Issues in Interprocedural Pointer Analysis

Uday Khedker

(www.cse.iitb.ac.in/~uday)

Department of Computer Science and Engineering,

Indian Institute of Technology, Bombay

Dec 2017

Outline

- Issues in Interprocedural Analysis
- Top-Down Approaches for Flow and Context-Sensitive Analysis
- Bottom-Up Approaches for Flow and Context-Sensitive Analysis
- Conclusions

Part 1

Issues in Interprocedural Analysis

Interprocedural Analysis: Overview

- Extends the scope of data flow analysis across procedure boundaries

  Incorporates the effects of

  - procedure calls in the caller procedures, and
  - calling contexts in the callee procedures

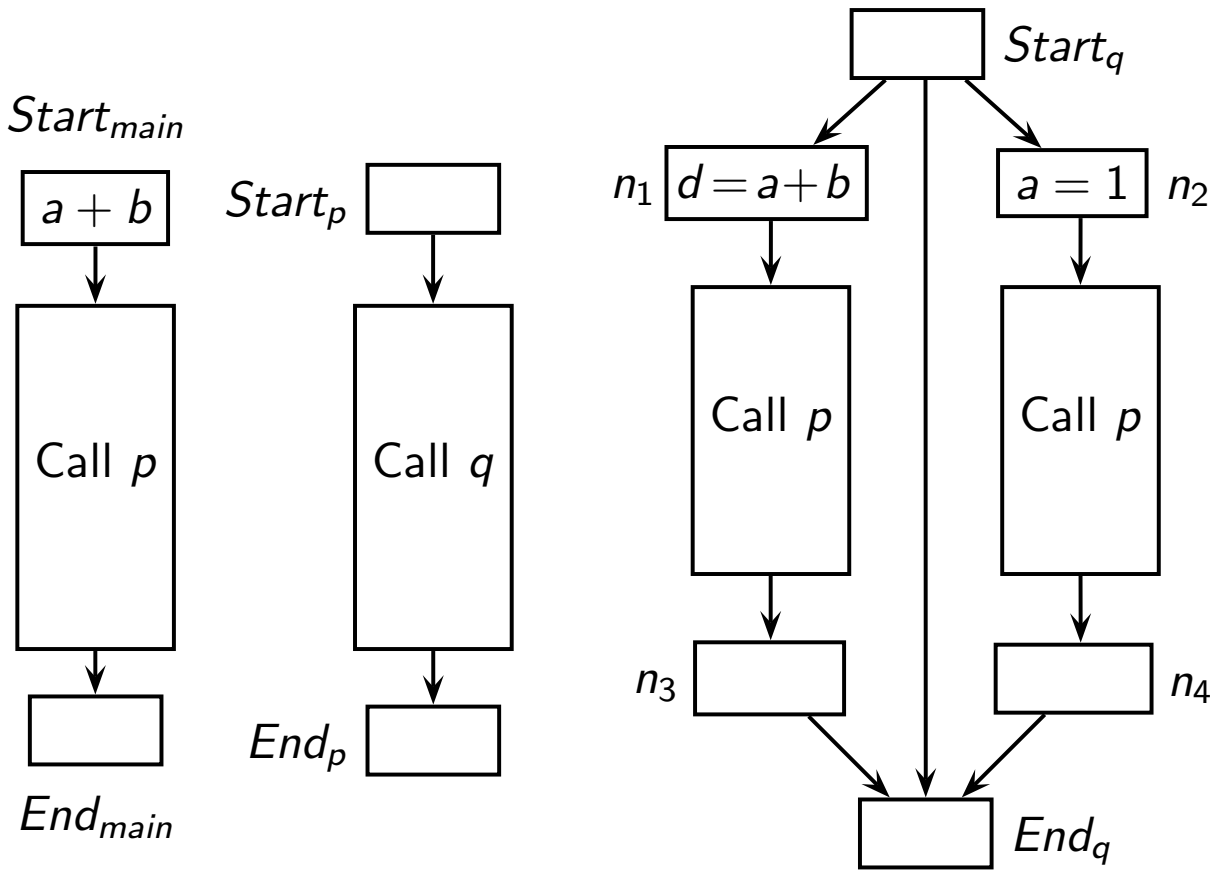
- Approaches :

  - Generic : Call strings approach, functional approach
  - Problem specific : Alias analysis, Points-to analysis, Partial redundancy elimination, Constant propagation

Why Interprocedural Analysis?

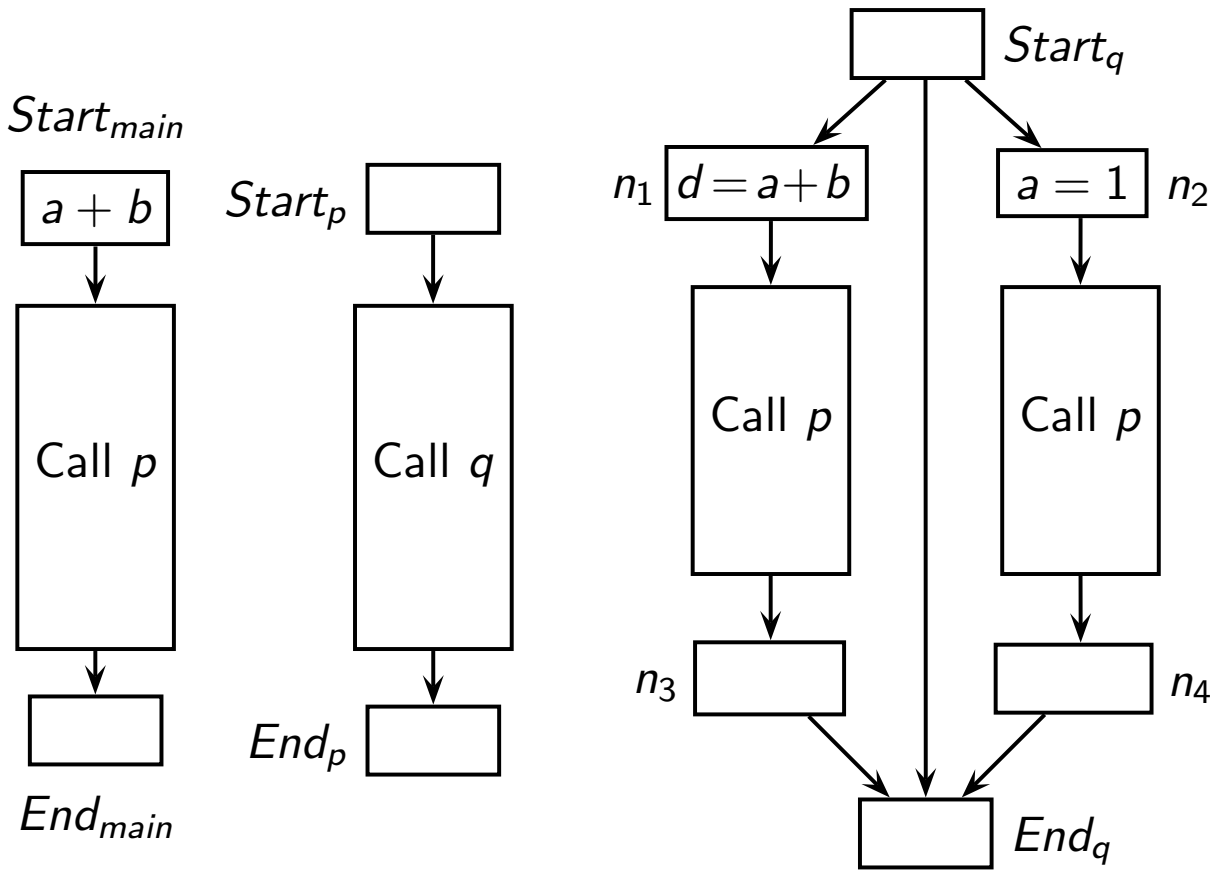
- Answering questions about formal parameters and global variables:
  - Which variables are constant?
  - Which variables aliased with each other?
  - Which locations can a pointer variable point to?
- Answering questions about side effects of a procedure call:
  - Which variables are defined or used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

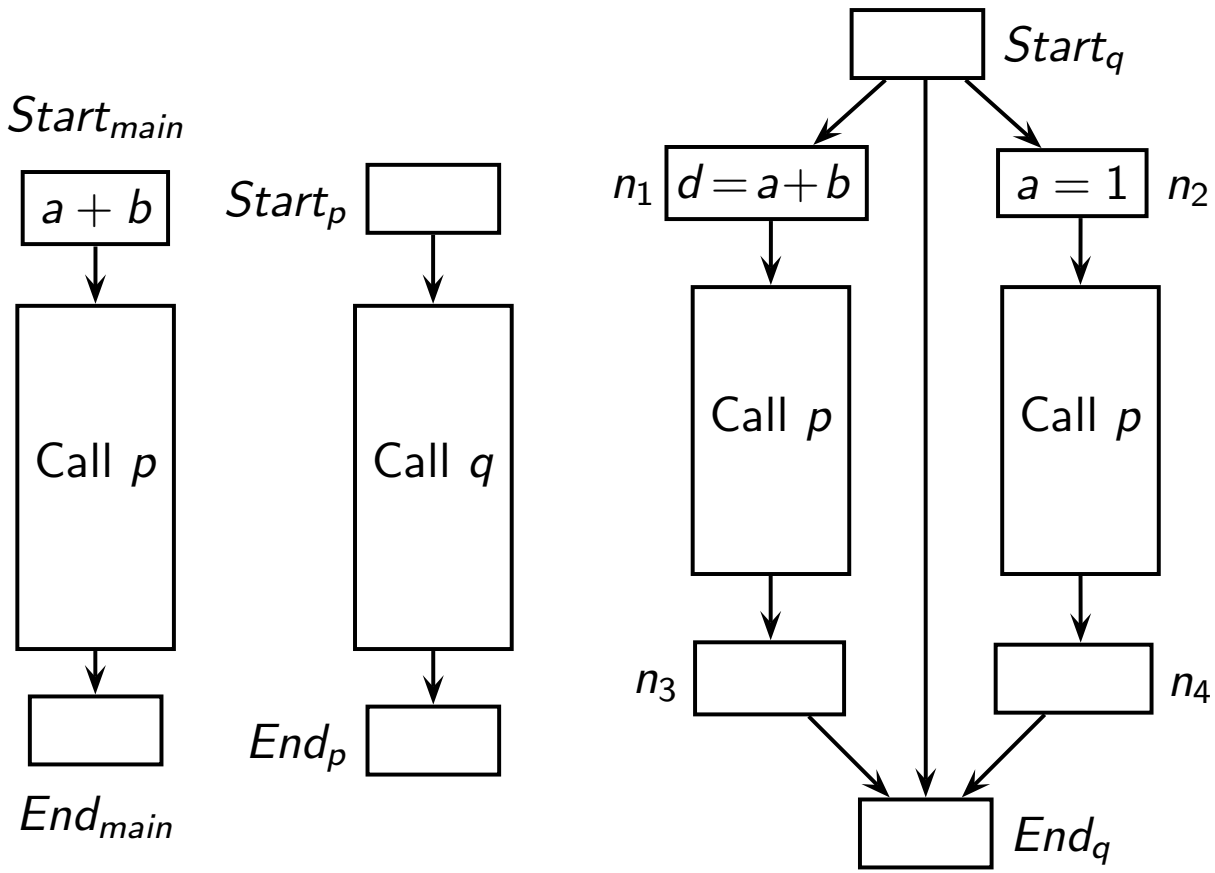
Program Representation for Interprocedural Data Flow Analysis: Call Multi-Graph



