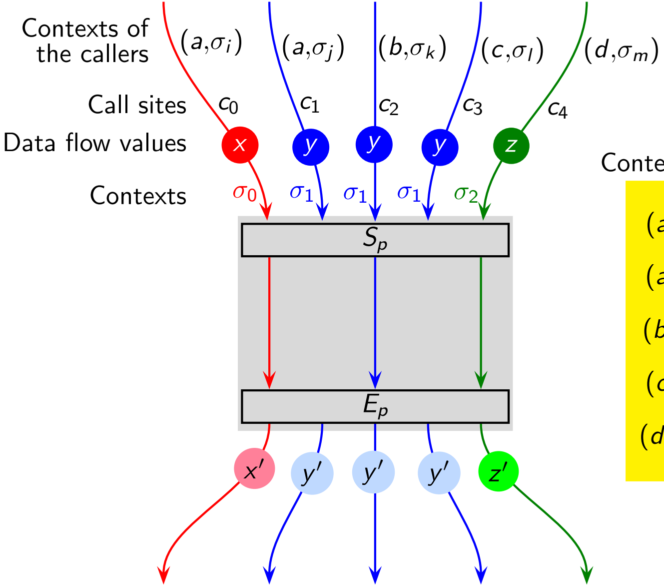
Understanding Value Contexts



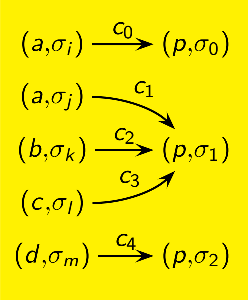
Context-sensitive call graph



Understanding Value Contexts



Context-sensitive call graph



Interprocedural Data Flow Analysis Using Value Contexts

- A value context is defined by a particular input data flow value reaching a procedure
- It is used to enumerate the summary flow functions as an extensional representation in terms of (input $\mapsto$ output) pairs
- In order to compute these pairs, data flow analysis within a procedure is performed separately for each context (i.e. input data flow value)
- When a new call to a procedure is encounterd, the pairs are consulted do decide if the procedure needs to be analysed again
    - If it was already analysed once for the input value, output can be directly processed
    - Otherwise, a new context is created and the procedure is analysed for this new context

Thus, each procedure is analysed only once for a specific input data flow value

Instantiating the Unified Model to Value Contexts

We need to define three things:

- The initial context σ_0
- We need to define the $\textsc{context}(p, C_i, \sigma)$ function
- We need to ensure that the set of context Σ is finite

Instantiating the Unified Model to Value Contexts

We need to define three things:

- The initial context σ_0

  σ_0 is the boundary information *BI*of the *main* procedure

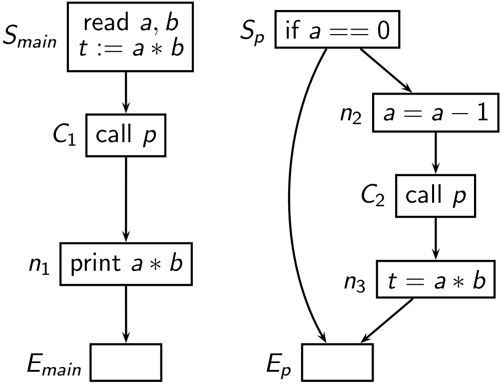
- We need to define the $\text{CONTEXT}(p, C_i, \sigma)$ function

  $\text{CONTEXT}(p, C_i, \sigma) = In_{C_i}^{p}(\sigma)$

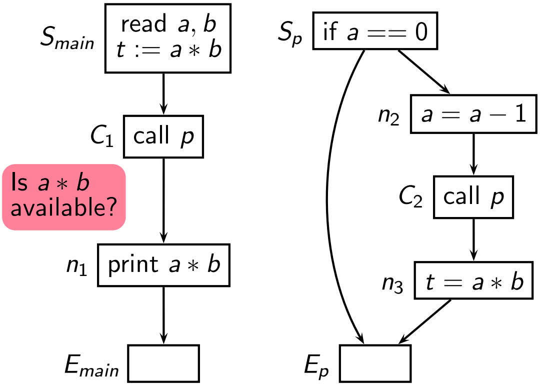
- We need to ensure that the set of context Σ is finite