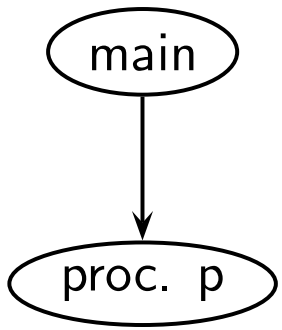
Supergraphs of procedures

Call multi-graph

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

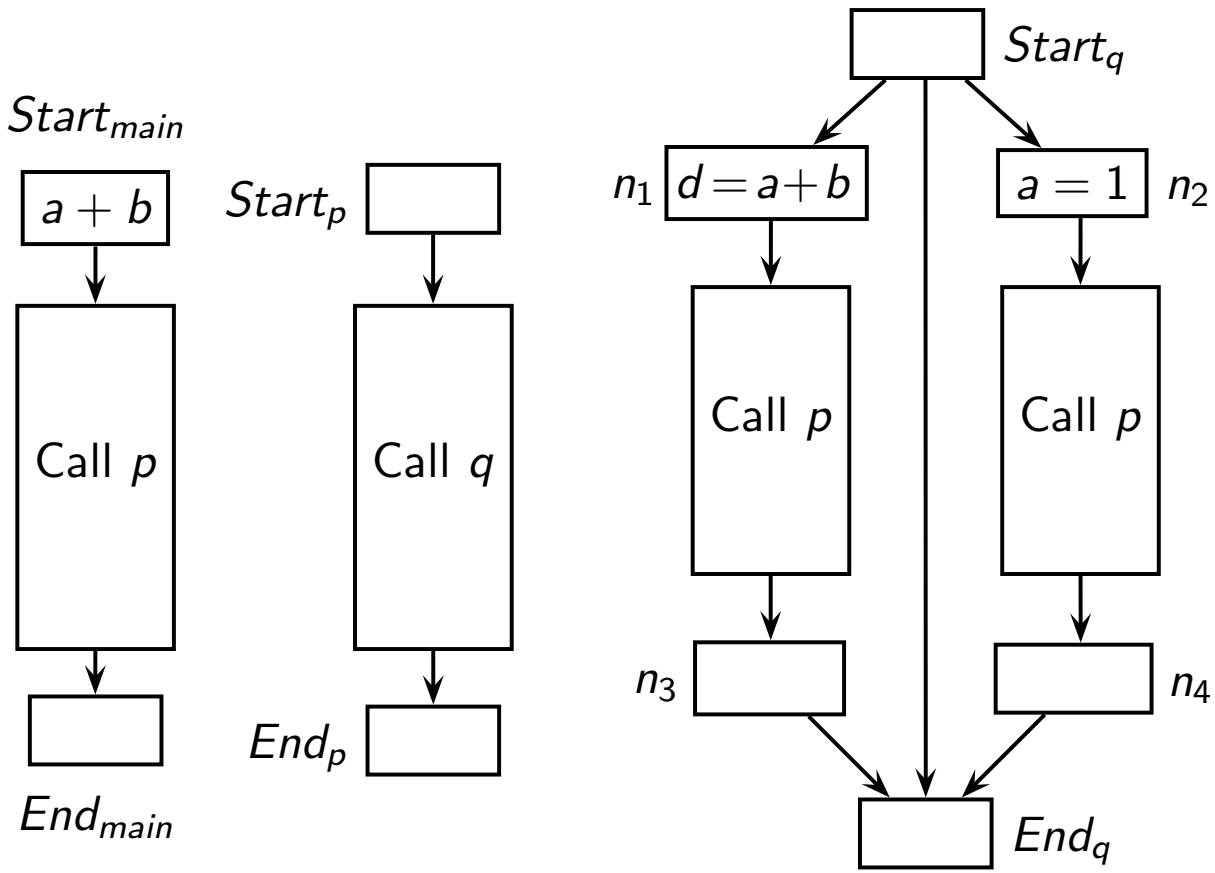


Supergraphs of procedures

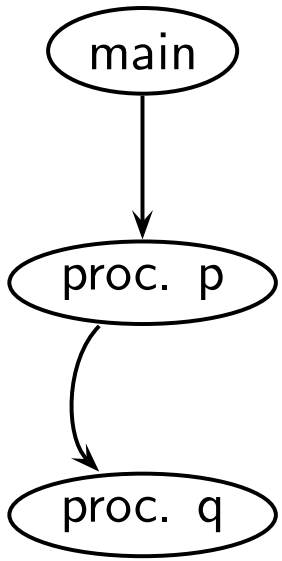


Call multi-graph

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

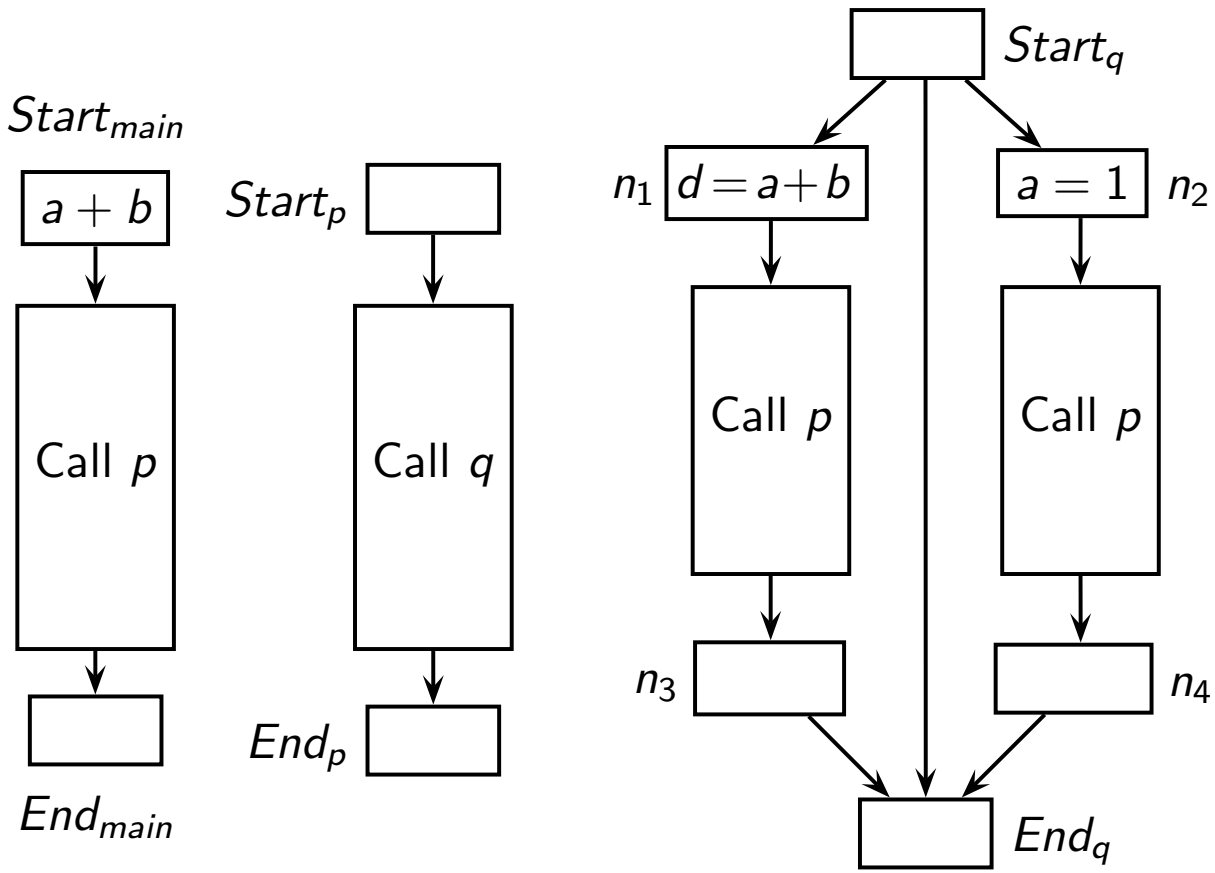


Supergraphs of procedures

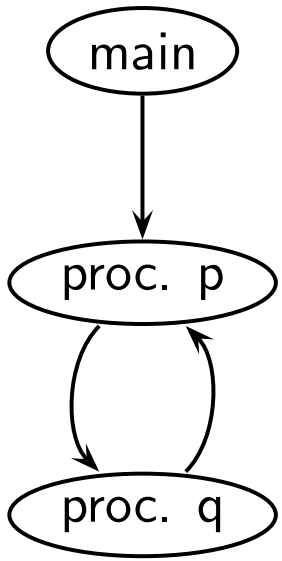


Call multi-graph

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

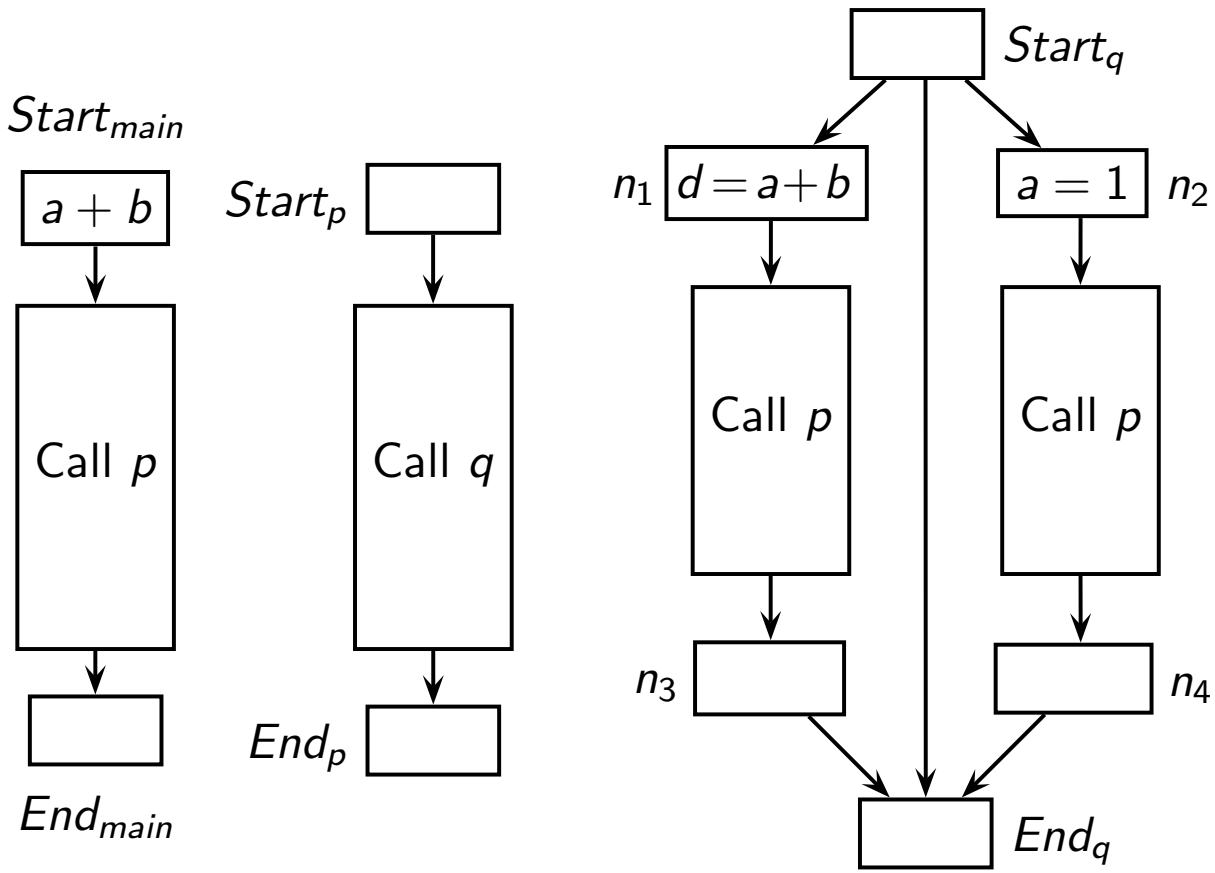


Supergraphs of procedures

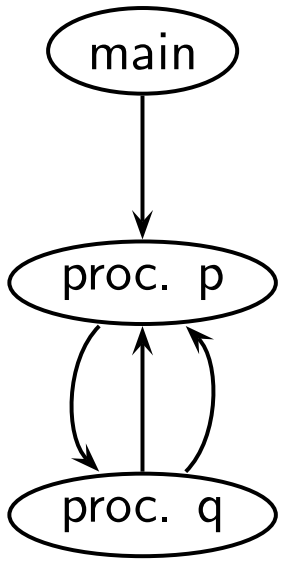


Call multi-graph

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

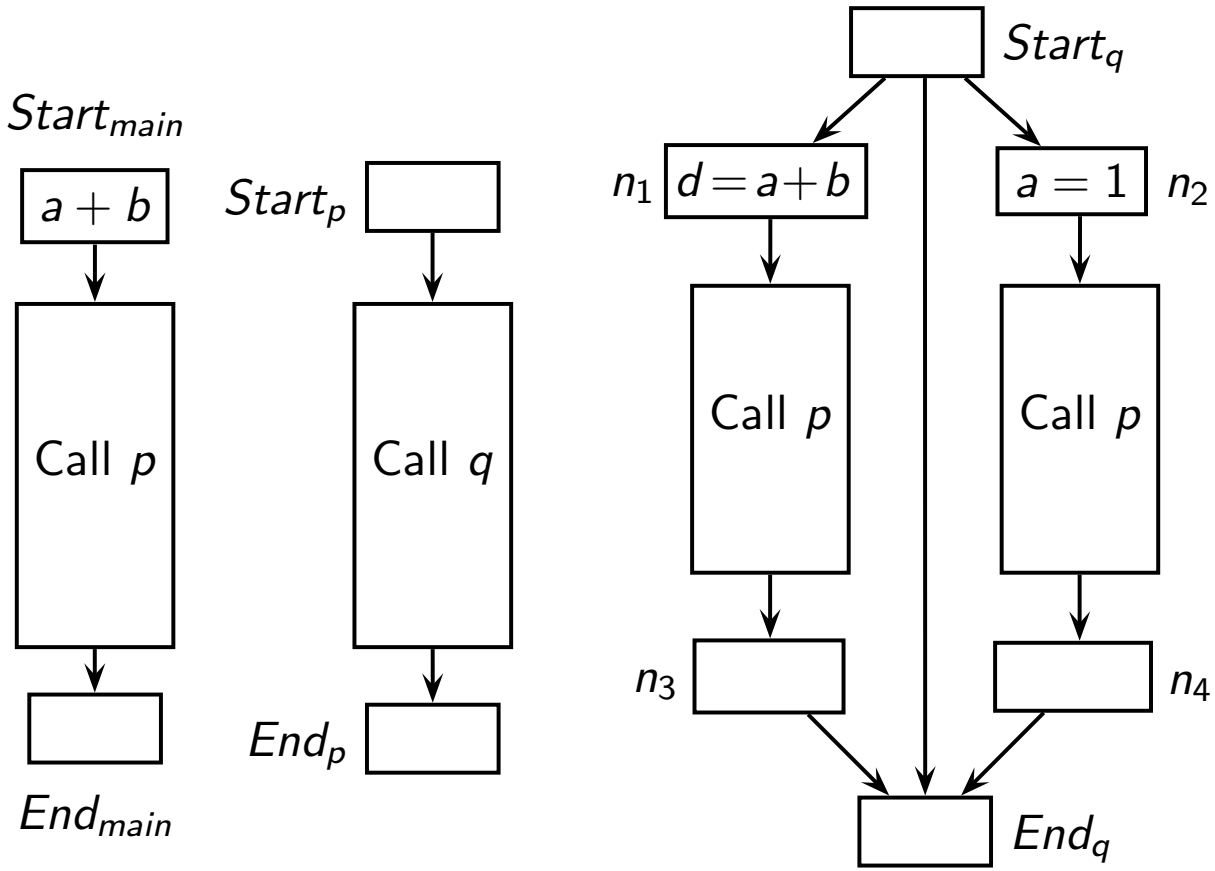


Supergraphs of procedures

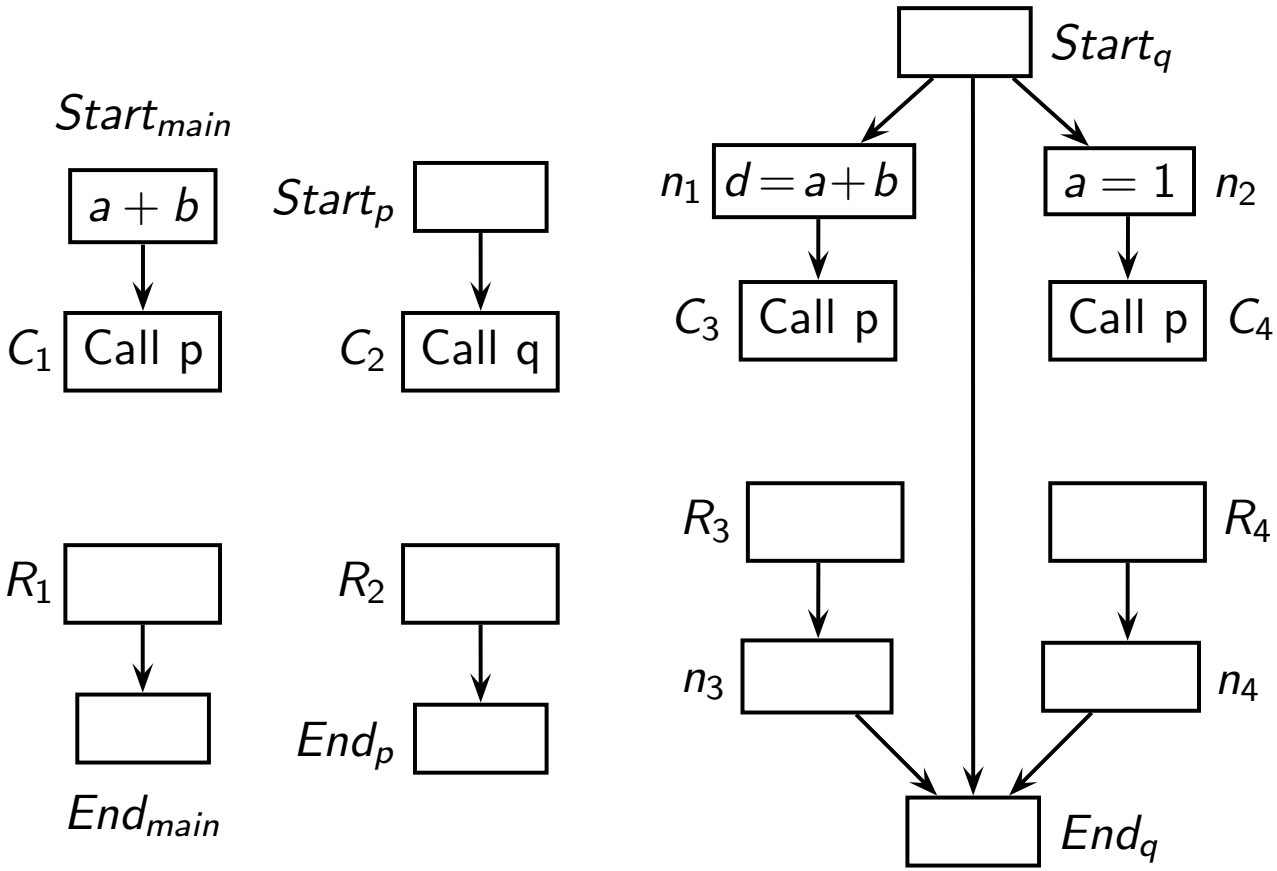


Call multi-graph

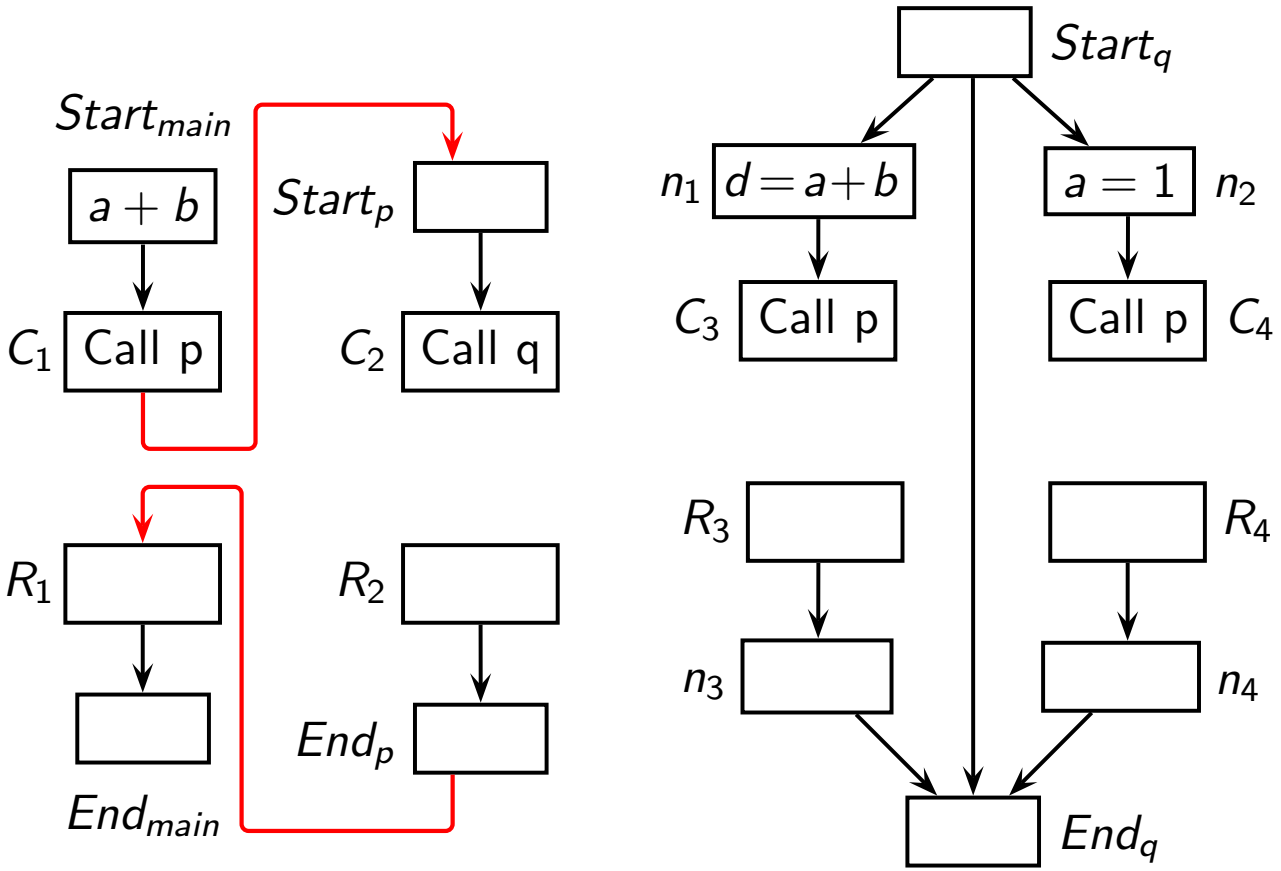
Program Representation for Interprocedural Data Flow Analysis: Supergraph



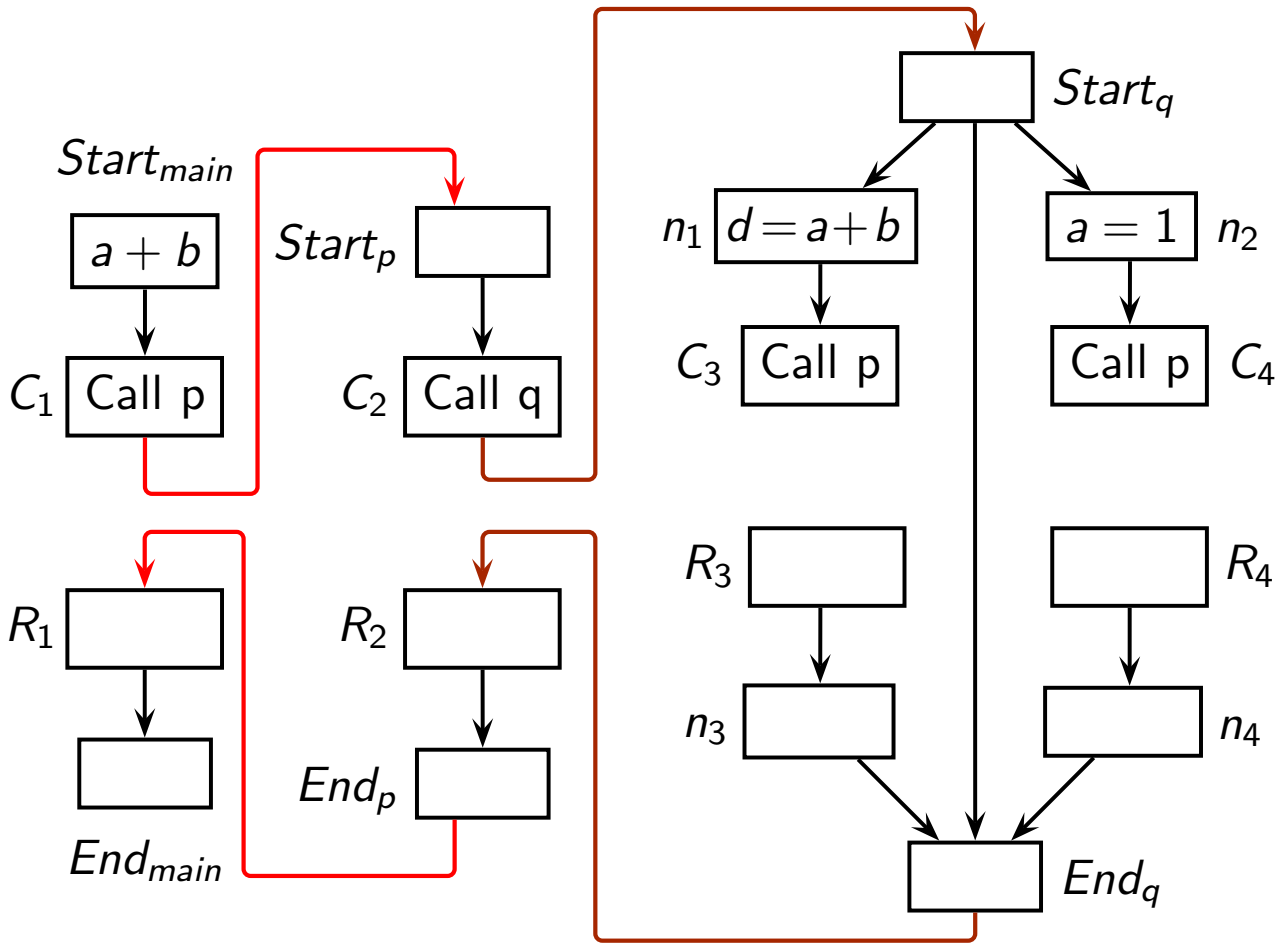
Program Representation for Interprocedural Data Flow Analysis: Supergraph



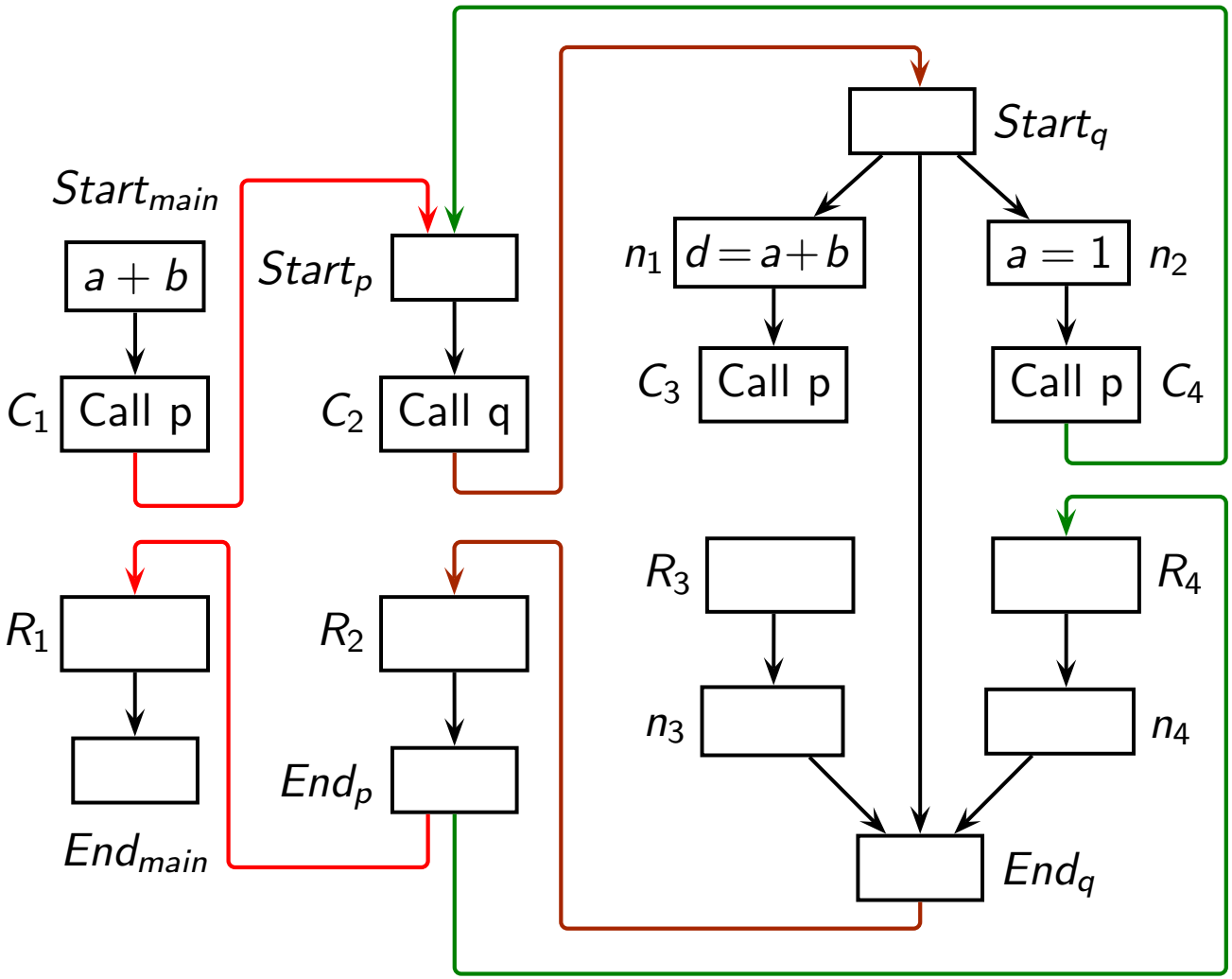
Program Representation for Interprocedural Data Flow Analysis: Supergraph



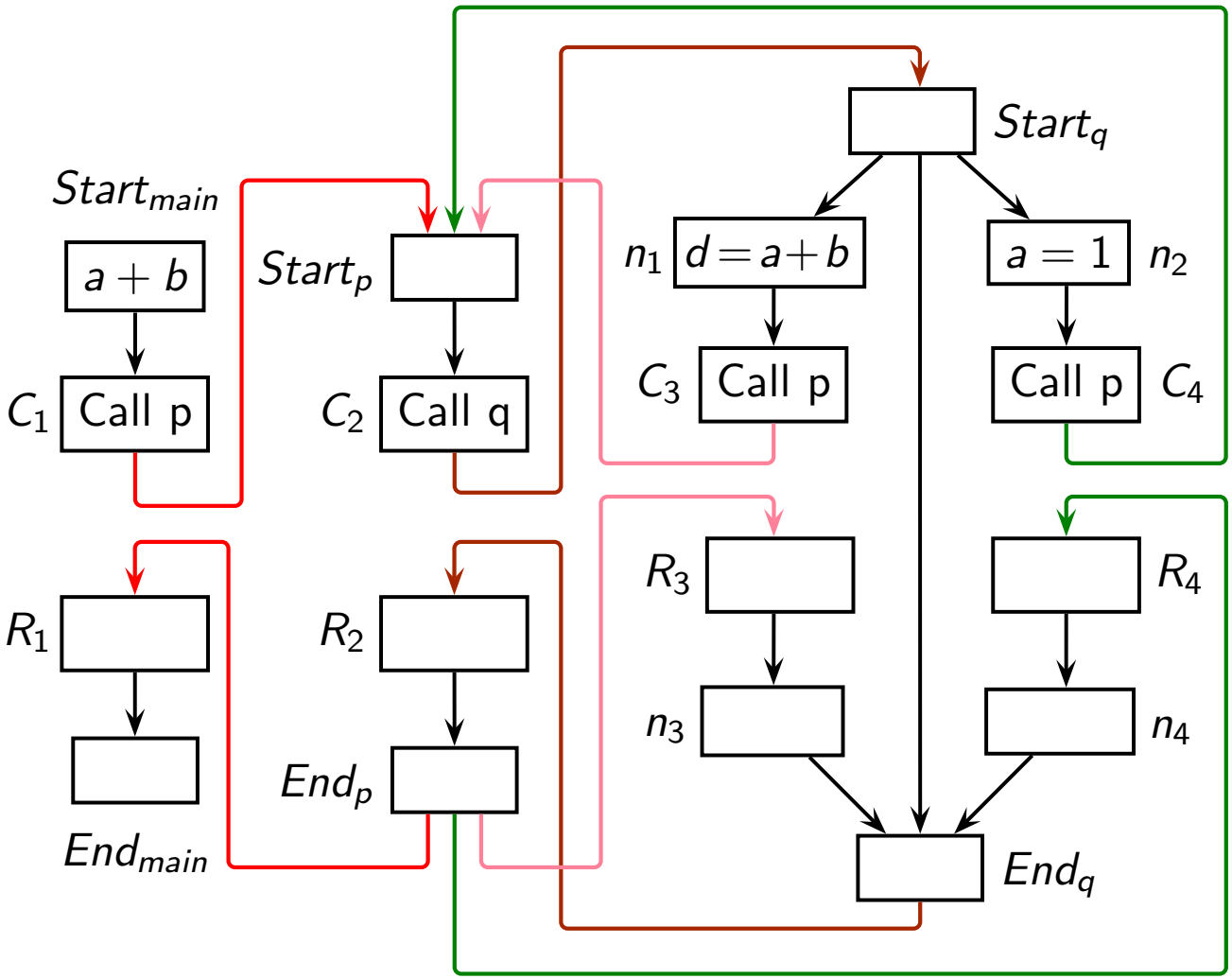
Program Representation for Interprocedural Data Flow Analysis: Supergraph



Program Representation for Interprocedural Data Flow Analysis: Supergraph



Program Representation for Interprocedural Data Flow Analysis: Supergraph



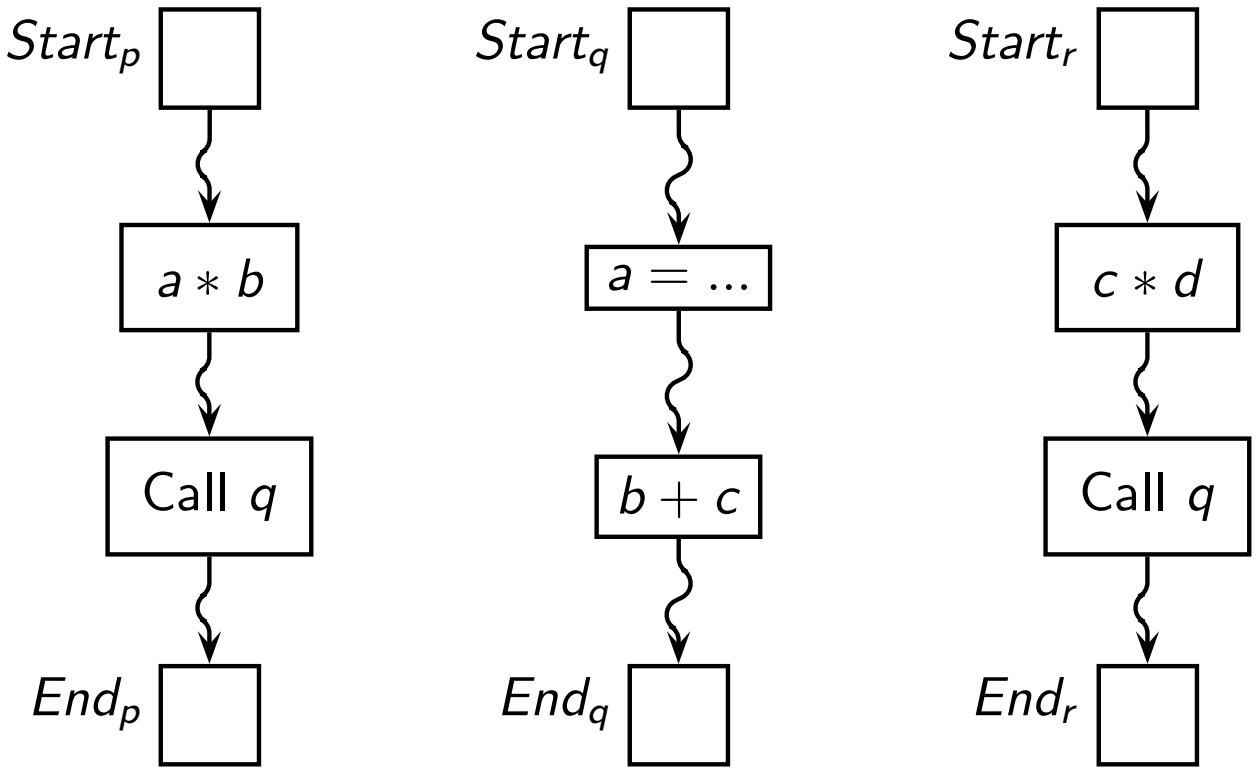
Top-down Vs. Bottom-up Interprocedural Analysis

- Bottom-up approach
  - Traverses the call graph bottom up
  - Computes a parameterized summary of each callee procedure
  - Can be viewed as procedure inlining

    Procedure summary (instead of procedure body) is inlined at call sites
- Top-down approach
  - Traverses the call graph top down
  - Needs to visit a procedure separately for every calling context
  - Number of calling contexts can be exponentially large

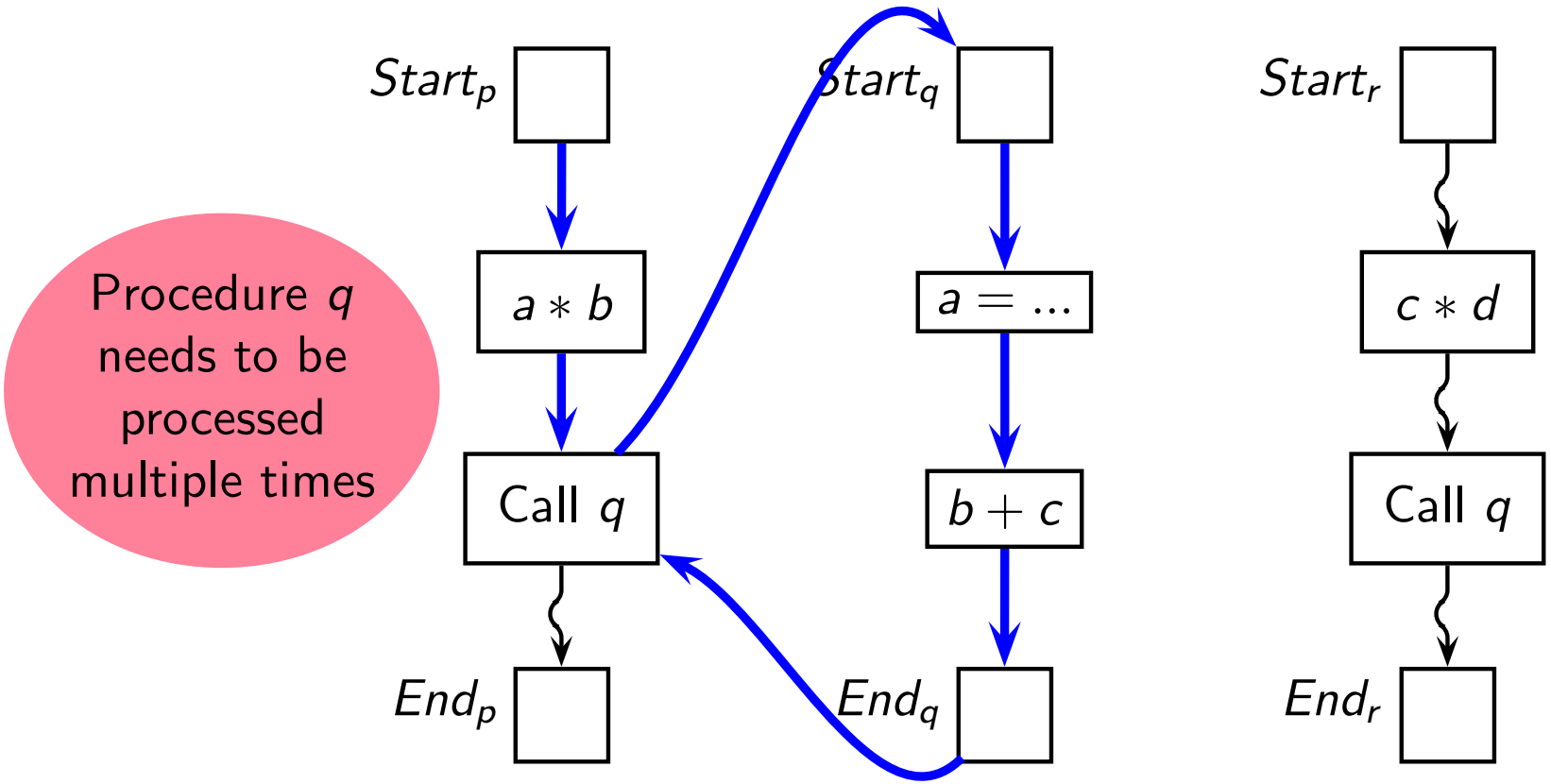
Top-down Vs. Bottom-up Interprocedural Analysis

Top-down Analysis for Available Expressions Analysis



Top-down Vs. Bottom-up Interprocedural Analysis

Top-down Analysis for Available Expressions Analysis

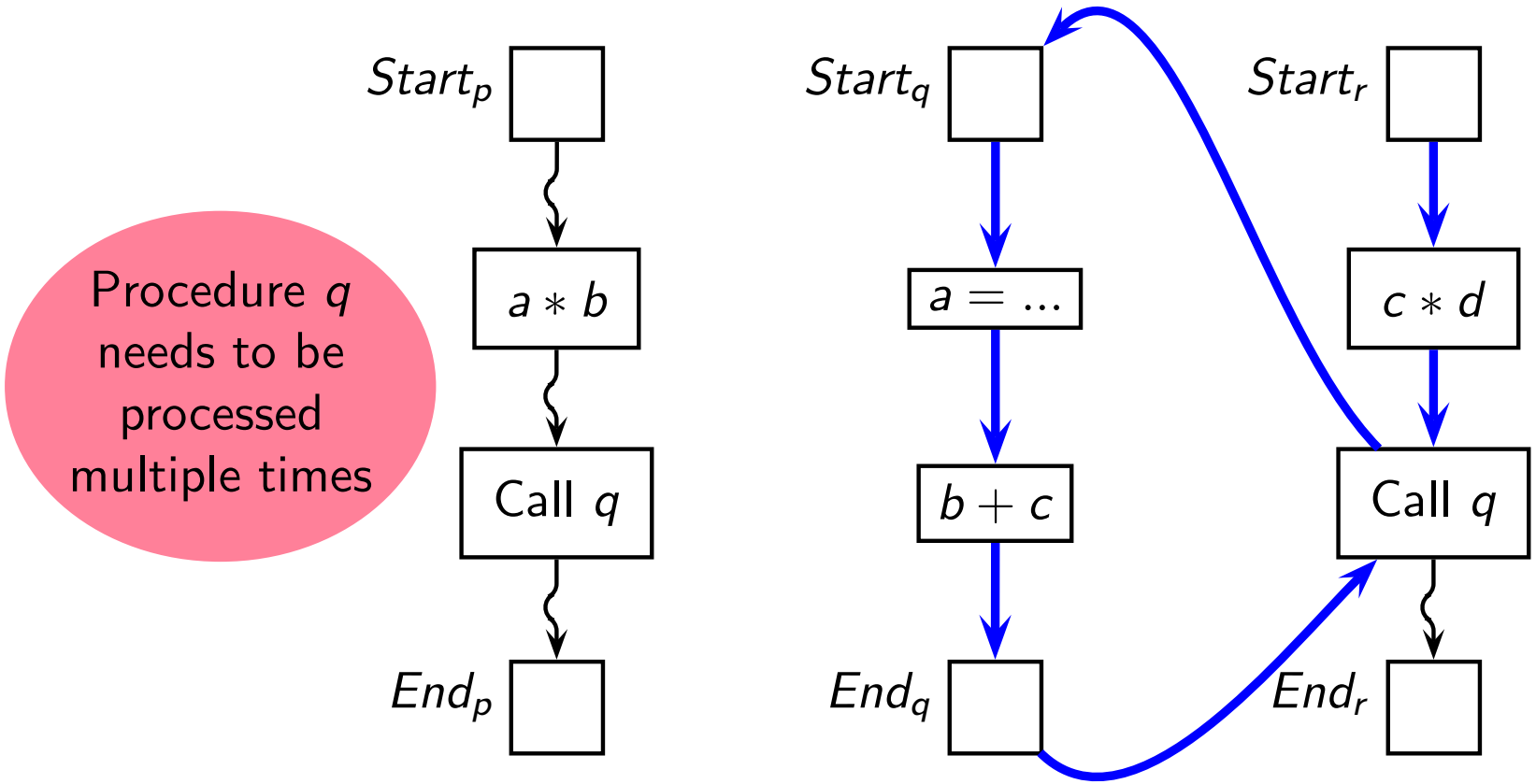


Expression $b + c$ is available in procedure p

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

Top-down Vs. Bottom-up Interprocedural Analysis

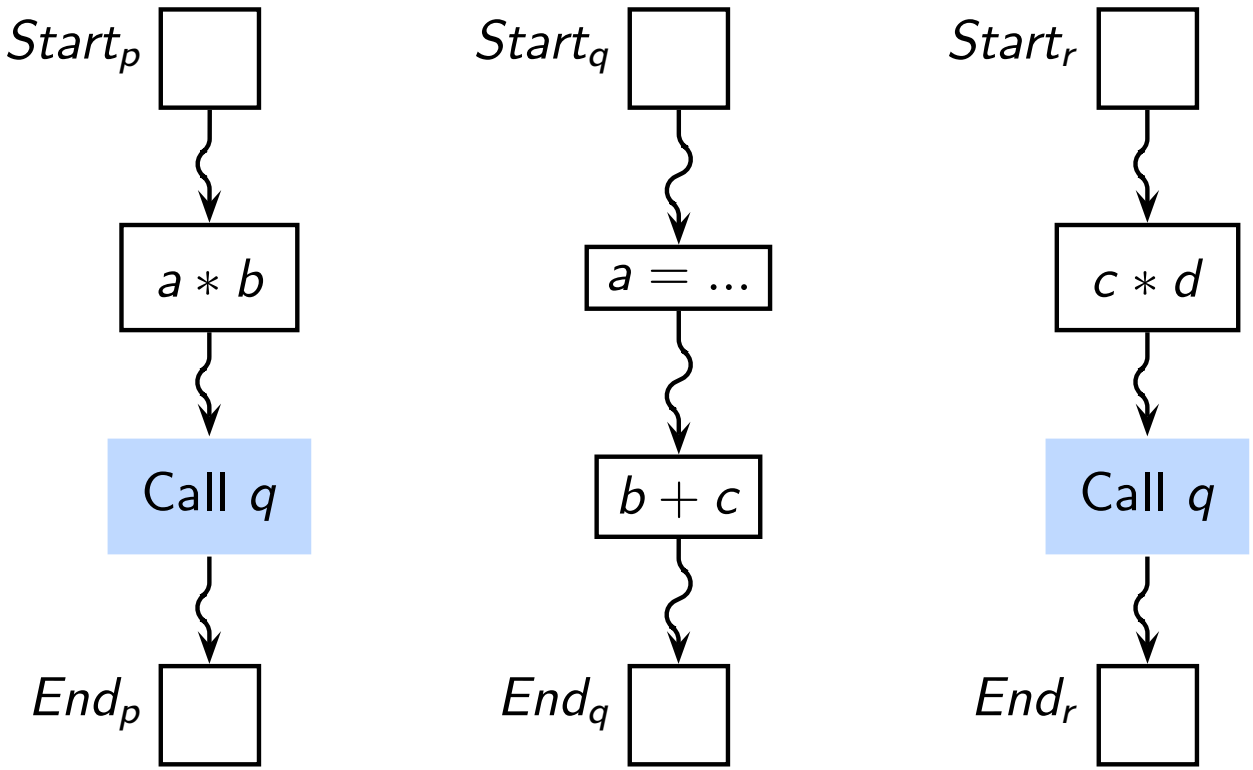
Top-down Analysis for Available Expressions Analysis



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

Top-down Vs. Bottom-up Interprocedural Analysis

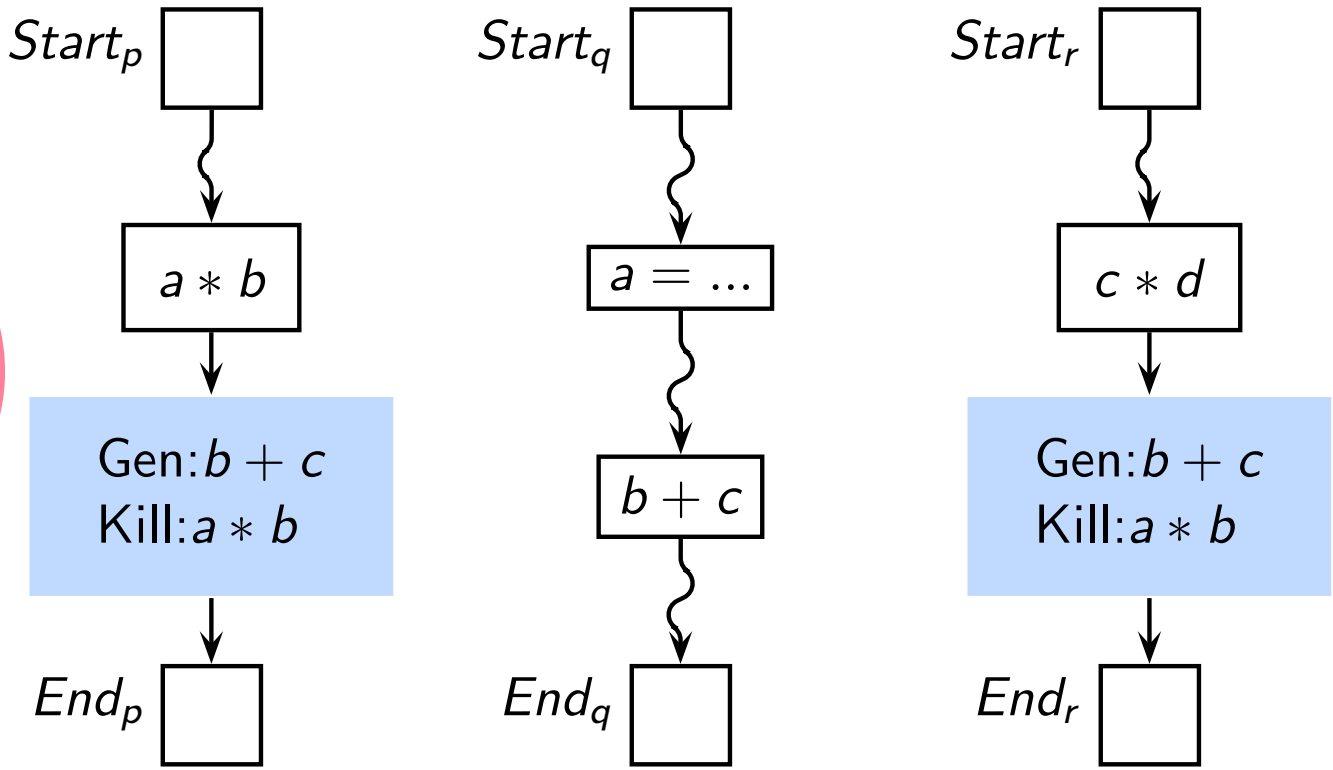
Bottom-Up Analysis for Available Expressions Analysis



Top-down Vs. Bottom-up Interprocedural Analysis

Bottom-Up Analysis for Available Expressions Analysis

Call is replaced by procedure summary

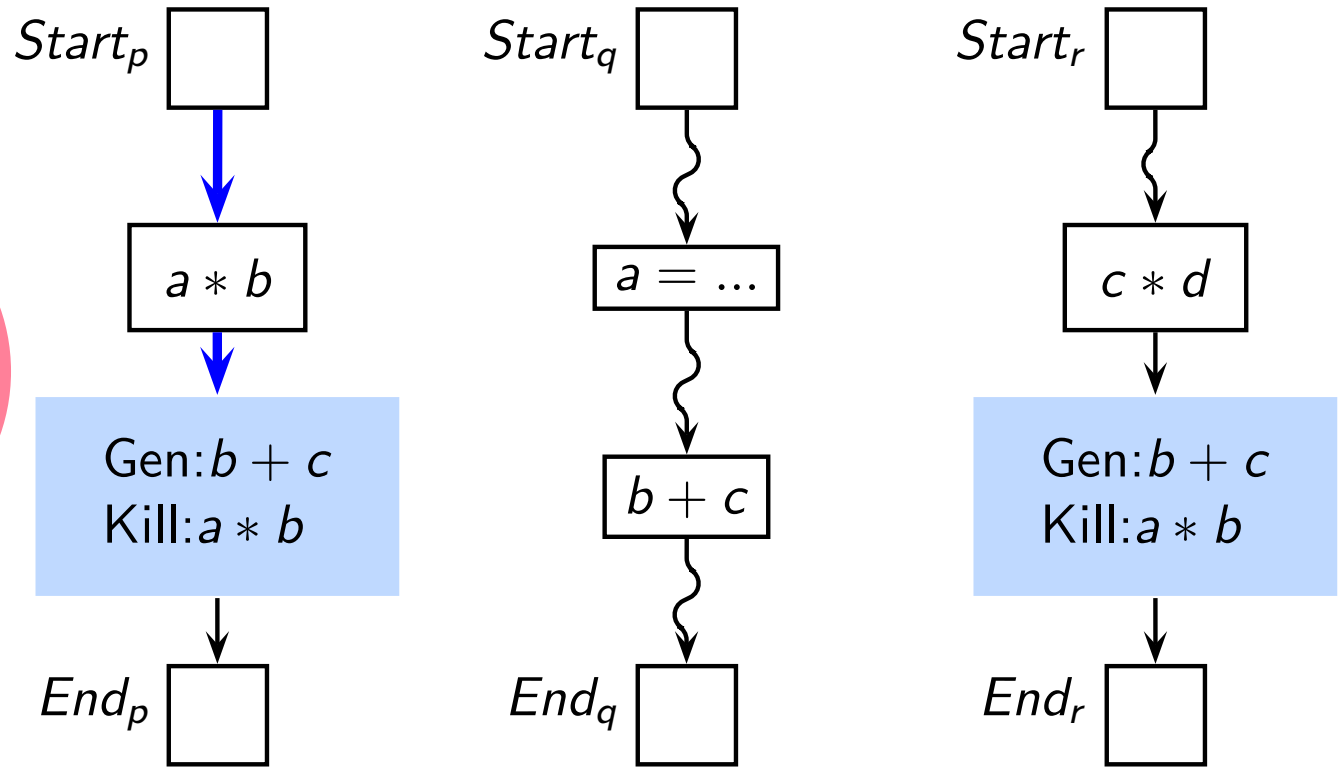


Using procedure summary of g at call sites

Top-down Vs. Bottom-up Interprocedural Analysis

Bottom-Up Analysis for Available Expressions Analysis

Call is replaced by procedure summary



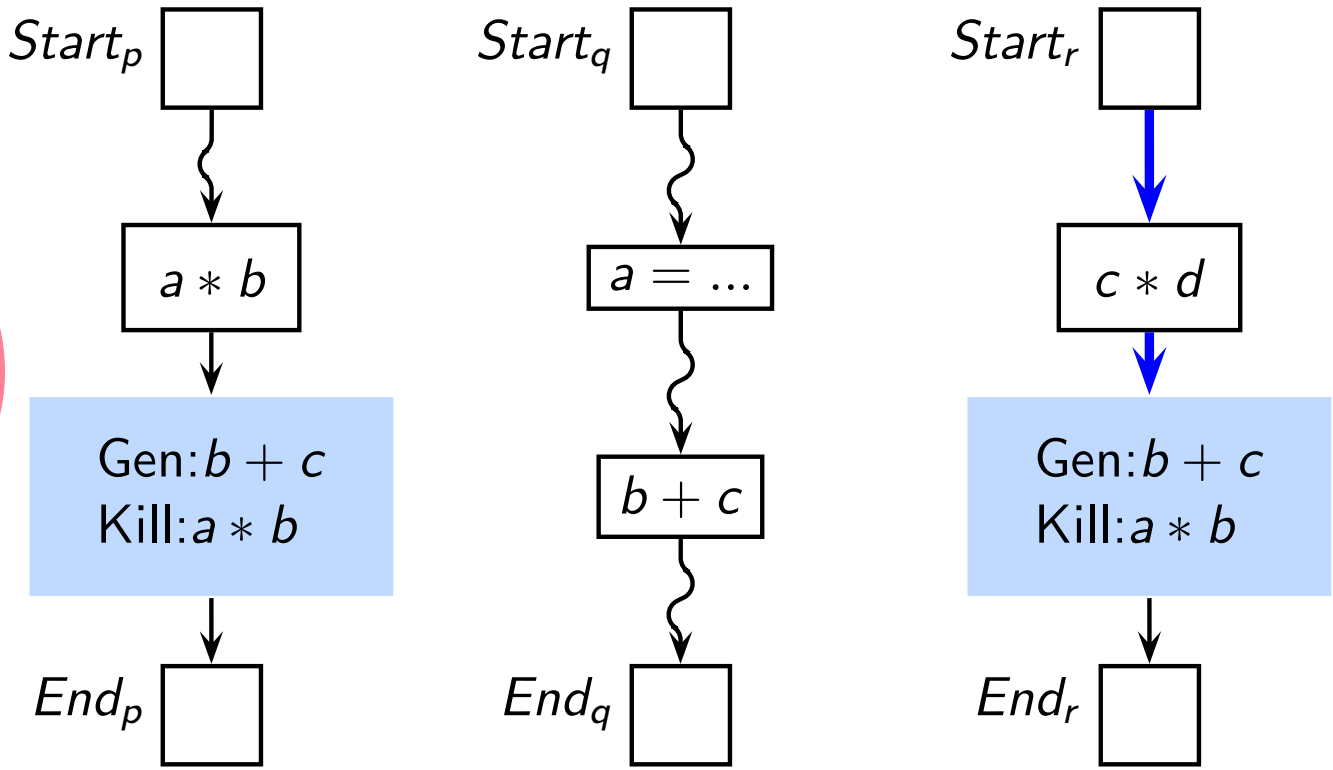
Expression $b + c$ is available in procedure p

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

Top-down Vs. Bottom-up Interprocedural Analysis

Bottom-Up Analysis for Available Expressions Analysis

Call is replaced by procedure summary

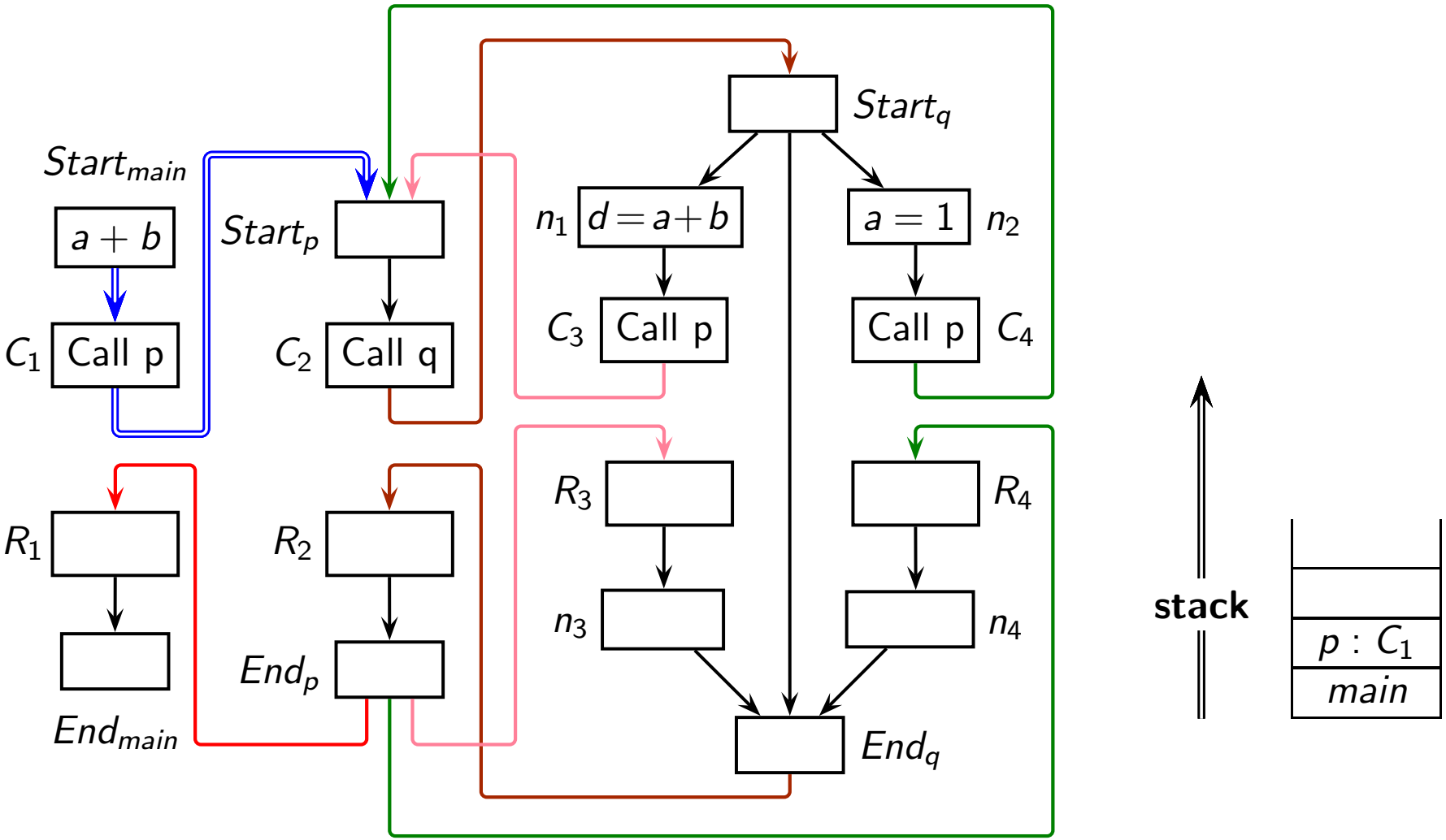


Expressions $b + c$ and $c * d$ are available in procedure r

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

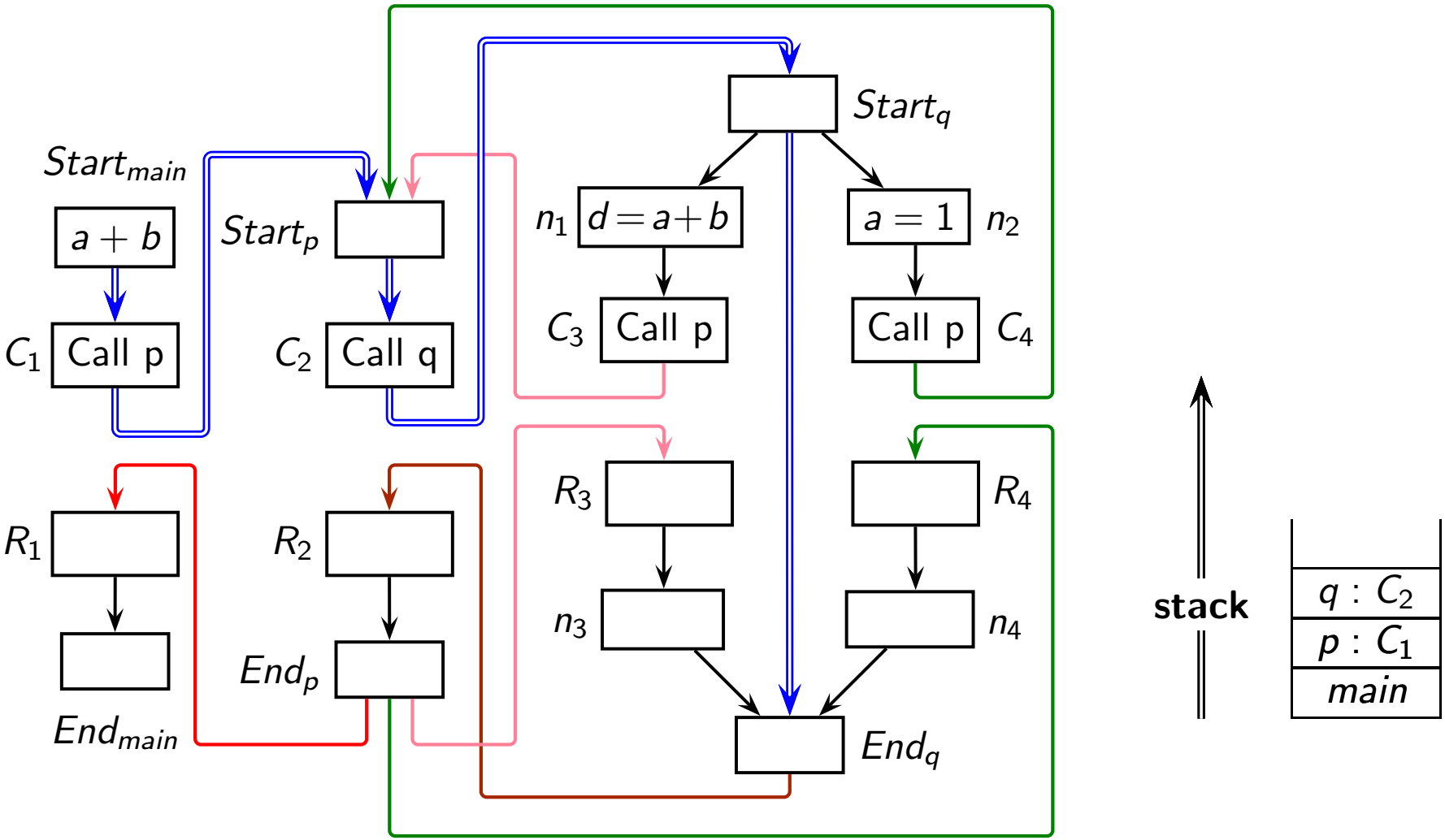
- Bottom-up approach
  - Compact representation
  - Information may depend on the calling context
- Top-down approach
  - Exponentially large number of calling contexts
  - Many contexts may have no effect on the procedure