  Follows from the finiteness of data flow values

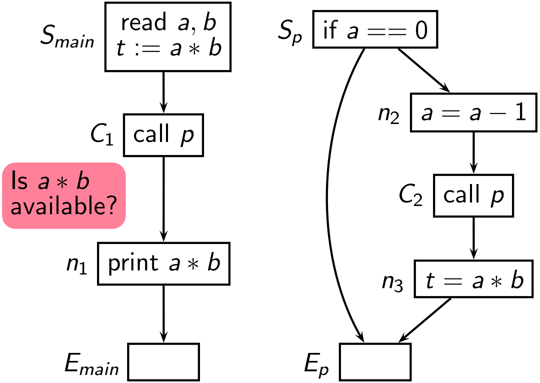
Available Expressions Analysis Using Value Contexts



Available Expressions Analysis Using Value Contexts

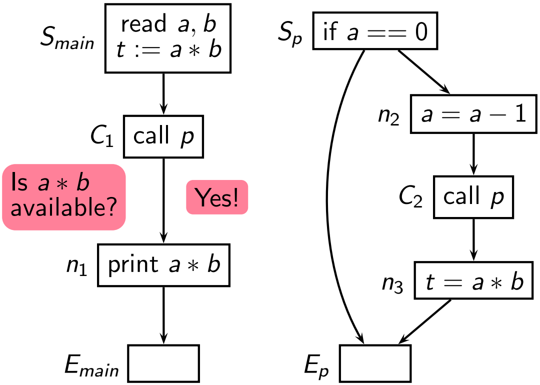


Available Expressions Analysis Using Value Contexts



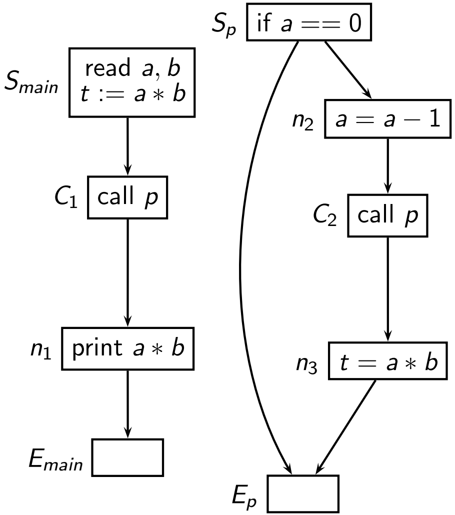

```
int a, b, t;
void p()
{  if (a == 0)
   {  a = a-1;
      p();
      t = a*b;
   }
}
```

Available Expressions Analysis Using Value Contexts

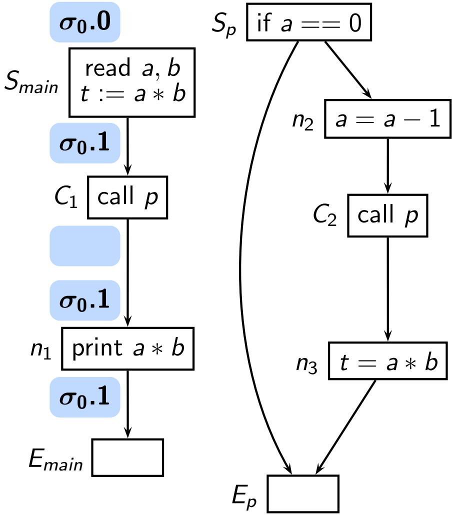



```
int a, b, t;
void p()
{  if (a == 0)
   {  a = a-1;
      p();
      t = a*b;
   }
}
```

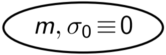
Available Expressions Analysis Using Value Contexts



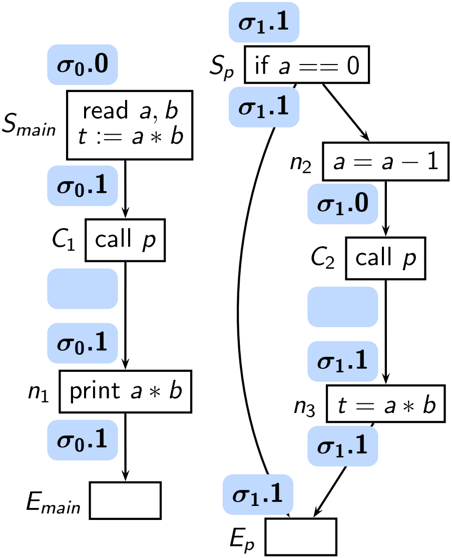
Available Expressions Analysis Using Value Contexts



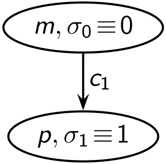
Create context-sensitive call graph with a node for *main* with context σ_0 as data flow value 0 (for *BI*) and compute the data flow values for *main*