Validity of Interprocedural Control Flow Paths



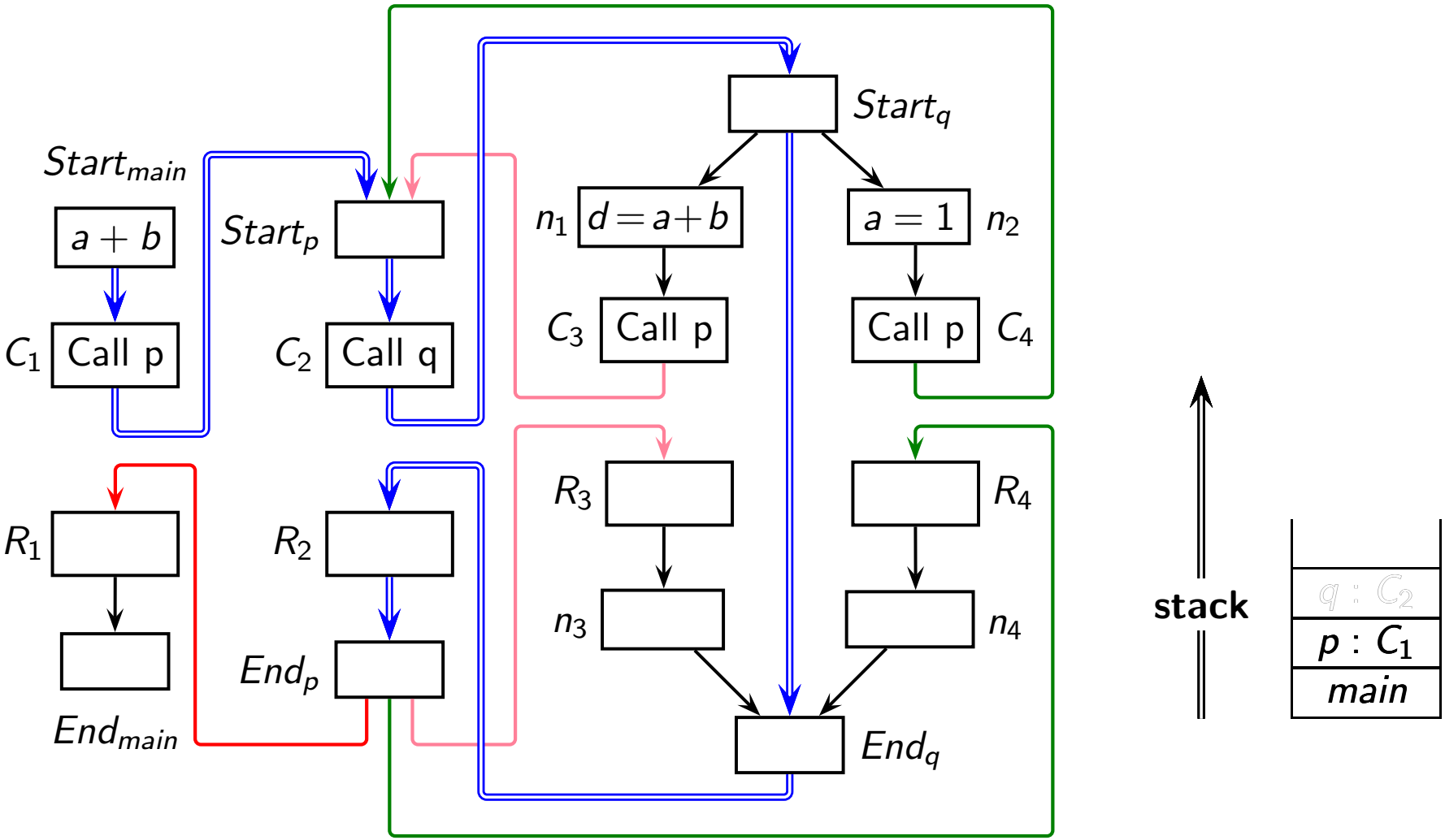
Interprocedurally valid control flow path

Validity of Interprocedural Control Flow Paths



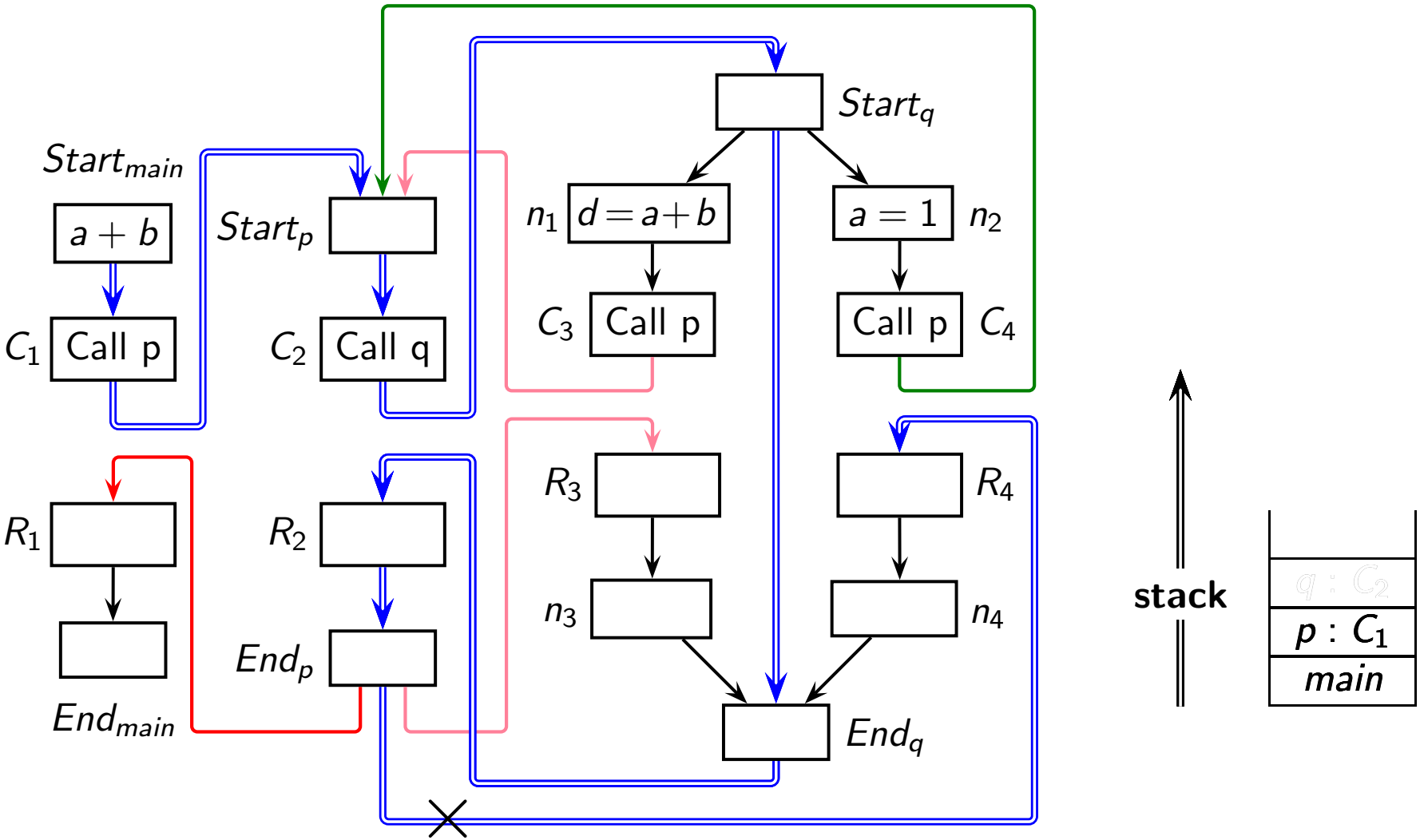
Interprocedurally valid control flow path

Validity of Interprocedural Control Flow Paths



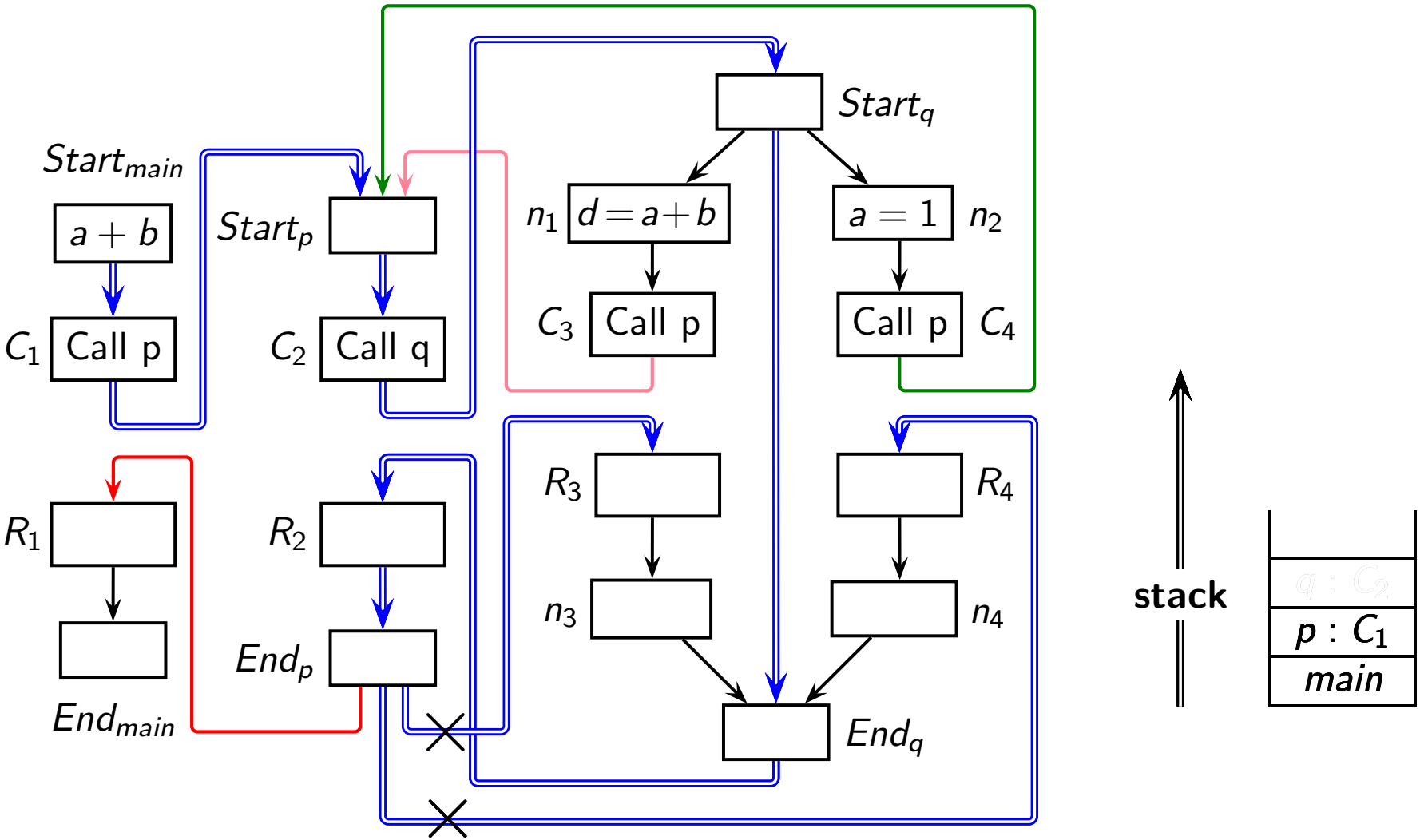
Interprocedurally valid control flow path

Validity of Interprocedural Control Flow Paths



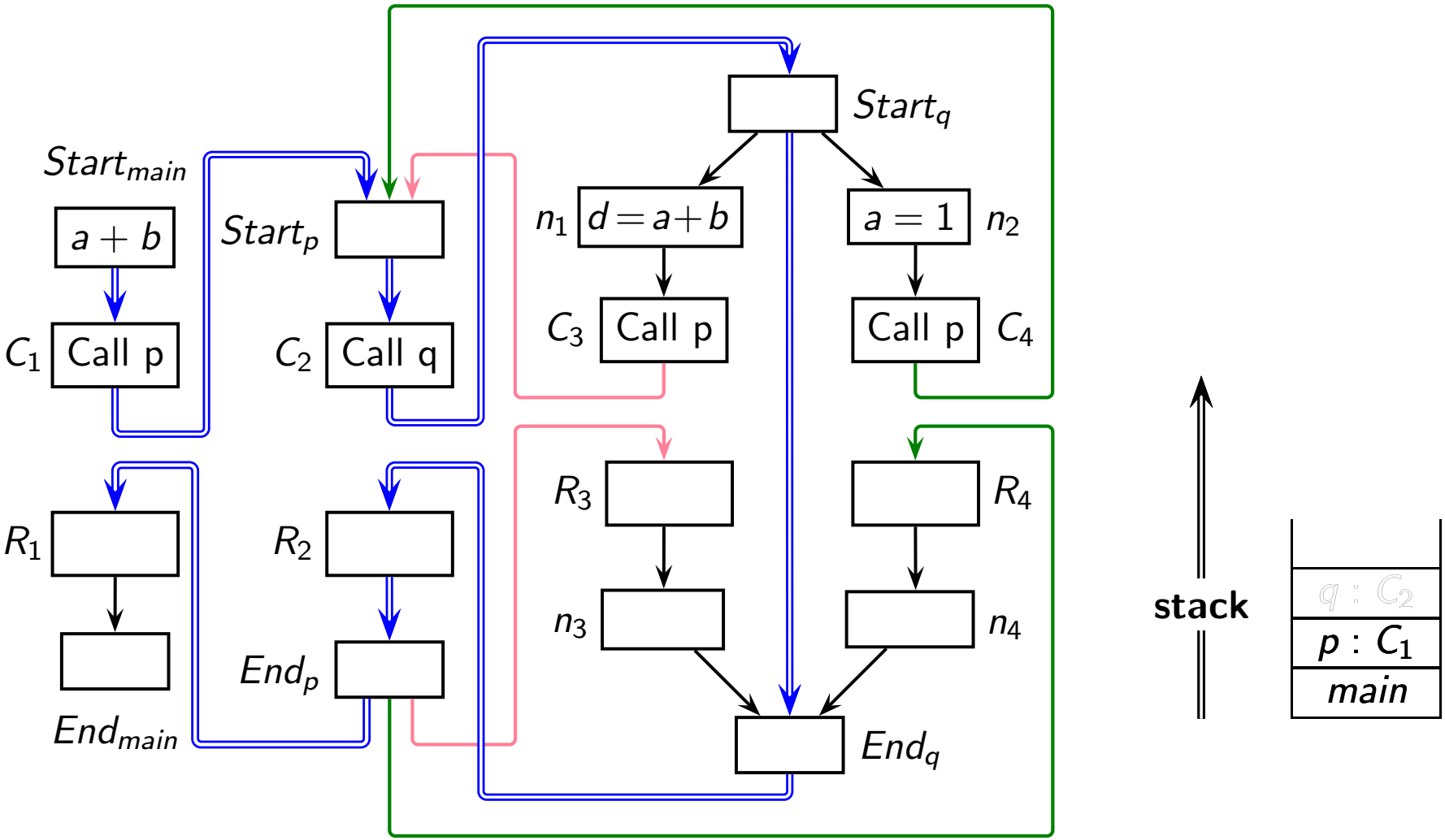
Interprocedurally invalid control flow path

Validity of Interprocedural Control Flow Paths



Interprocedurally invalid control flow path

Validity of Interprocedural Control Flow Paths



Interprocedurally valid control flow path

Soundness, Precision, and Efficiency of Data Flow Analysis

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

Soundness, Precision, and Efficiency of Data Flow Analysis

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

Soundness, Precision, and Efficiency of Data Flow Analysis

A path which represents legal control flow

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

Soundness, Precision, and Efficiency of Data Flow Analysis

A path which represents legal control flow

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

Soundness, Precision, and Efficiency of Data Flow Analysis

A path which represents legal control flow

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

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

Soundness, Precision, and Efficiency of Data Flow Analysis

A path which represents legal control flow

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

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

Soundness, Precision, and Efficiency of Data Flow Analysis

A path which represents legal control flow

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

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

A path which yields information that affects the summary information

Flow and Context Sensitivity

- Flow sensitive analysis:

  Considers intraprocedurally valid paths

Flow and Context Sensitivity

- Flow sensitive analysis:

  Considers intraprocedurally valid paths

- Context sensitive analysis:

  Considers interprocedurally valid paths

Flow and Context Sensitivity

- Flow sensitive analysis:

  Considers intraprocedurally valid paths

- Context sensitive analysis:

  Considers interprocedurally valid paths

- For maximum statically attainable precision, analysis must be both flow and context sensitive

Flow and Context Sensitivity

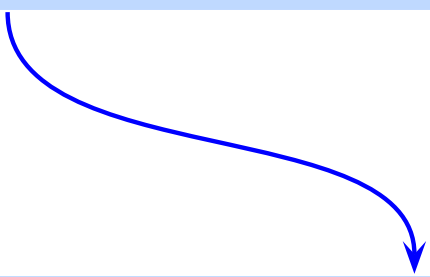
- Flow sensitive analysis:

  Considers intraprocedurally valid paths

- Context sensitive analysis:

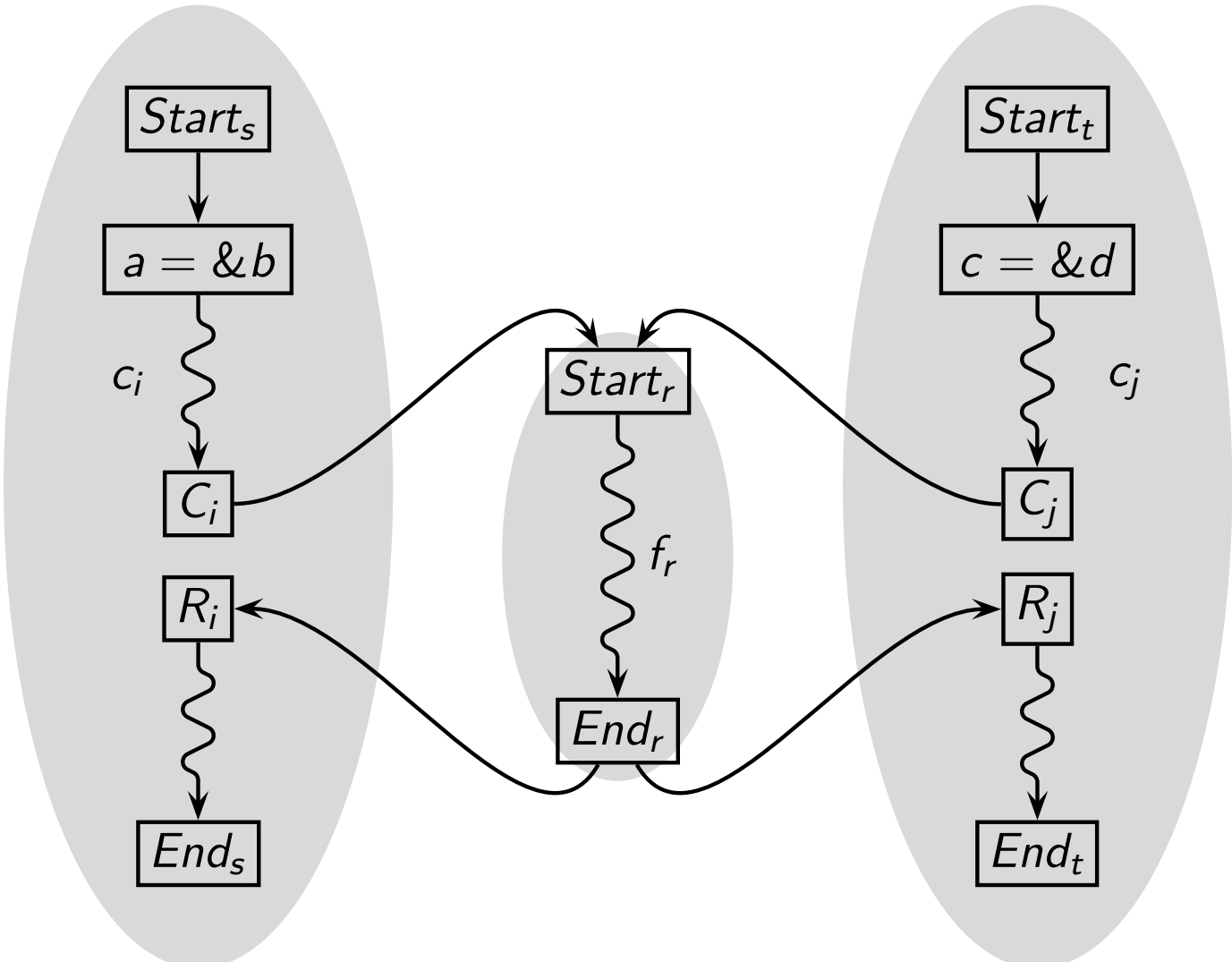
  Considers interprocedurally valid paths

- For maximum statically attainable precision, analysis must be both flow and context sensitive

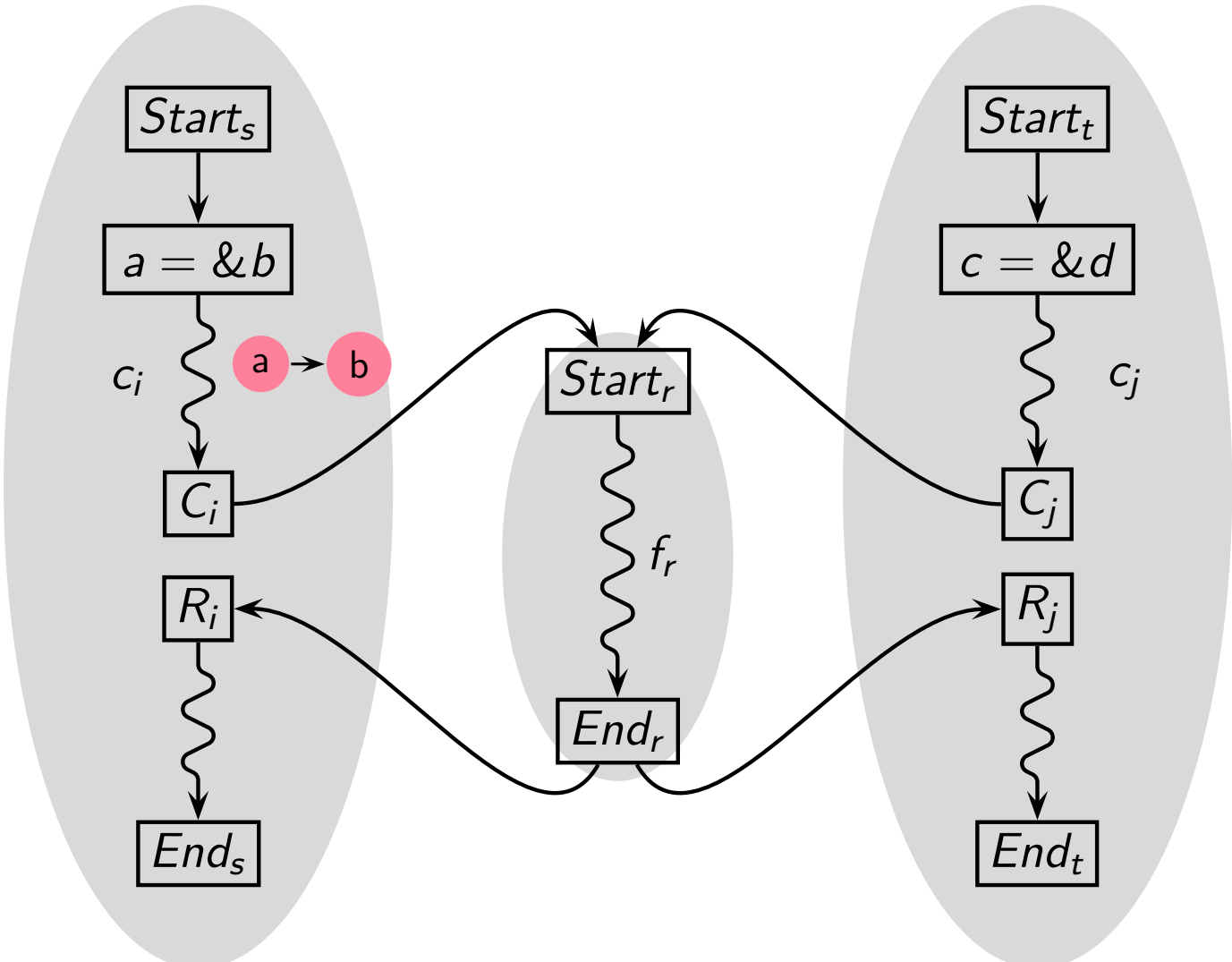


MFP computation restricted to valid paths only

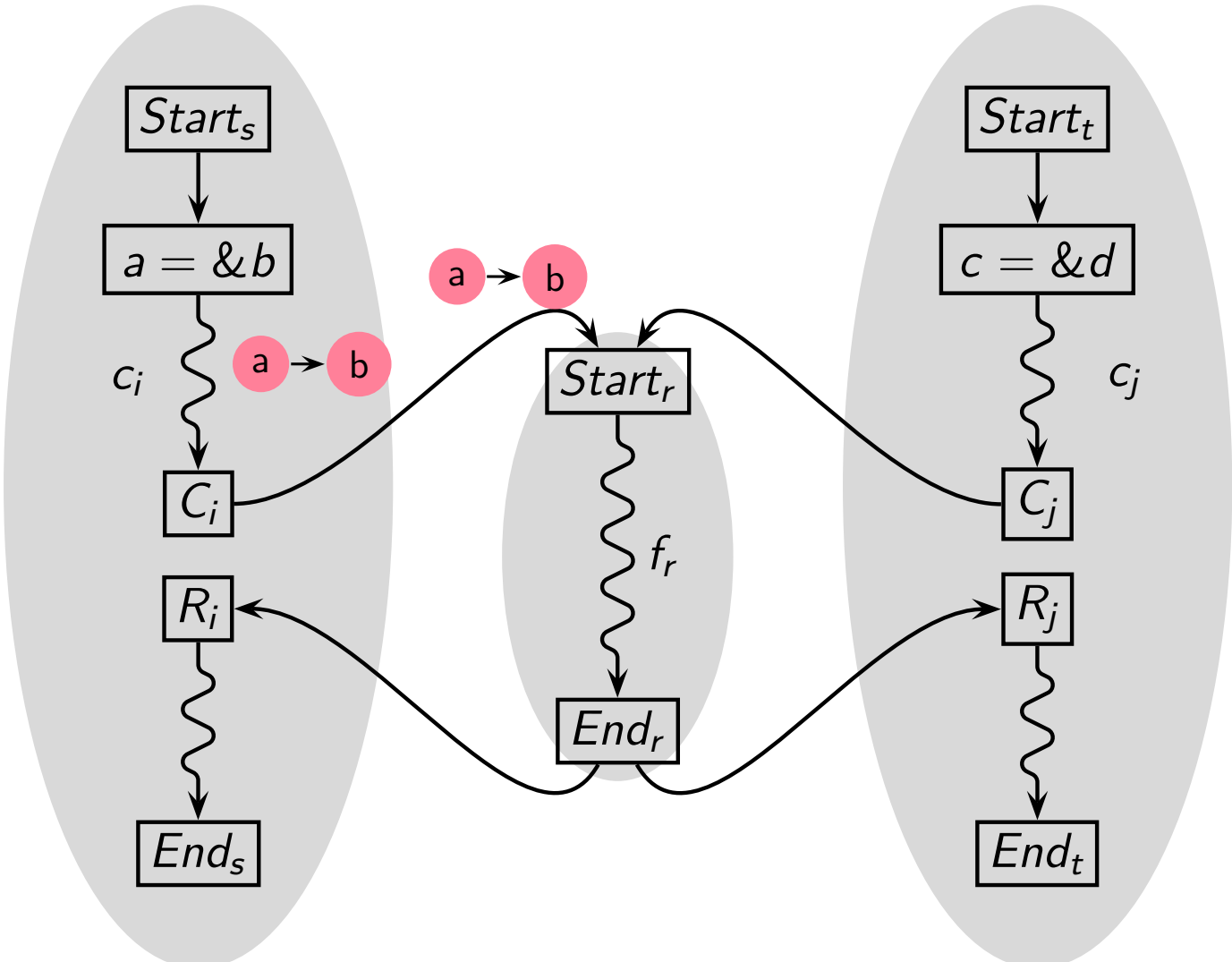
Context Sensitivity in Interprocedural Analysis



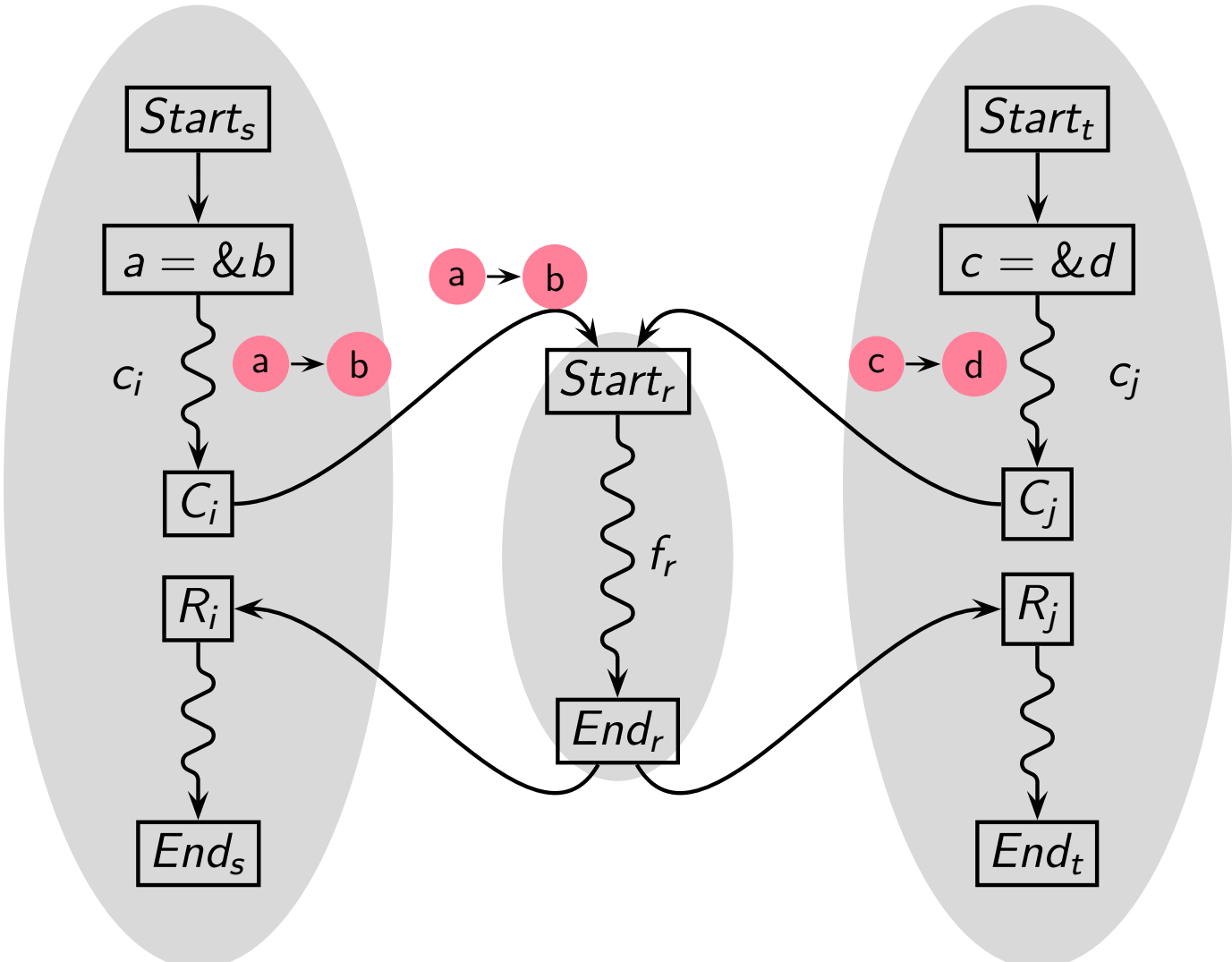
Context Sensitivity in Interprocedural Analysis



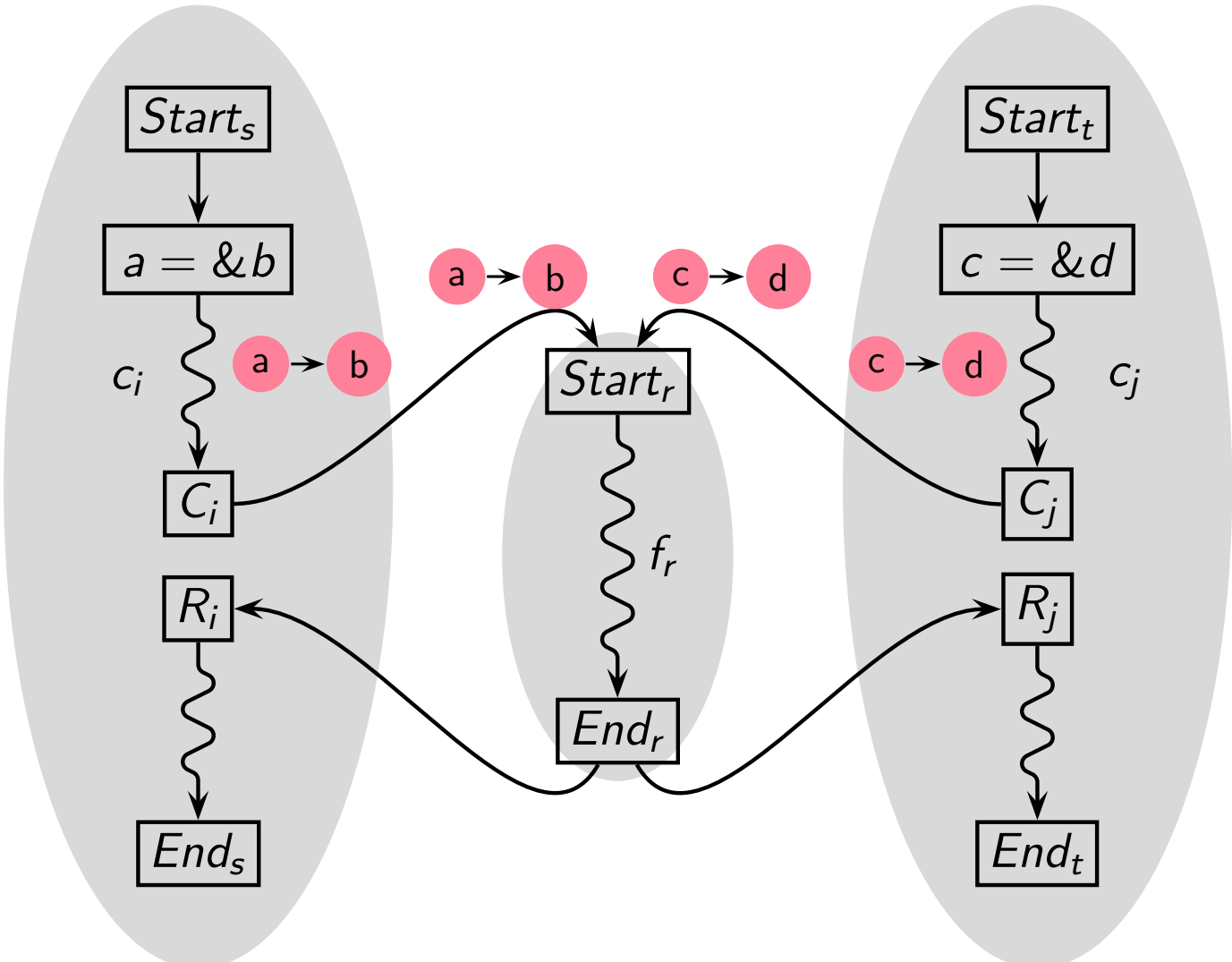
Context Sensitivity in Interprocedural Analysis



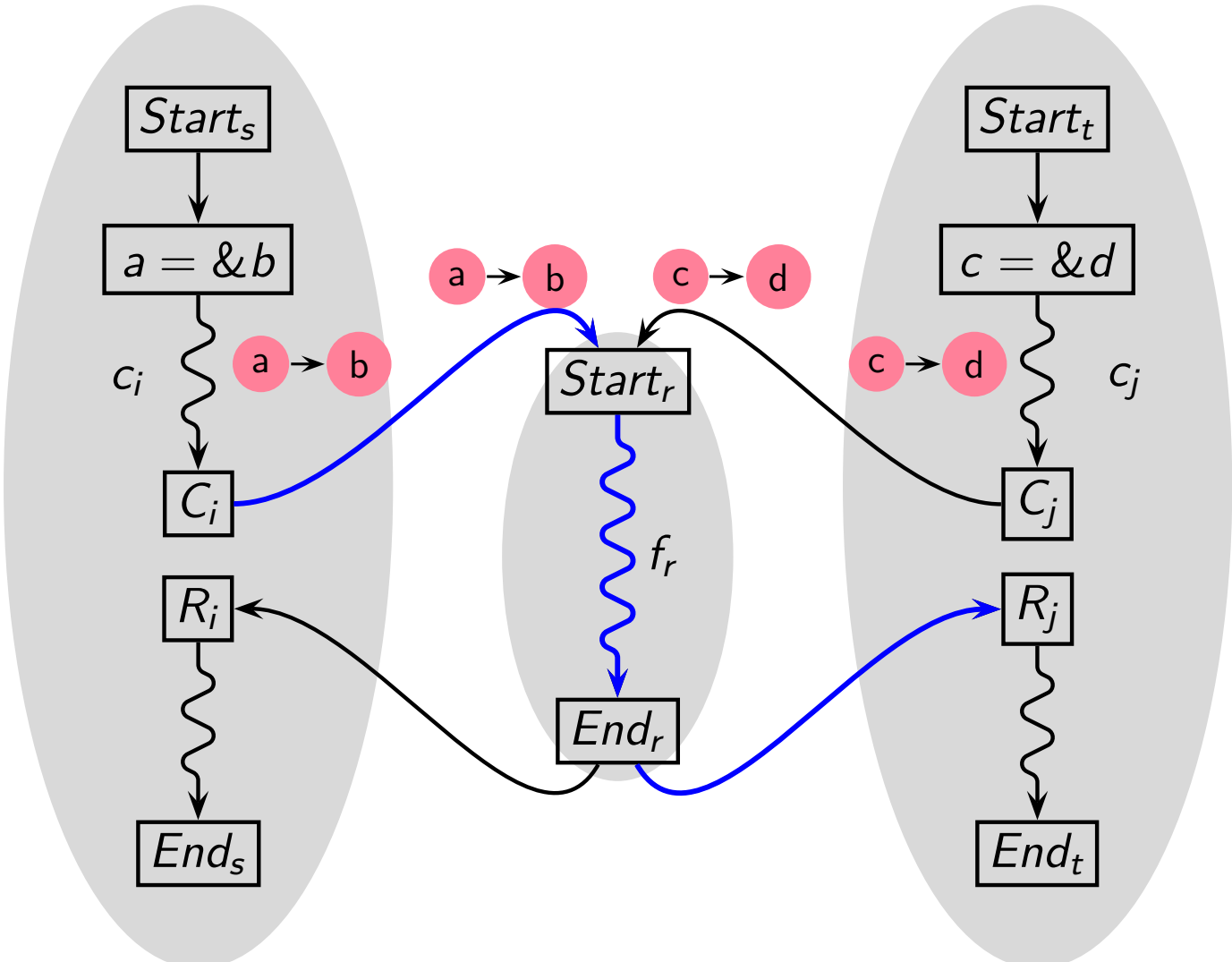
Context Sensitivity in Interprocedural Analysis



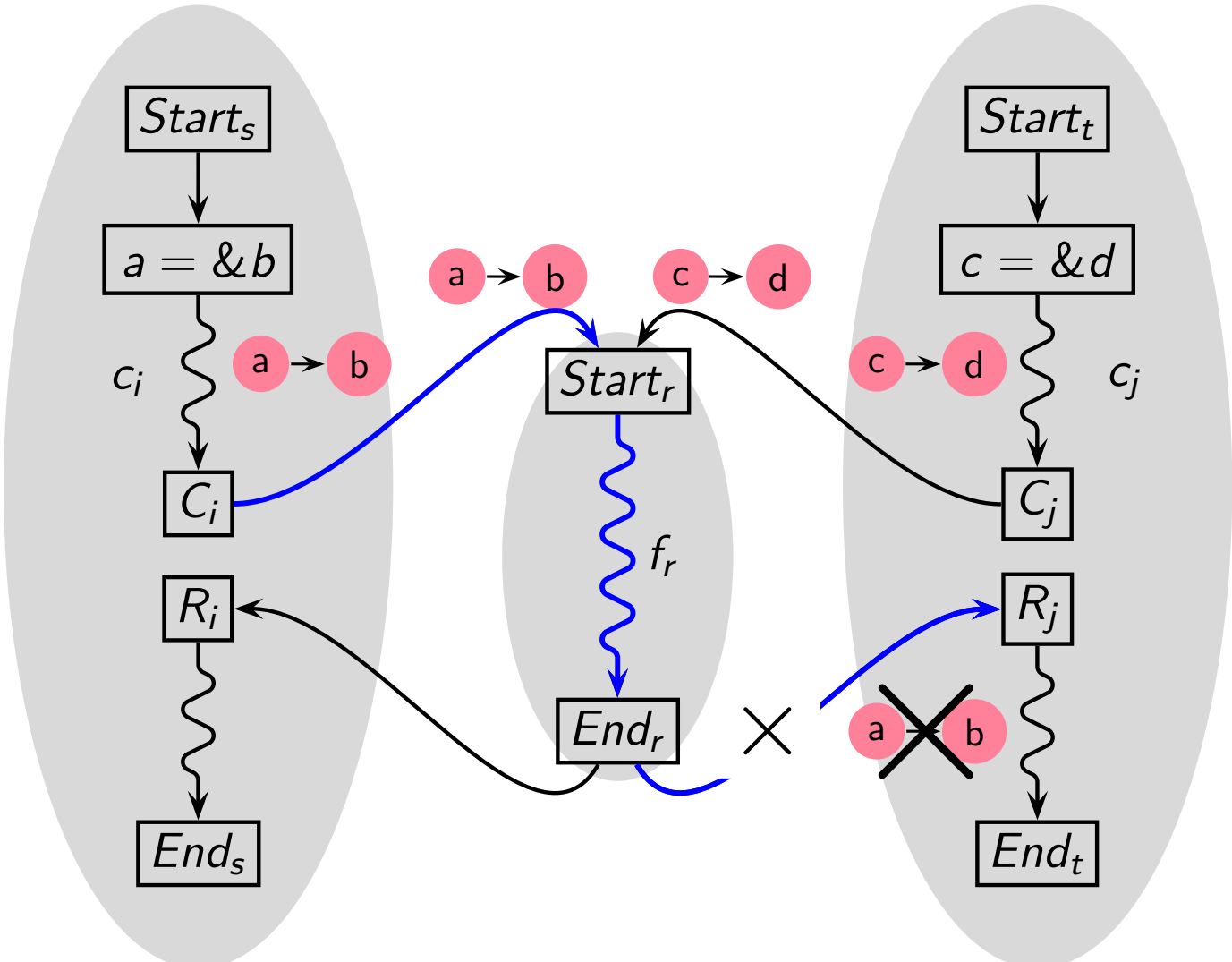
Context Sensitivity in Interprocedural Analysis



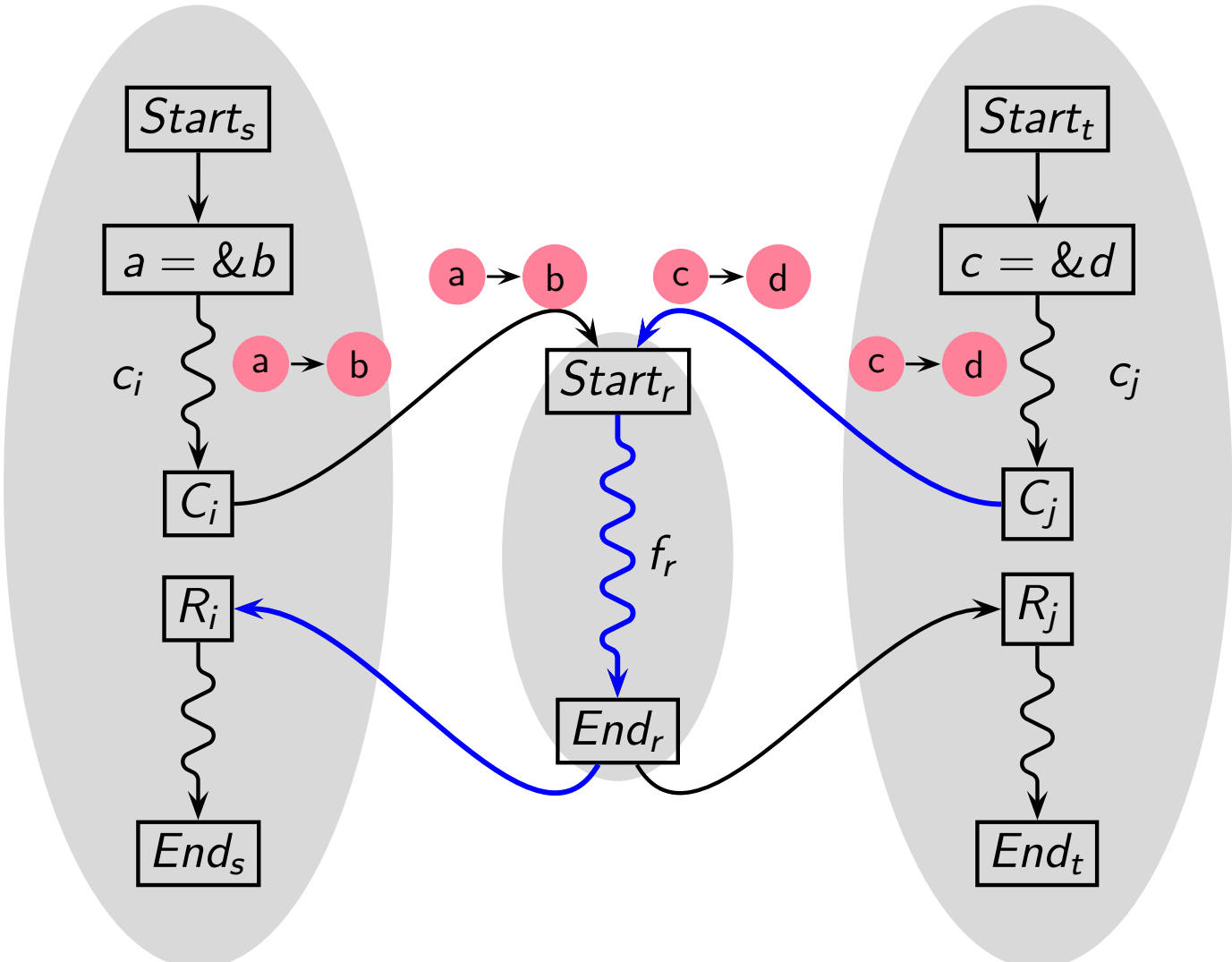
Context Sensitivity in Interprocedural Analysis



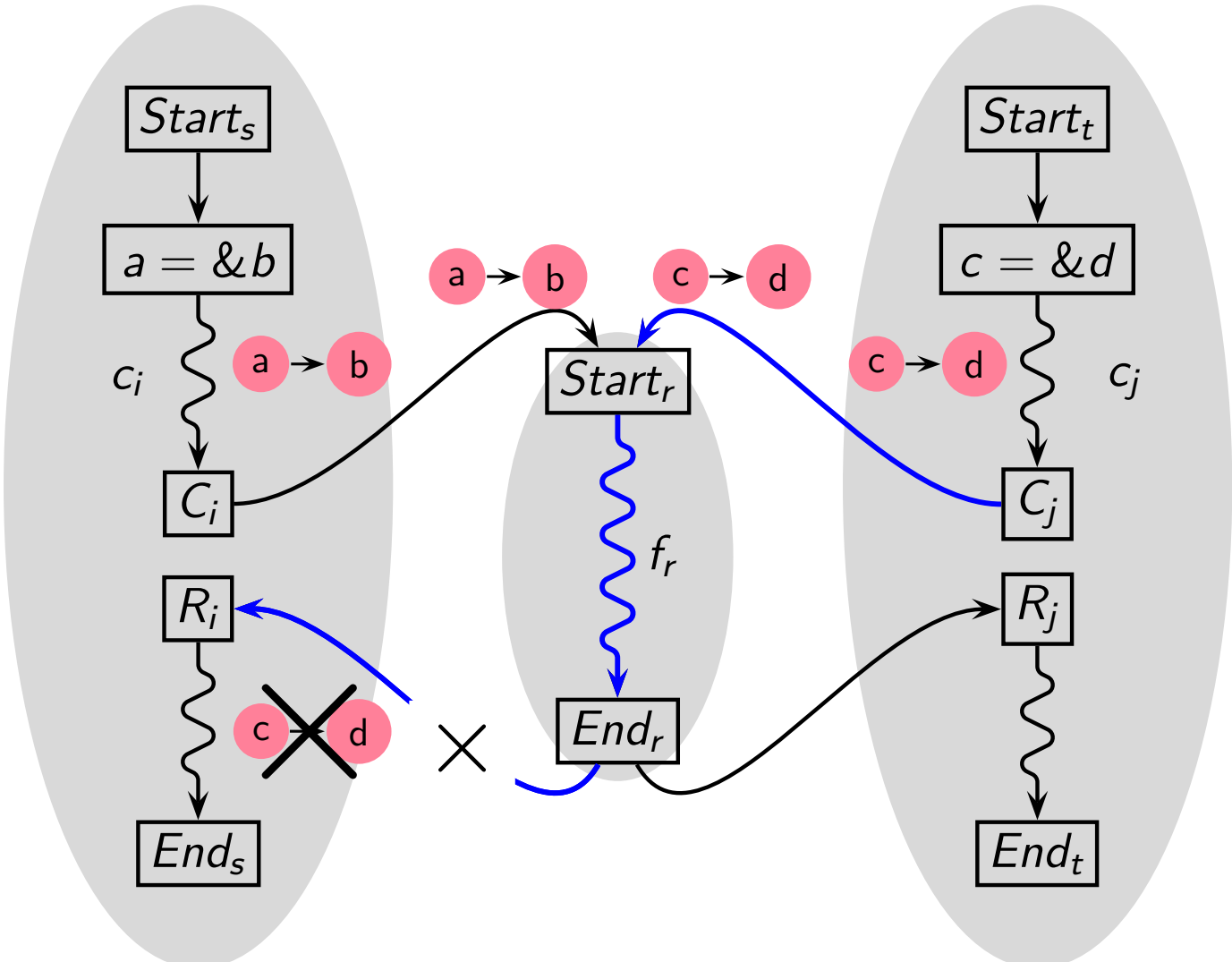
Context Sensitivity in Interprocedural Analysis



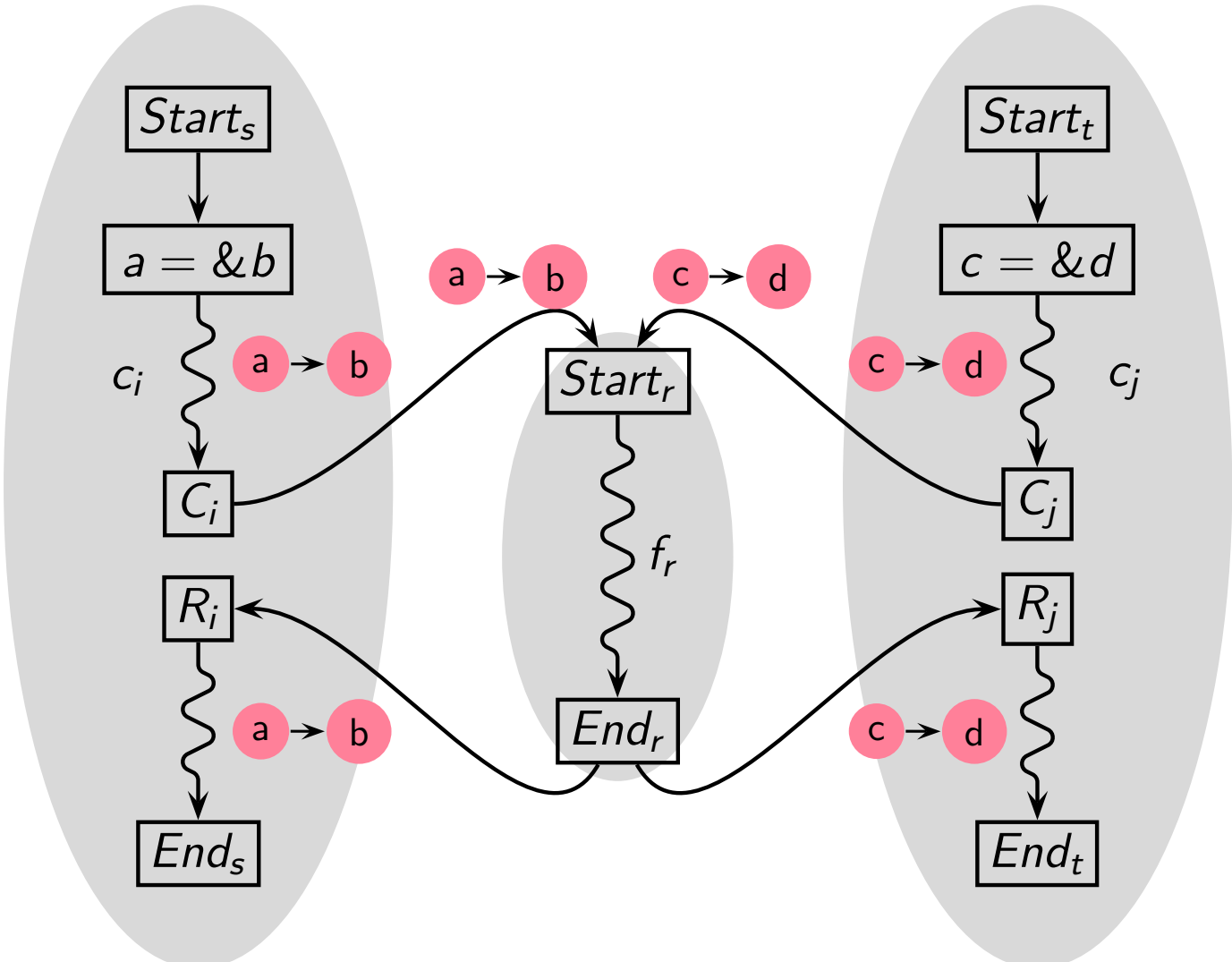
Context Sensitivity in Interprocedural Analysis