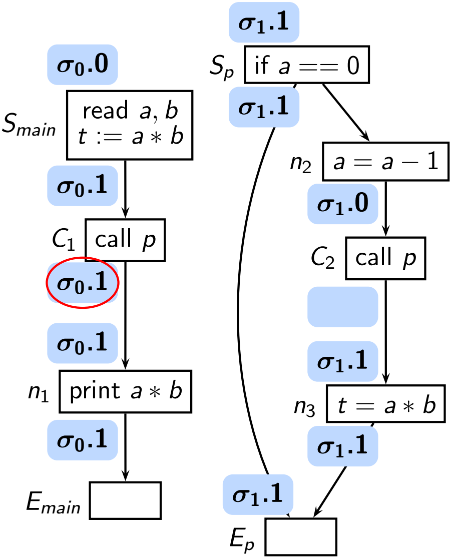
Available Expressions Analysis Using Value Contexts



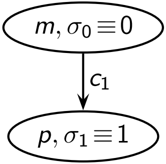
Create a new node for p in the context-sensitive call graph with a new context σ_1 as data flow value 1 reaching S_p from call node C_1 in *main* and compute the data flow values for p



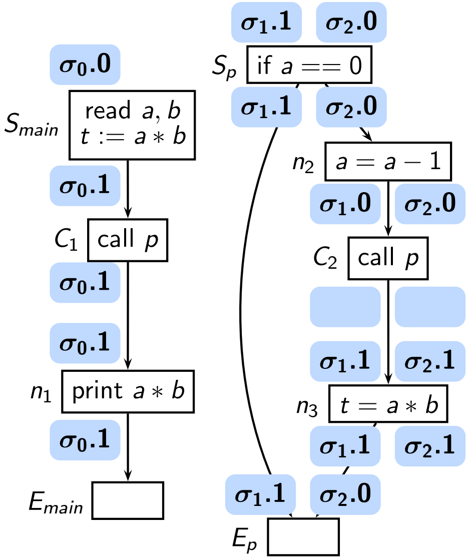
Available Expressions Analysis Using Value Contexts



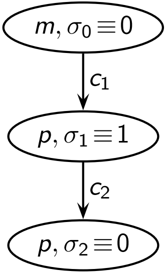
Recompute the value at the exit of C_1 for context σ_0 from the value of context σ_1 at E_p (tracing the edge $(m, \sigma_0) \xrightarrow{c_1} (p, \sigma_1)$ in reverse in the context-sensntive call graph)



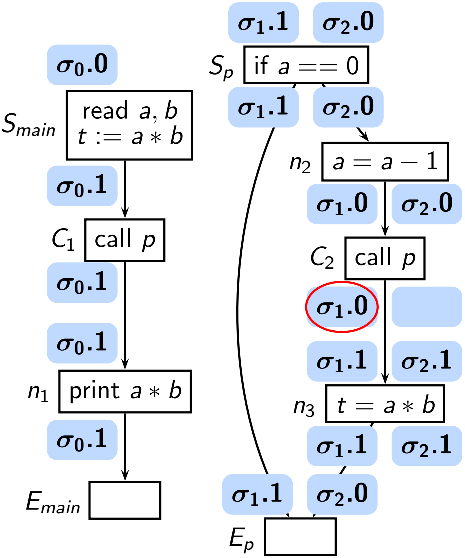
Available Expressions Analysis Using Value Contexts



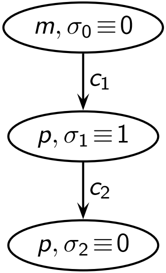
Create a new node for p in the context-sensitive call graph with a new context σ_2 as data flow value 0 reaching S_p from call node C_2 within p and compute the data flow values for p