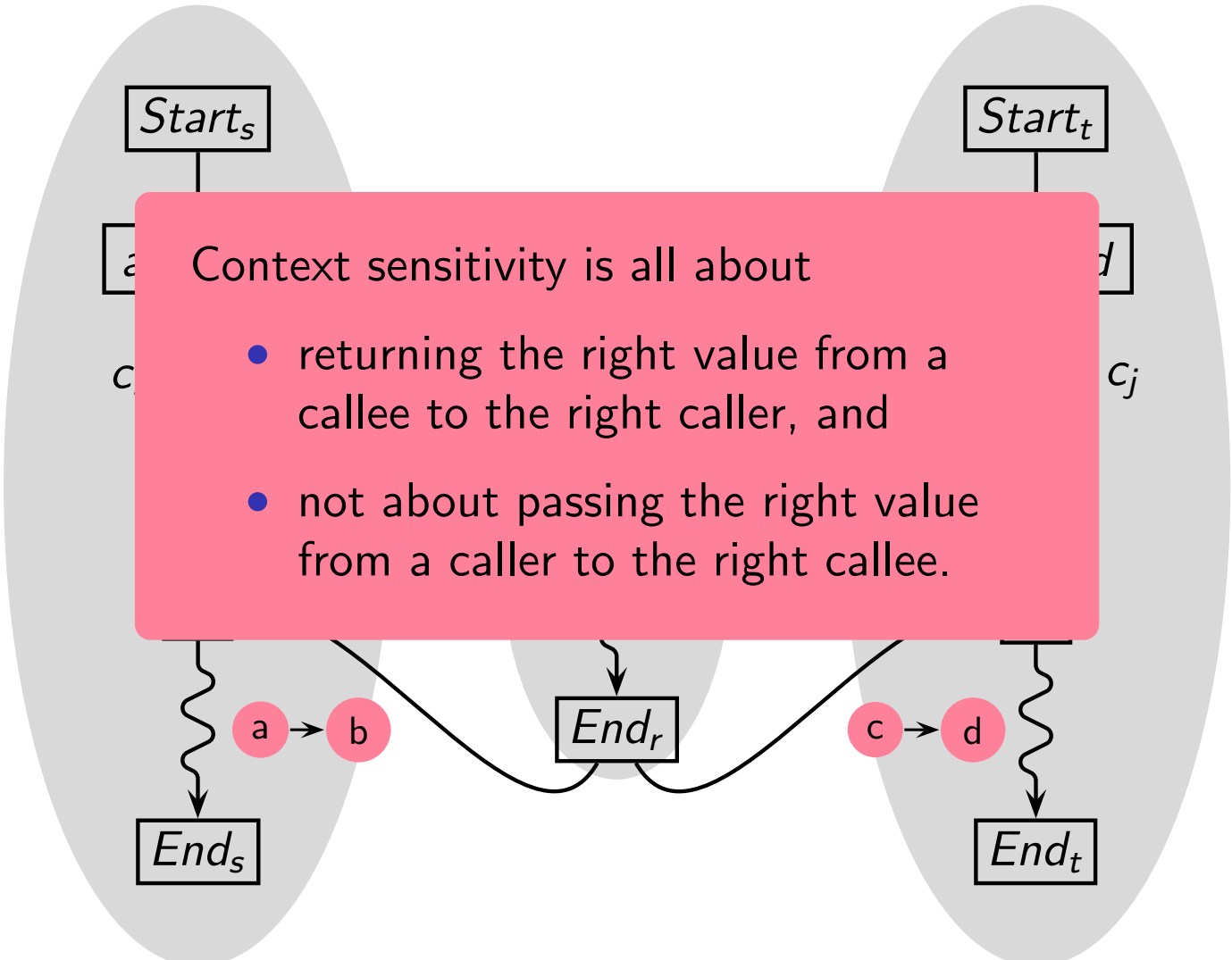
Context Sensitivity in Interprocedural Analysis



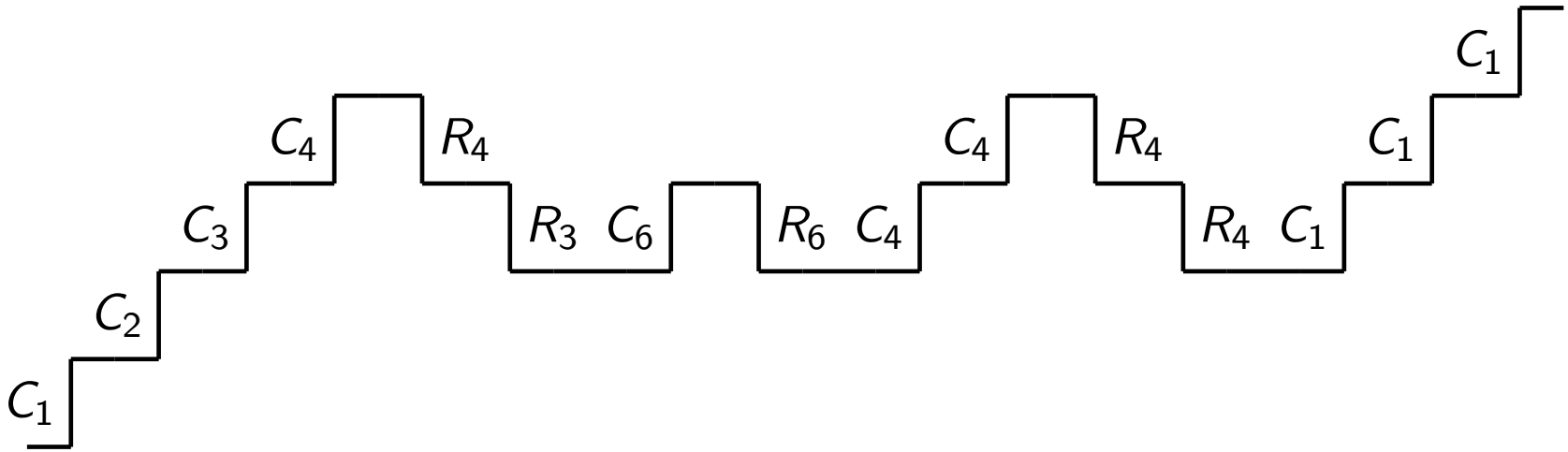
Context Sensitivity in Interprocedural Analysis



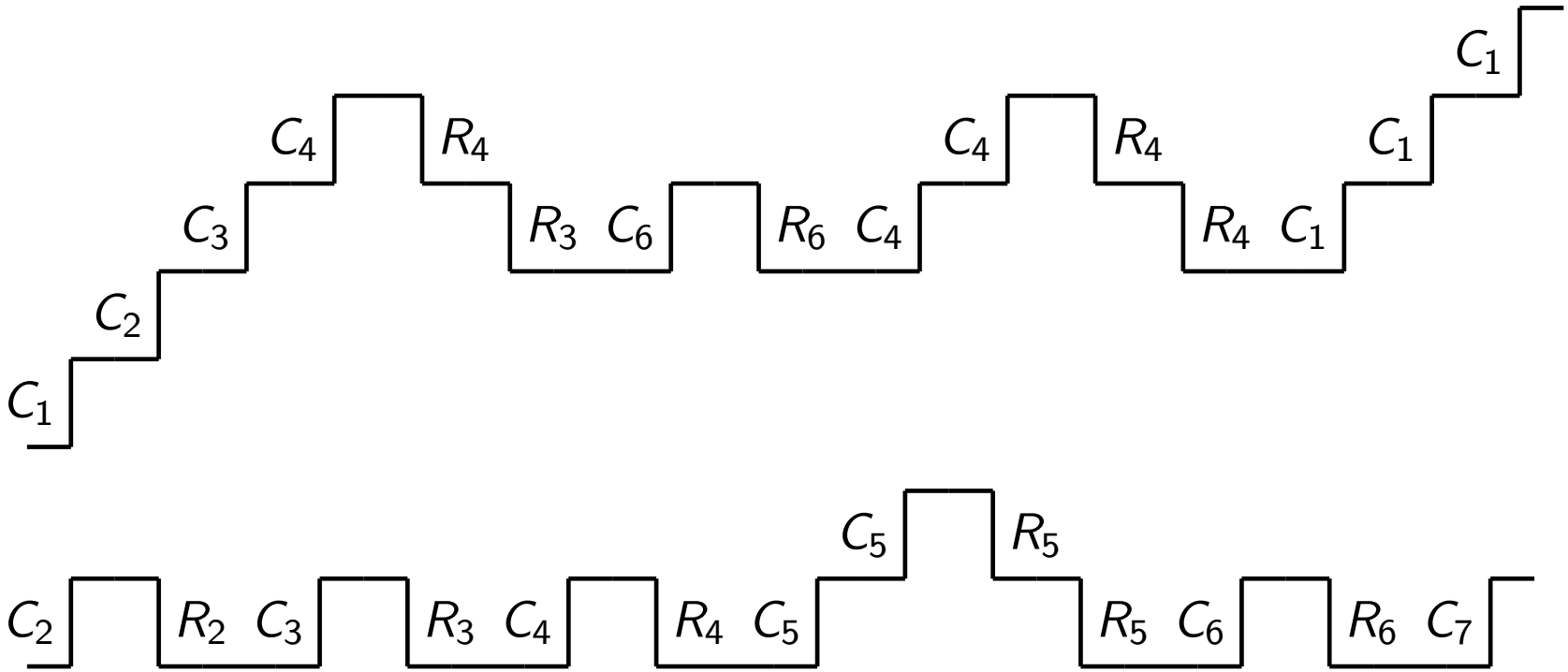
Context Sensitivity in Interprocedural Analysis



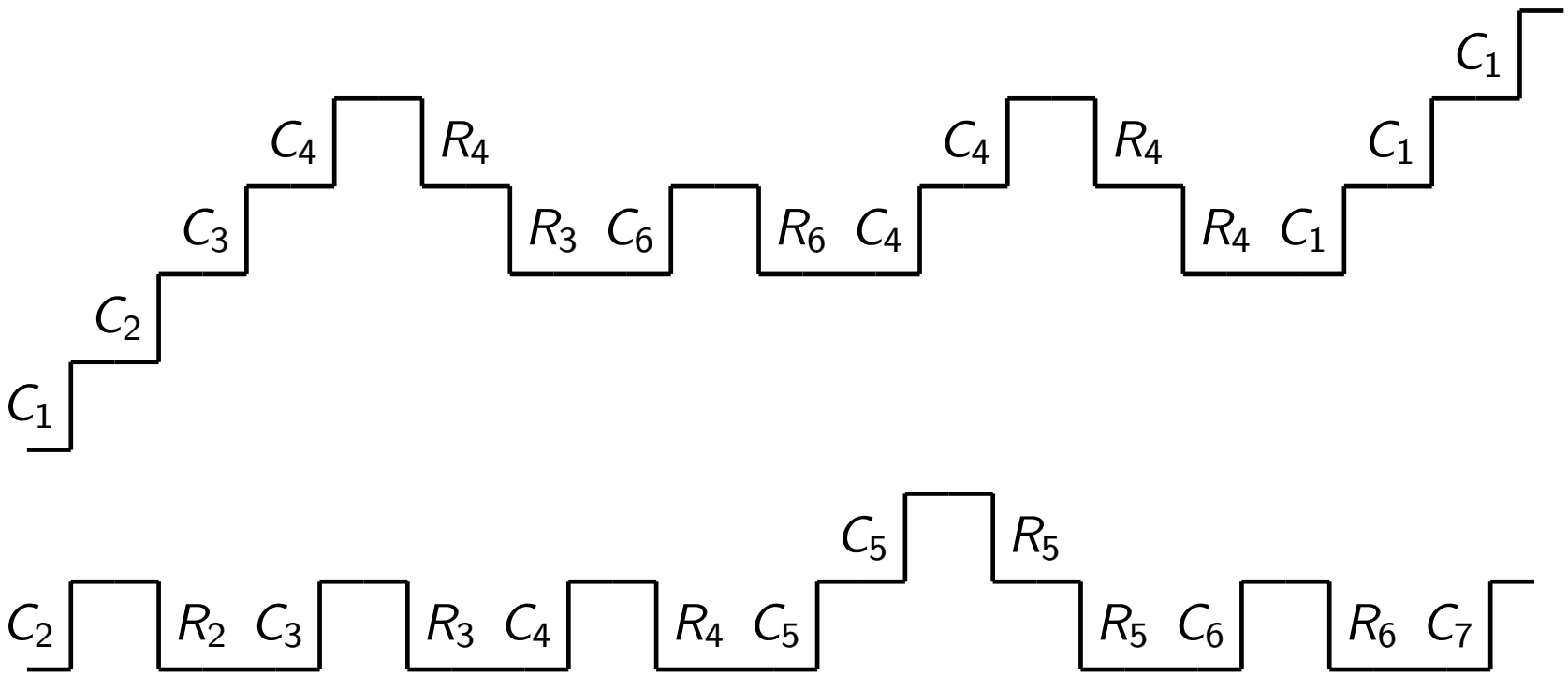
Staircase Diagrams of Interprocedurally Valid Paths



Staircase Diagrams of Interprocedurally Valid Paths

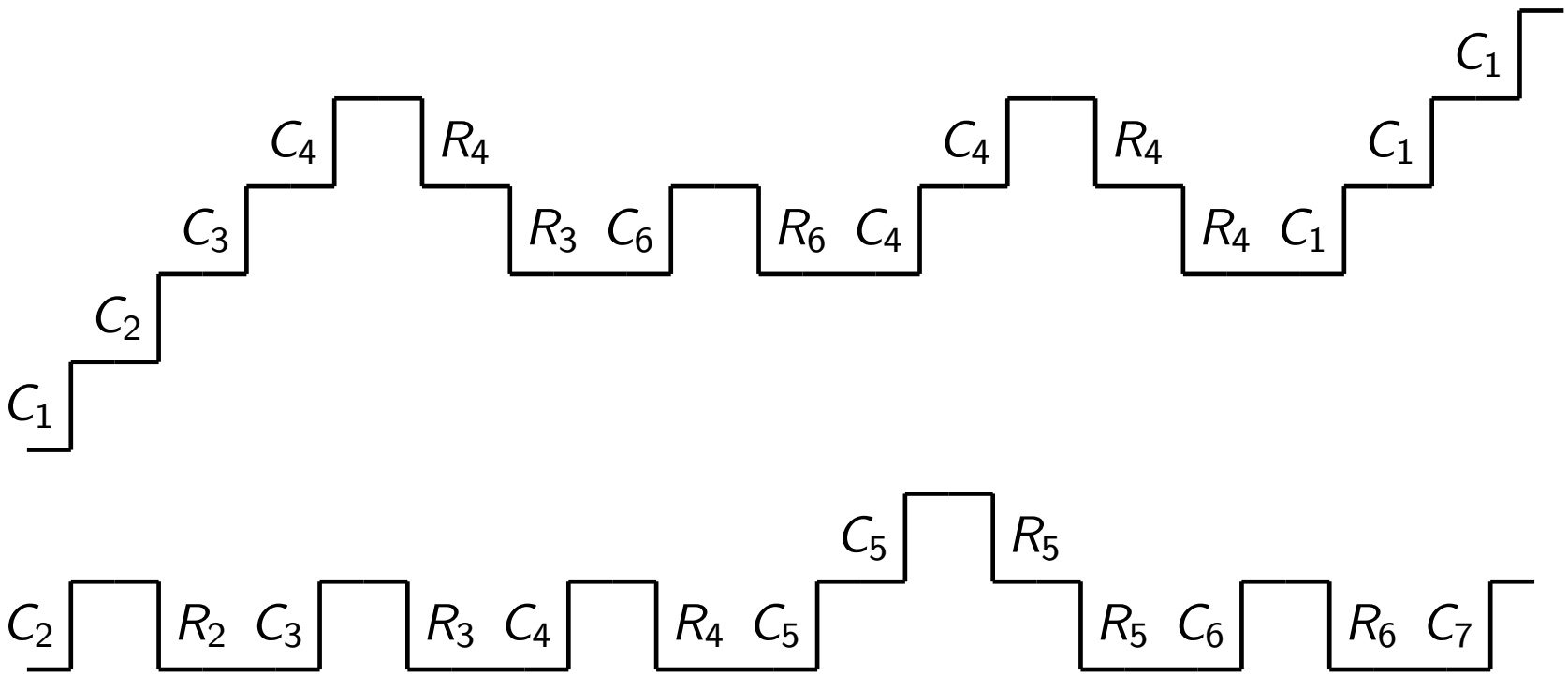


Staircase Diagrams of Interprocedurally Valid Paths



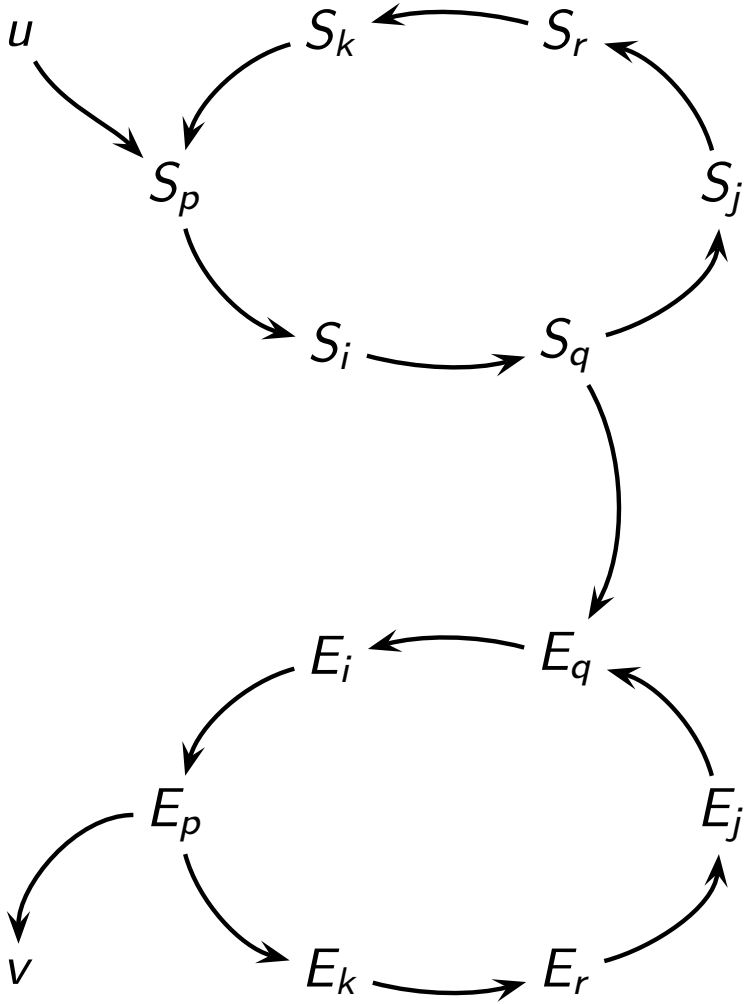
- "You can descend only as much as you have ascended!"

Staircase Diagrams of Interprocedurally Valid Paths

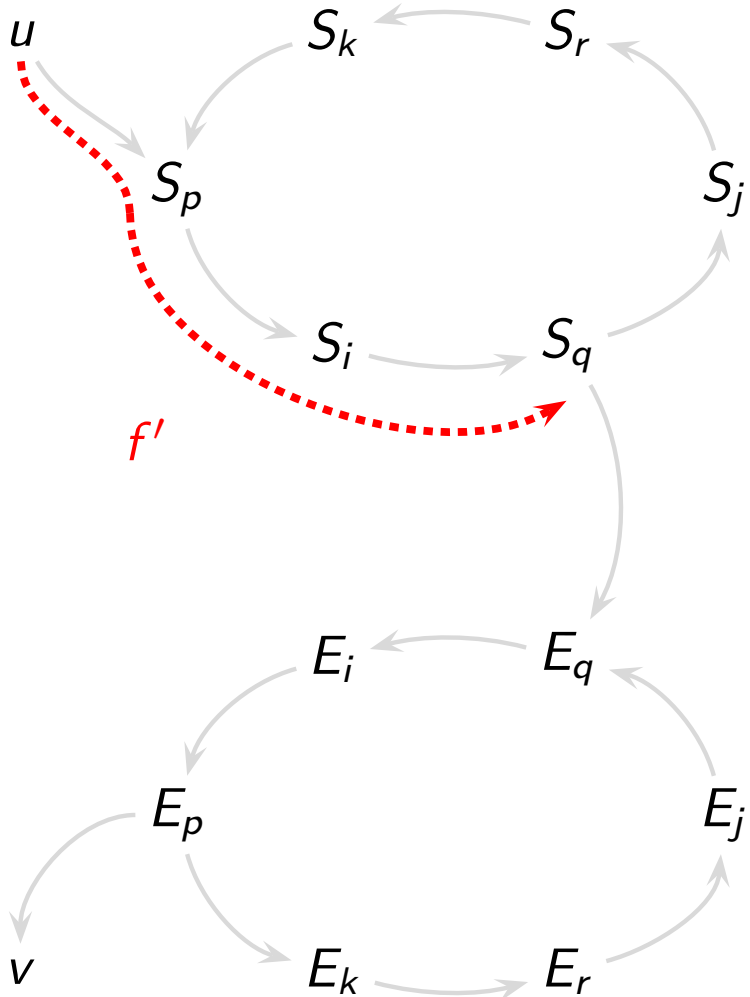


- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step

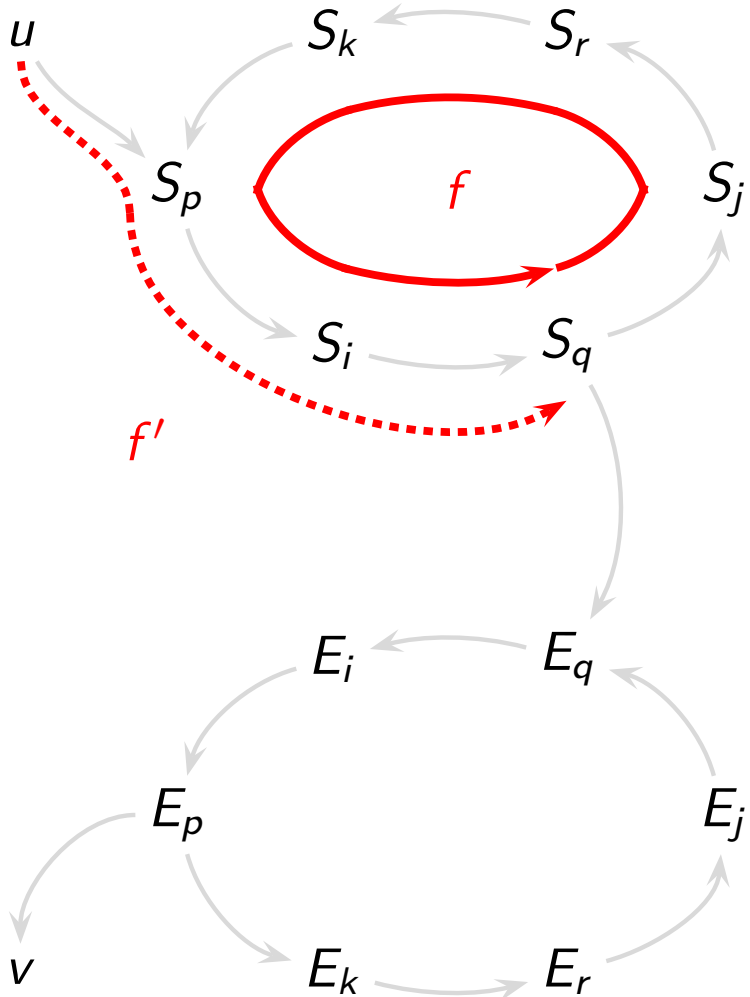
Context Sensitivity in Presence of Recursion



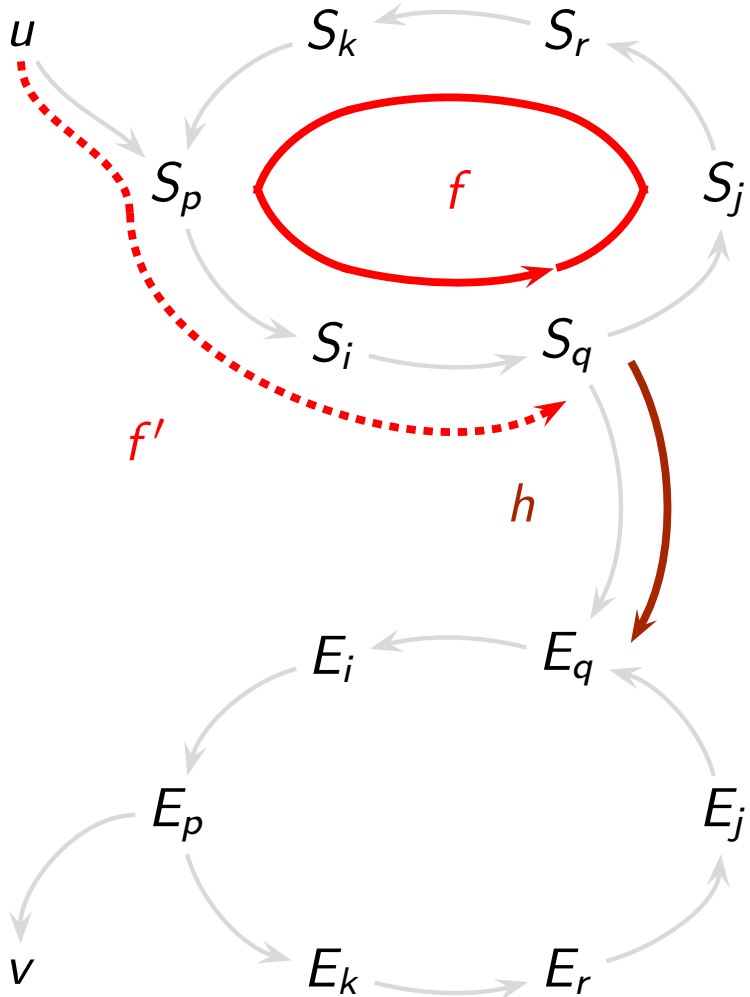
Context Sensitivity in Presence of Recursion