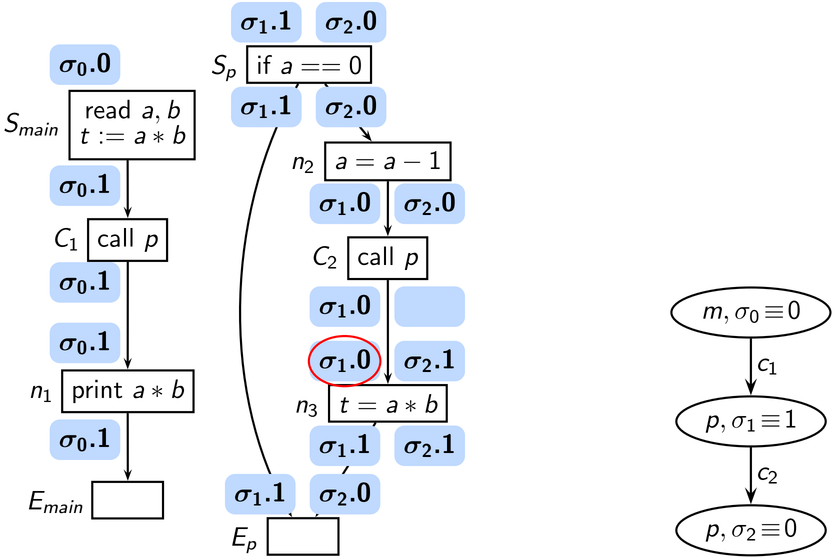
Available Expressions Analysis Using Value Contexts



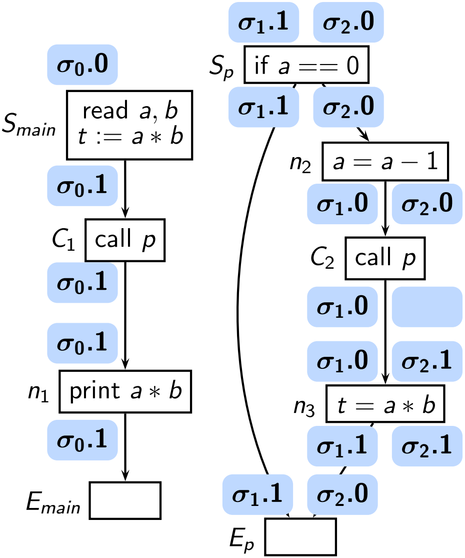
Recompute the value at the exit of C_2 for context σ_1 from the value of context σ_2 at E_p (tracing the edge $(p,\sigma_1)\xrightarrow{c_2}(p,\sigma_2)$ in reverse in the context-sensntive call graph)



Available Expressions Analysis Using Value Contexts

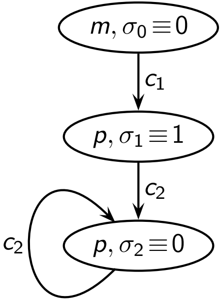


Available Expressions Analysis Using Value Contexts

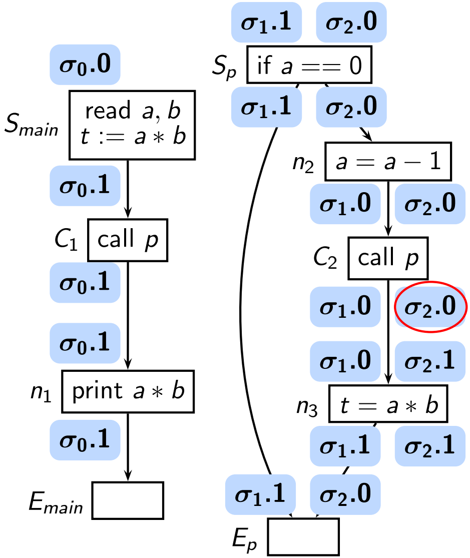


Since data flow value for σ_2 is 0 the entry of C_2, we need to create a context for p with value 0

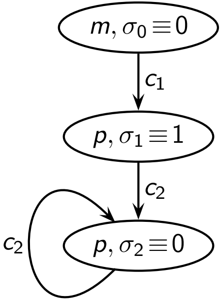
Such a context already exists in the form of σ_2 so we merely add an edge from (p, σ_2) to itself in the context-sensitive call graph



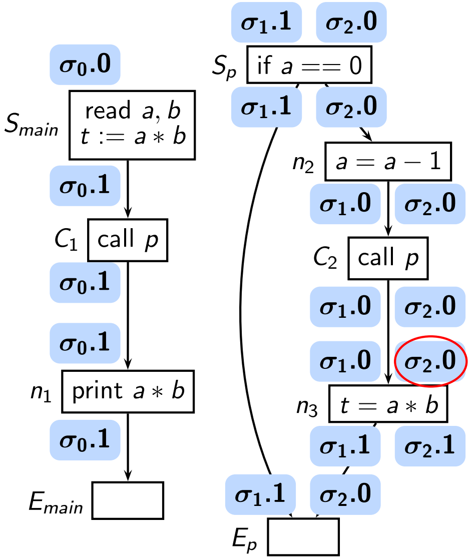
Available Expressions Analysis Using Value Contexts