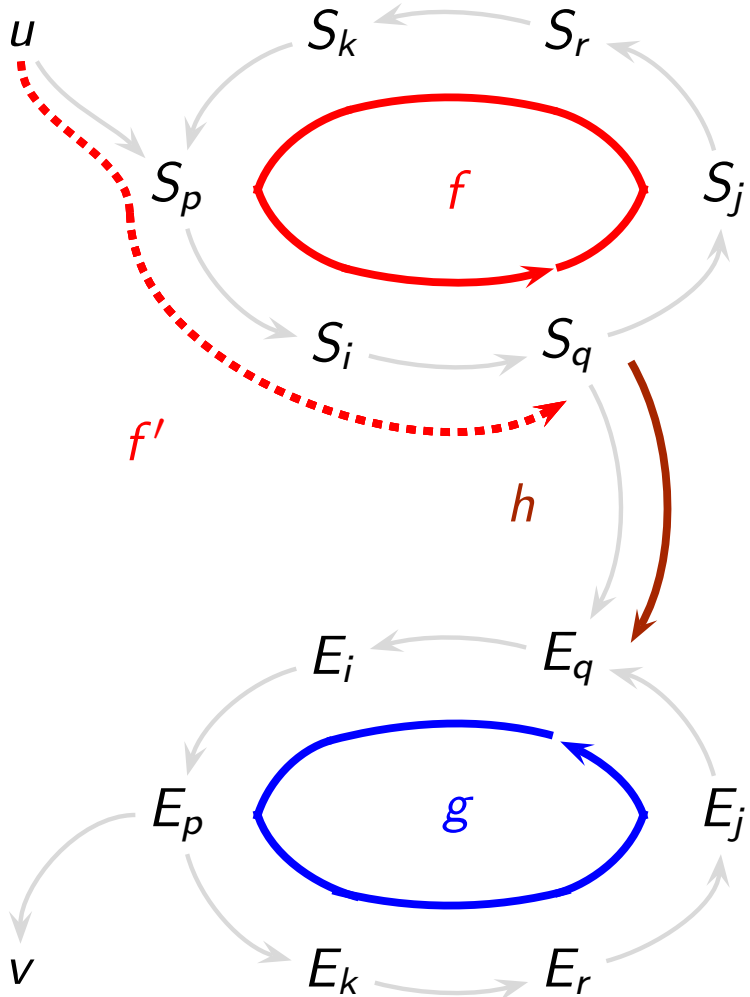
Context Sensitivity in Presence of Recursion



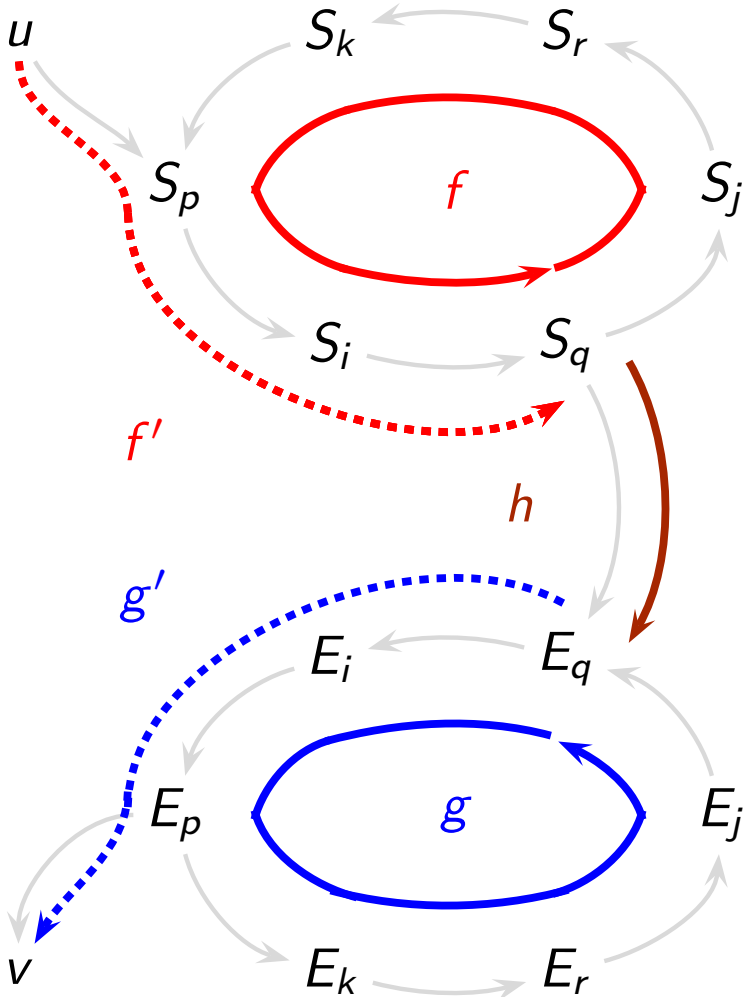
Context Sensitivity in Presence of Recursion



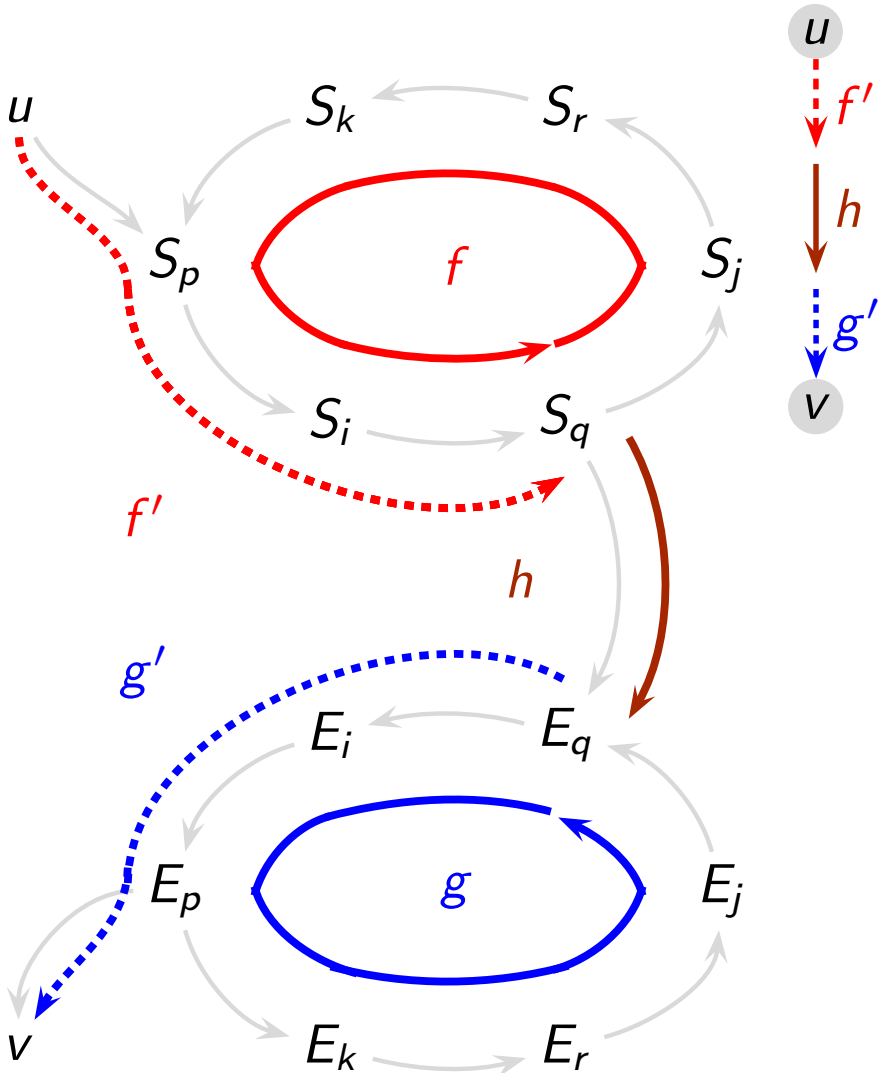
Context Sensitivity in Presence of Recursion



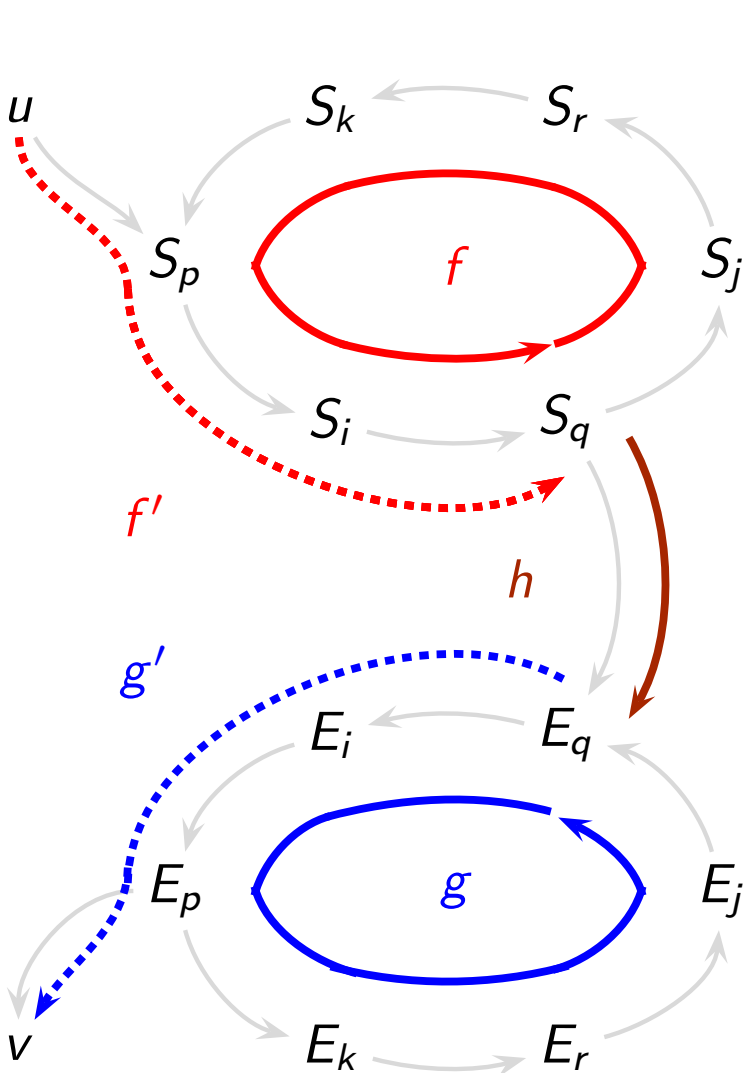
Context Sensitivity in Presence of Recursion

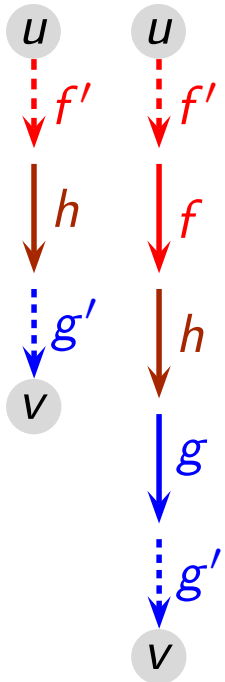


Context Sensitivity in Presence of Recursion

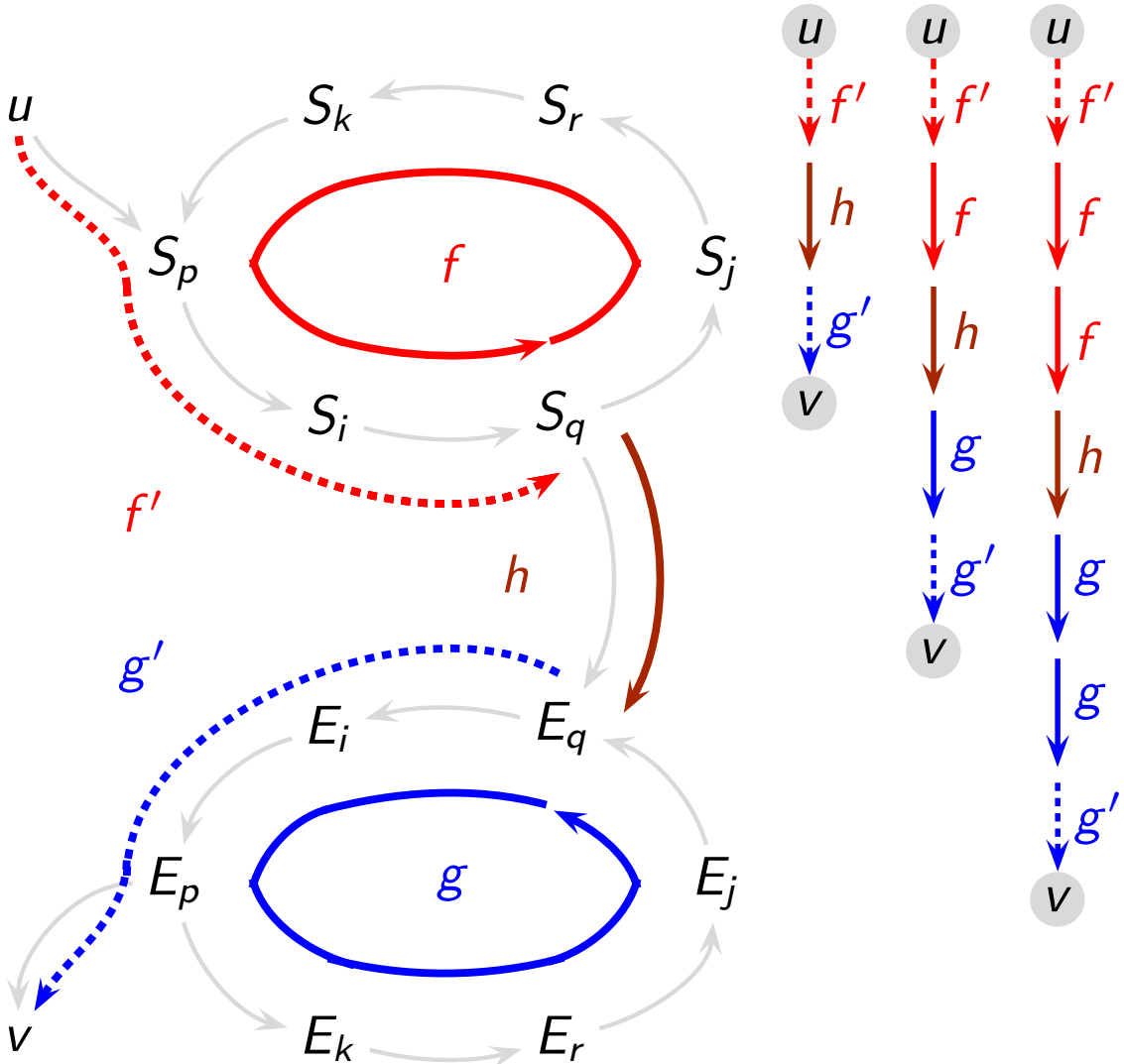


Context Sensitivity in Presence of Recursion

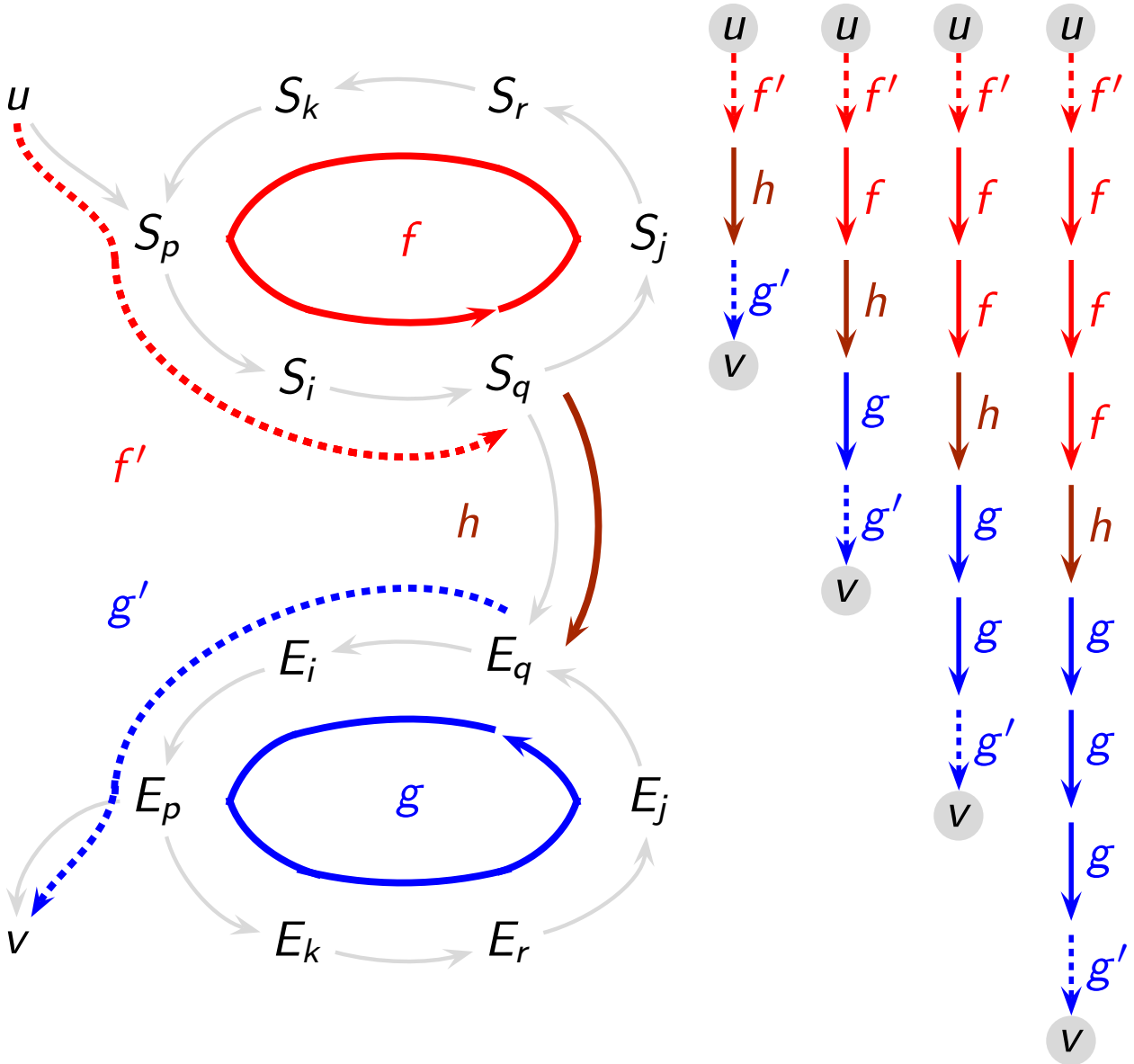




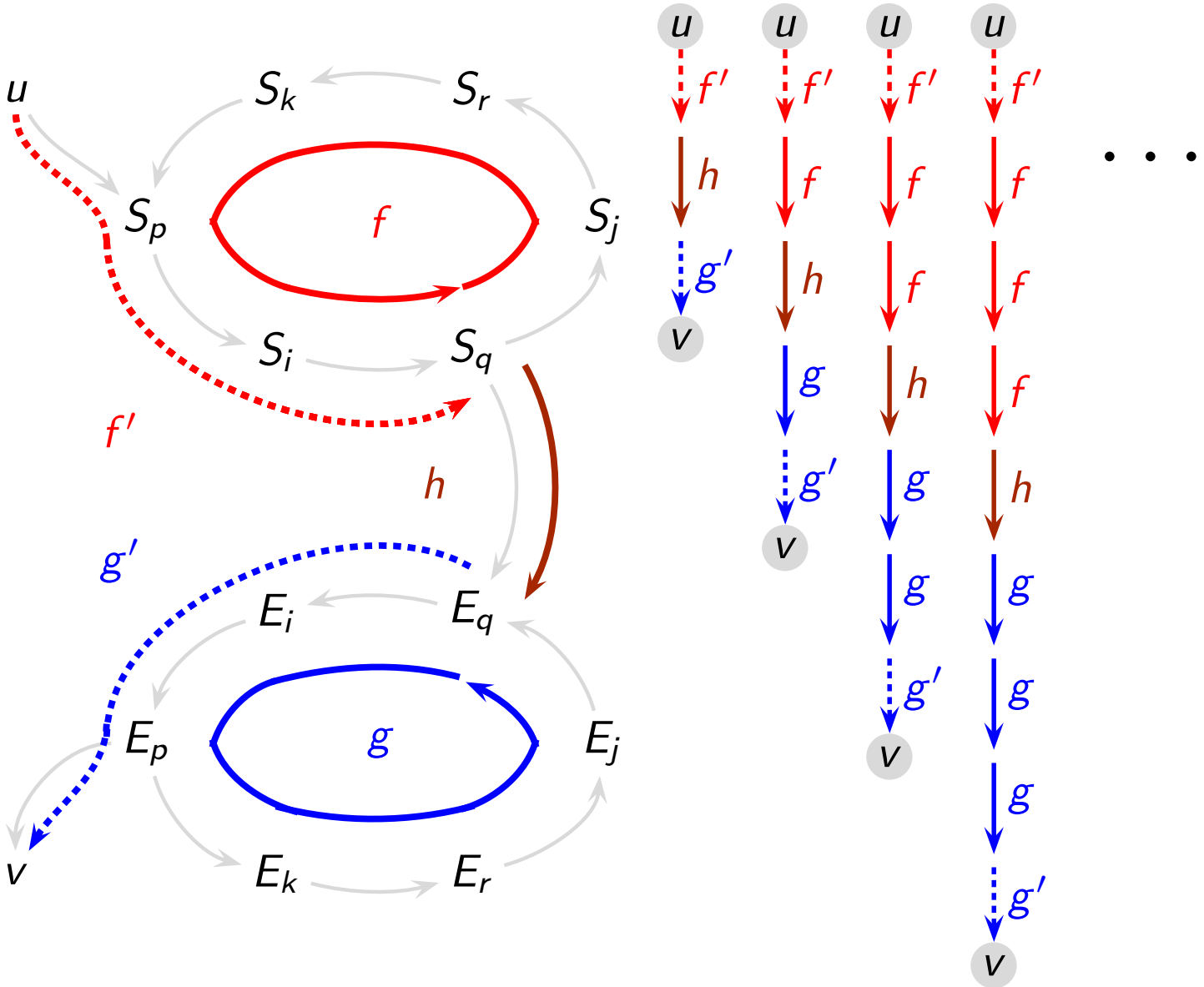
Context Sensitivity in Presence of Recursion



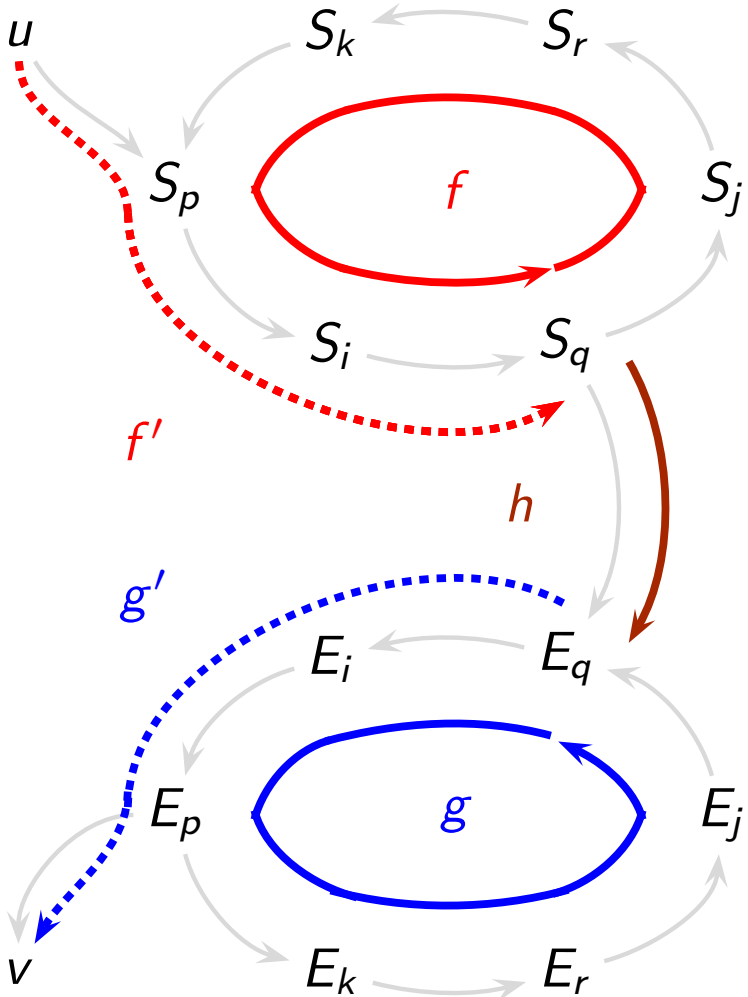
Context Sensitivity in Presence of Recursion