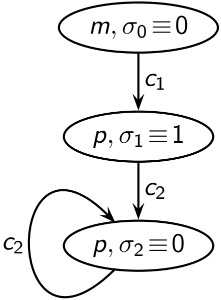


Recompute the value at the exit of C_2 for context σ_2 from the value of context σ_2 at E_p (tracing the edge $(p, \sigma_2) \xrightarrow{c_2} (p, \sigma_2)$ in reverse in the context-sensntive call graph)

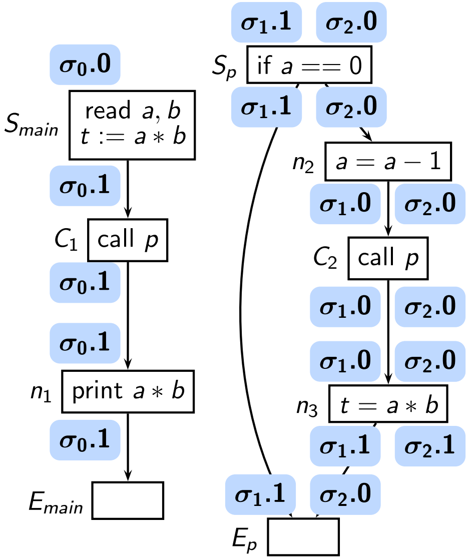


Available Expressions Analysis Using Value Contexts

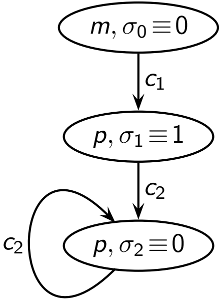




Available Expressions Analysis Using Value Contexts

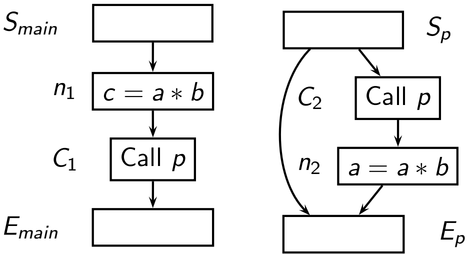


No new data flow reaches any call point and the data flow analysis terminates



Tutorial Problem #1 for Value Contexts

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



Tutorial Problem #2 for Value Contexts

Perform interprocedural live variables analysis using value contexts

<pre>main()
{
   p();
}</pre>
	<pre>p()
	{ while (...)
	   { printf ("%d\n",a);
	      p();
	   }
	}</pre>

Observe the change in edges in the context-sensitive call graph

Tutorial Problem #3 for Value Contexts

Perform interprocedural available expressions analysis using value contexts

```
main()
{
    c = a*b;
    p();
}
```

```
p()
{
    while (a > b)
    {
        p();
        a = a*b;
    }
}
```

Observe the change in edges in the context-sensitive call graph

Tutorial Problem #4 for Value Contexts

Perform interprocedural available expressions analysis using value contexts

```
main()
{
        c = a*b;
        p();
        a = a*b;
}
```

```
p()
{   if (...)
    {   a = a*b;
        p();
    }
    else if (...)
    {   c = a * b;
        p();
        c = a;
    }
    else
        ; /* ignore */
}
```

Tutorial Problem #5 for Value Contexts

Perform interprocedural live variables analysis using value contexts

```
main()
{
  a = 5; b = 3;
  c = 7; d = 2;
  p();
  a = a + 2;
  e = c+d;
  d = a*b;
  q();
  print a+c+e;
}
```

```
p()
{
  b = 2;
  if (b<d)
     c = a+b;
  else
     q();
  print c+d;
}
```

```
q()
{
  a = 1;
  p();
  a = a*b;
}
```

Context sensitivity: e is live on entry to p but not before its call in main

Result of Tutorial #5

```
main()
{
  a = 5; b = 3;
  c = 7; d = 2;
  /*{a,d}*/
  p();
  /*{a,b,c,d}*/
  a = a + 2;
  e = c+d;
  /*{a,b,e}*/
  d = a*b;
  /*{d,e}*/
  q();
  /*{a,c,e}*/
  print a+c+e;
}
```

```
p()
{  /*{a,d,e}*/
  b = 2;
  if (b<d)
    /*{a,b,d,e}*/
     c = a+b;
  else
    /*{d,e}*/
     q();
  /*{a,b,c,d,e}*/
  print c+d;
}
```

```
q()
{
  /*{d,e}*/
  a = 1;
  /*{a,d,e}*/
  p();
  /*{a,b,c,d,e}*/
  a = a*b;
}
```