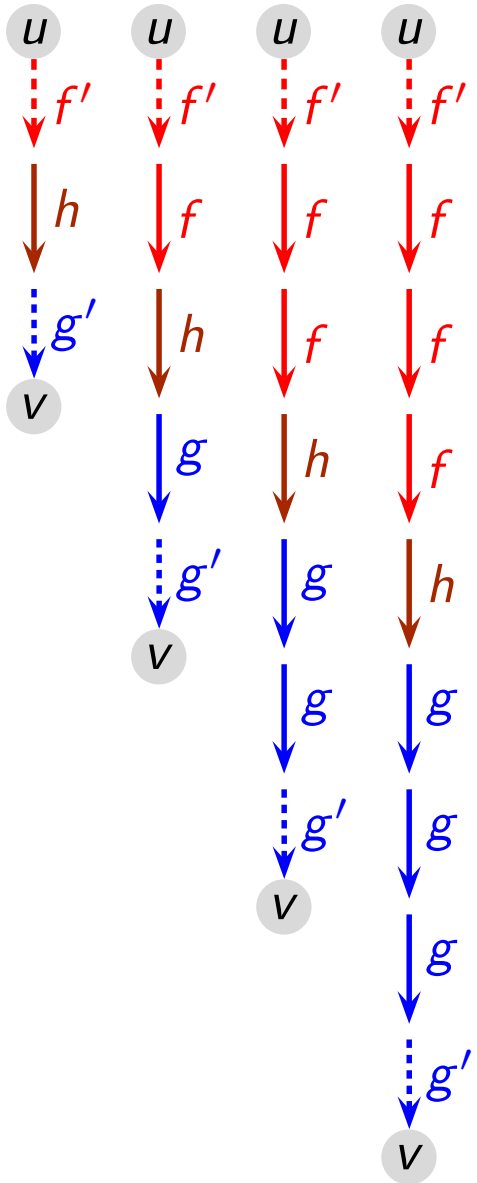


Context Sensitivity in Presence of Recursion



Context Sensitivity in Presence of Recursion

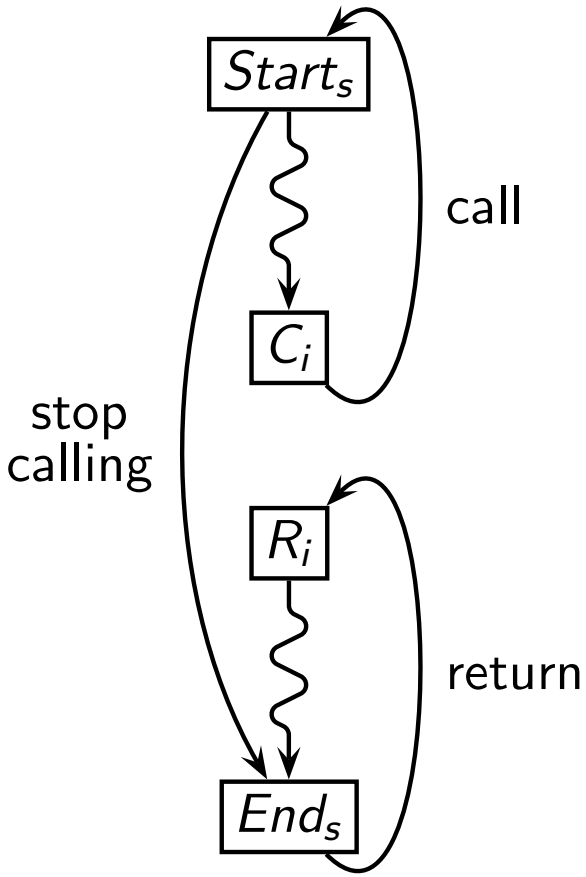




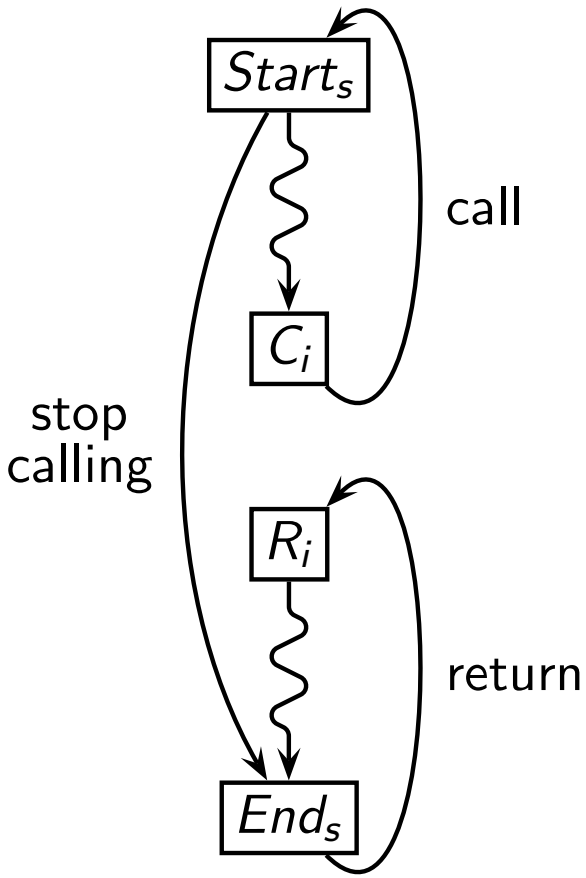
. . .

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

Context Sensitivity in the Presence of Recursion



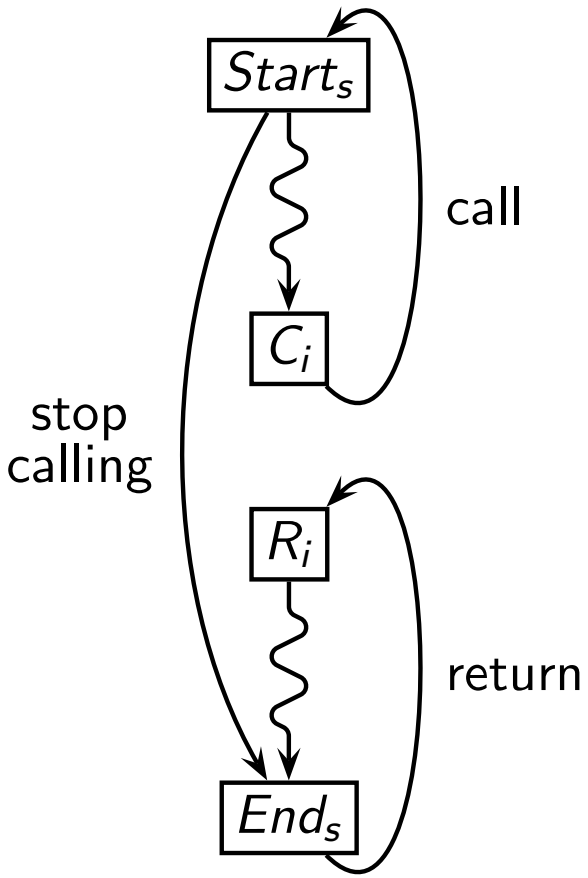
Context Sensitivity in the Presence of Recursion



- Paths from $Start_s$ to End_s should constitute a context free language

$$\text{call}^n \cdot \text{stop} \cdot \text{return}^n$$

Context Sensitivity in the Presence of Recursion



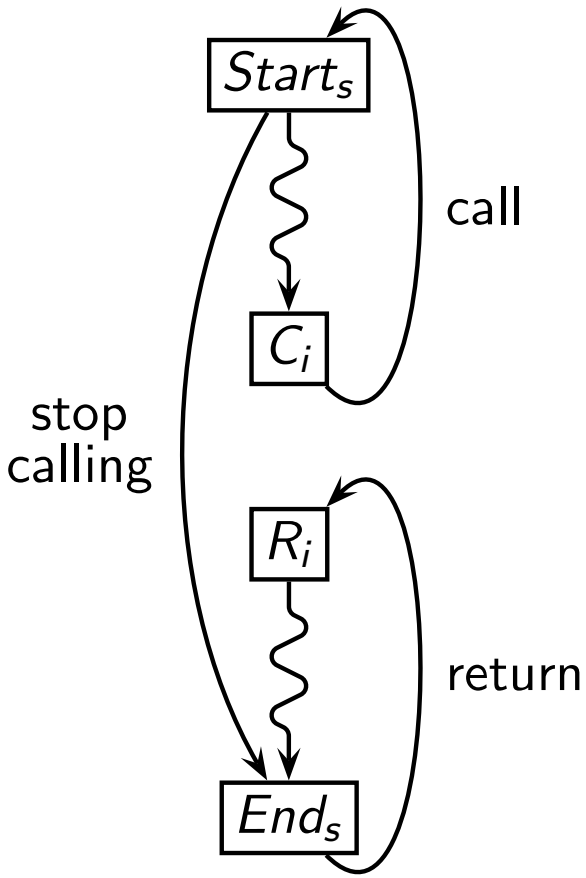
- Paths from $Start_s$ to End_s should constitute a context free language

$$\text{call}^n \cdot \text{stop} \cdot \text{return}^n$$

- If we treat cycle of recursion as an SCC
  - Calls and returns become jumps, and
  - paths are approximated by a regular language

$$\text{call}^* \cdot \text{stop} \cdot \text{return}^*$$

Context Sensitivity in the Presence of Recursion



- Paths from $Start_s$ to End_s should constitute a context free language

$$\text{call}^n \cdot \text{stop} \cdot \text{return}^n$$

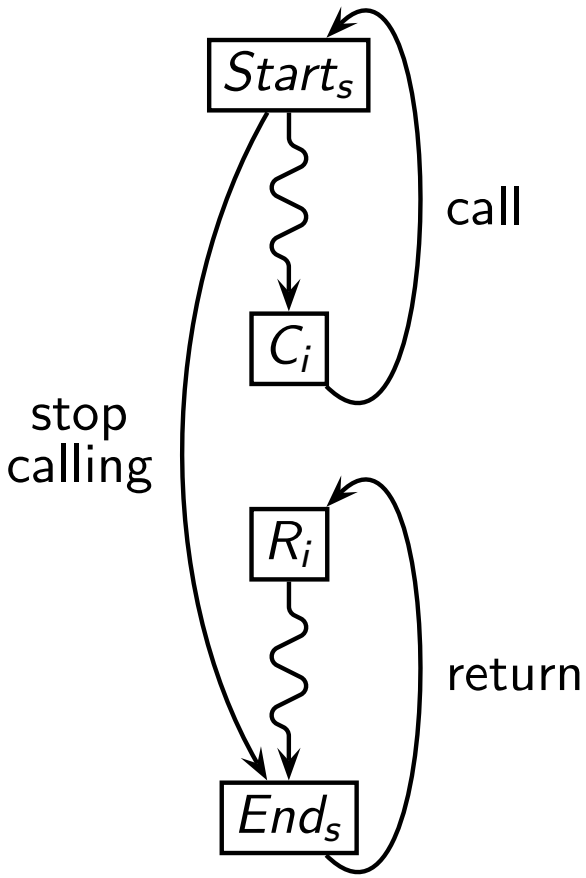
- If we treat cycle of recursion as an SCC
  - Calls and returns become jumps, and
  - paths are approximated by a regular language

$$\text{call}^* \cdot \text{stop} \cdot \text{return}^*$$

Most context sensitive approaches

- either do not consider recursion, or
- do not consider recursive pointer manipulation ("`p=p->n`"), or
- are context insensitive in recursion

Context Sensitivity in the Presence of Recursion



- Paths from $Start_s$ to End_s should constitute a context free language

$$\text{call}^n \cdot \text{stop} \cdot \text{return}^n$$

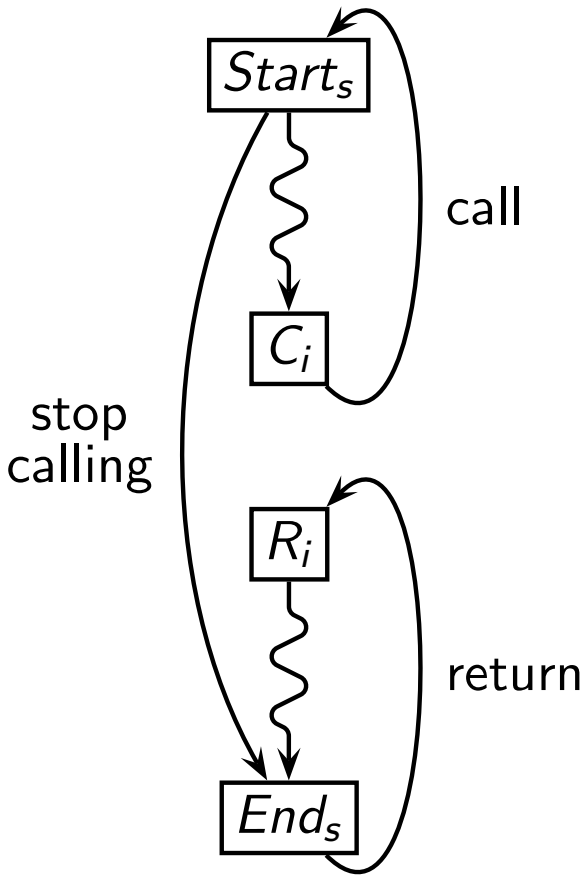
- If we treat cycle of recursion as an SCC
  - Calls and returns become jumps, and
  - paths are approximated by a regular language

$$\text{call}^* \cdot \text{stop} \cdot \text{return}^*$$

Most context sensitive approaches

- either do not consider recursion, or
- do not consider recursive pointer manipulation ("`p=p->n`"), or
- are context insensitive in recursion

Context Sensitivity in the Presence of Recursion



- Paths from $Start_s$ to End_s should constitute a context free language

$$\text{call}^n \cdot \text{stop} \cdot \text{return}^n$$

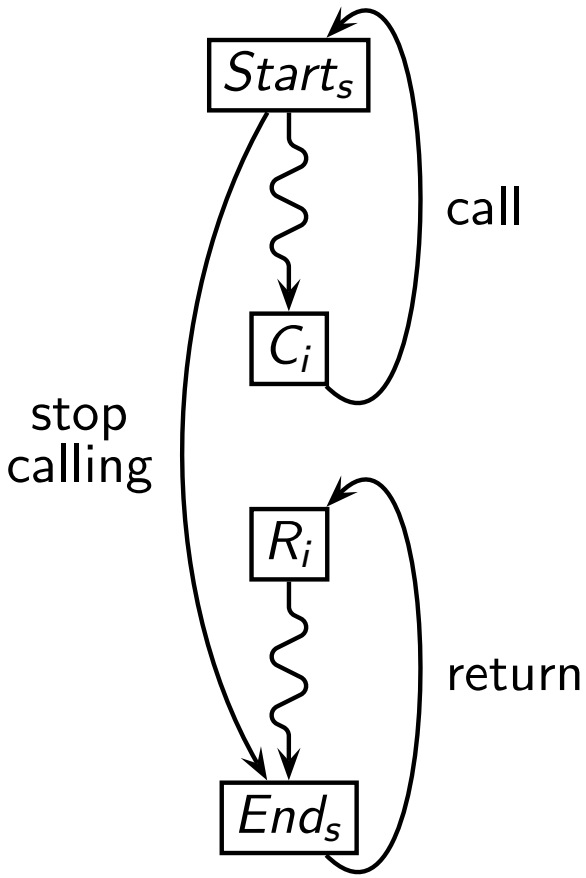
- If we treat cycle of recursion as an SCC
  - Calls and returns become jumps, and
  - paths are approximated by a regular language

$$\text{call}^* \cdot \text{stop} \cdot \text{return}^*$$

Most context sensitive approaches

- either do not consider recursion, or
- do not consider recursive pointer manipulation (“`p=p->n`”), or
- are context insensitive in recursion

Context Sensitivity in the Presence of Recursion



- Paths from $Start_s$ to End_s should constitute a context free language

$$\text{call}^n \cdot \text{stop} \cdot \text{return}^n$$

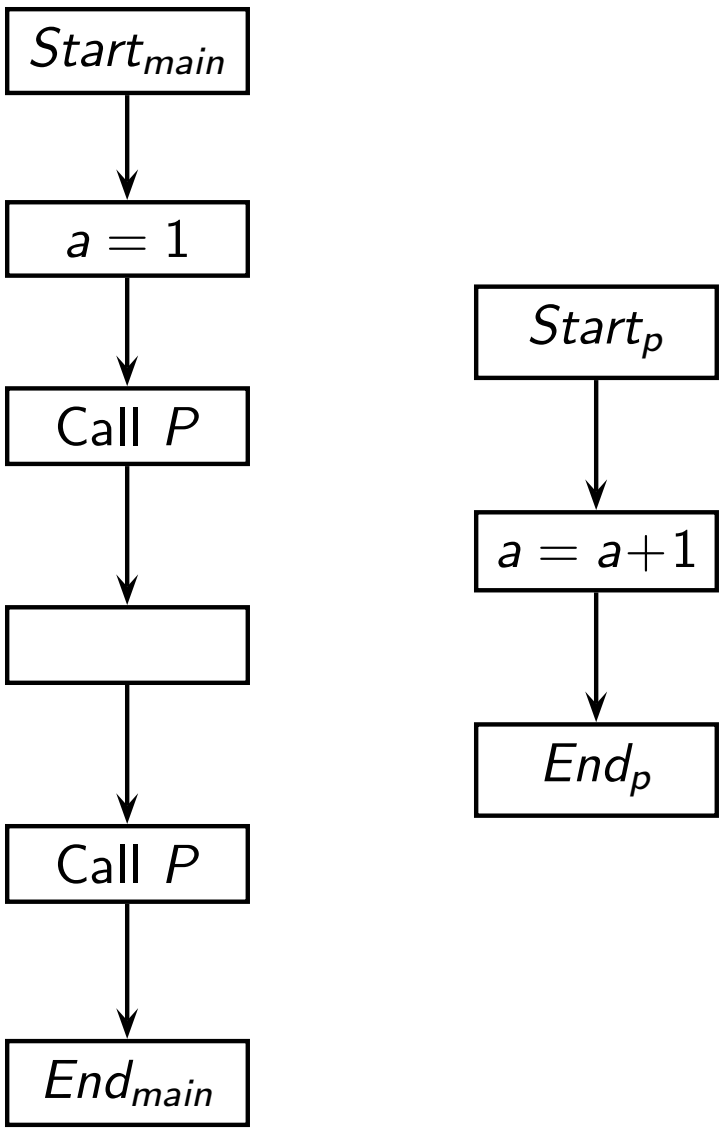
- If we treat cycle of recursion as an SCC
  - Calls and returns become jumps, and
  - paths are approximated by a regular language

$$\text{call}^* \cdot \text{stop} \cdot \text{return}^*$$

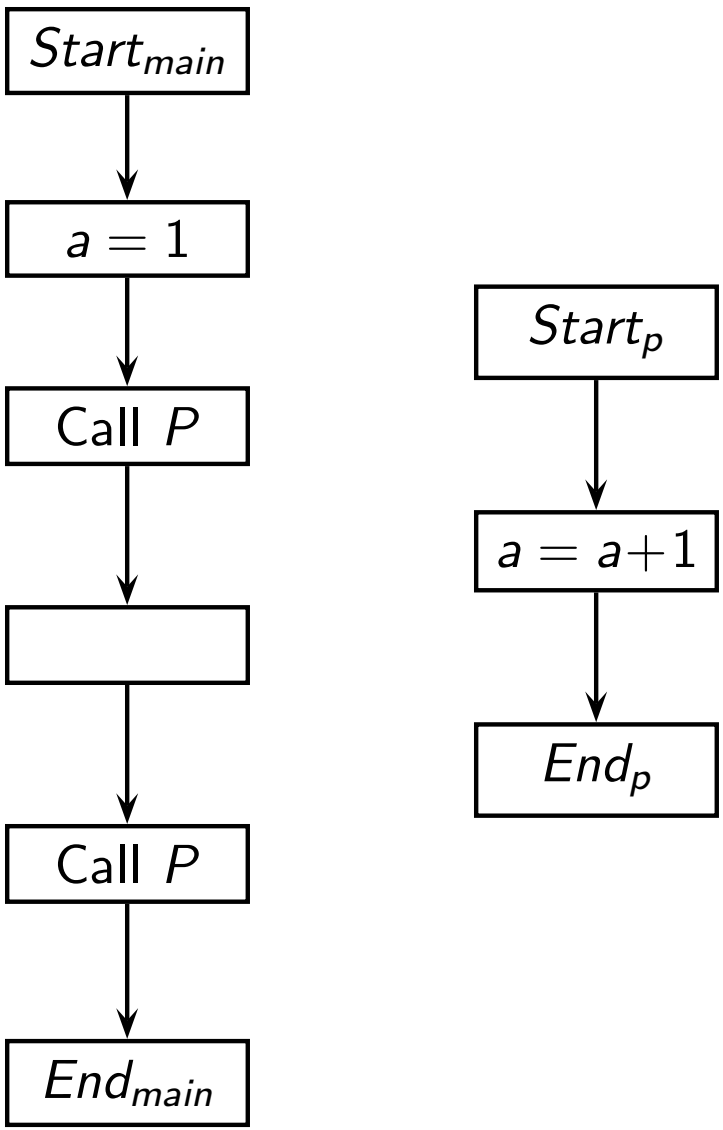
Most context sensitive approaches

- either do not consider recursion, or
- do not consider recursive pointer manipulation ("`p=p->n`"), or
- are context insensitive in recursion

Context Insensitivity = Imprecision + Potential Inefficiency

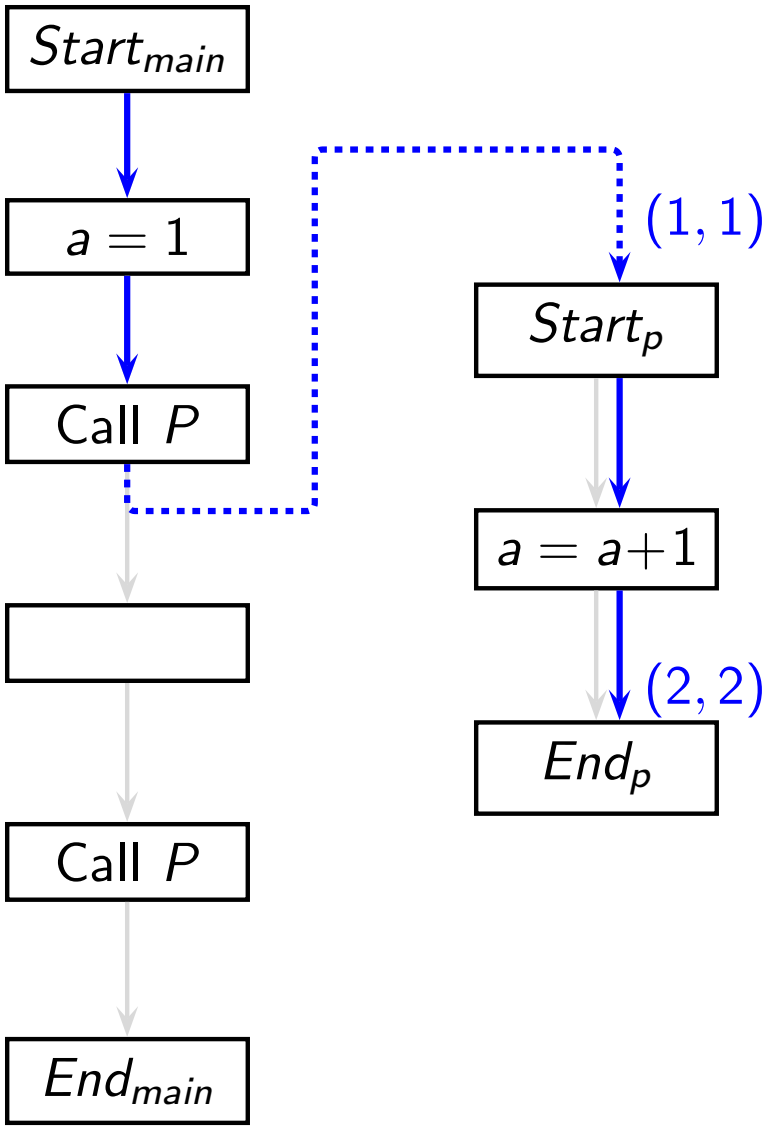


Context Insensitivity = Imprecision + Potential Inefficiency



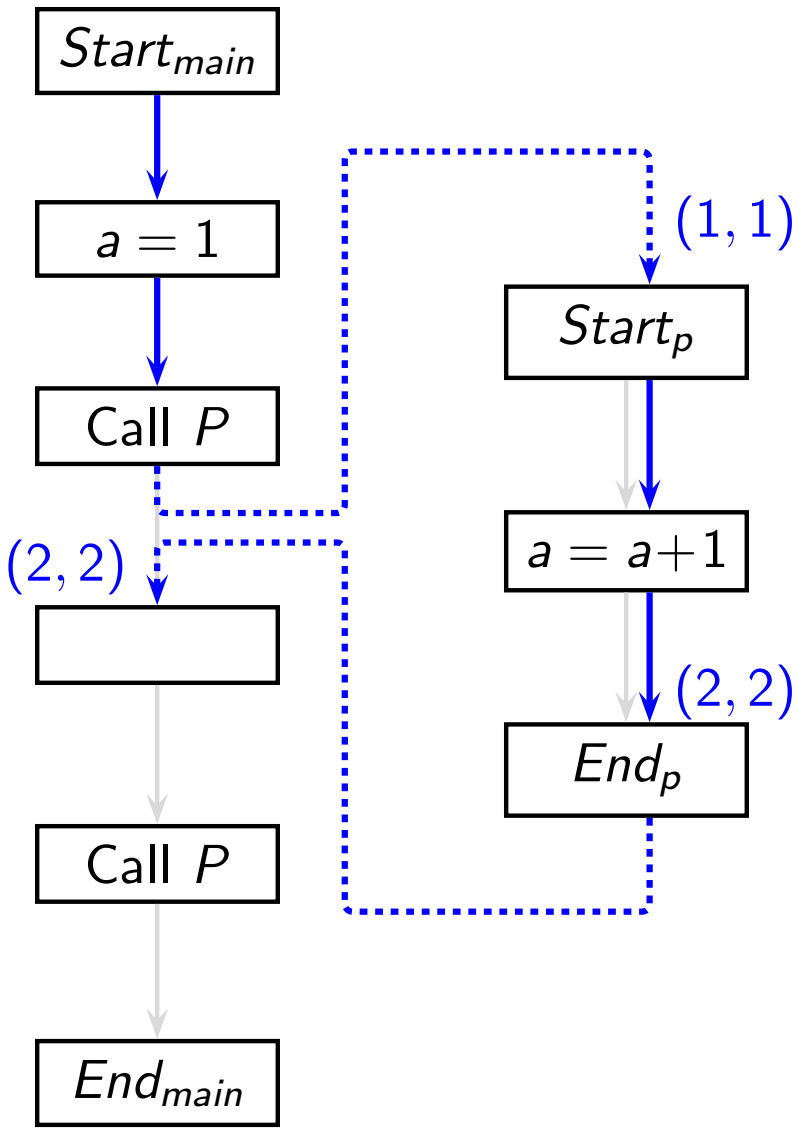
- What is the value range of a?

Context Insensitivity = Imprecision + Potential Inefficiency



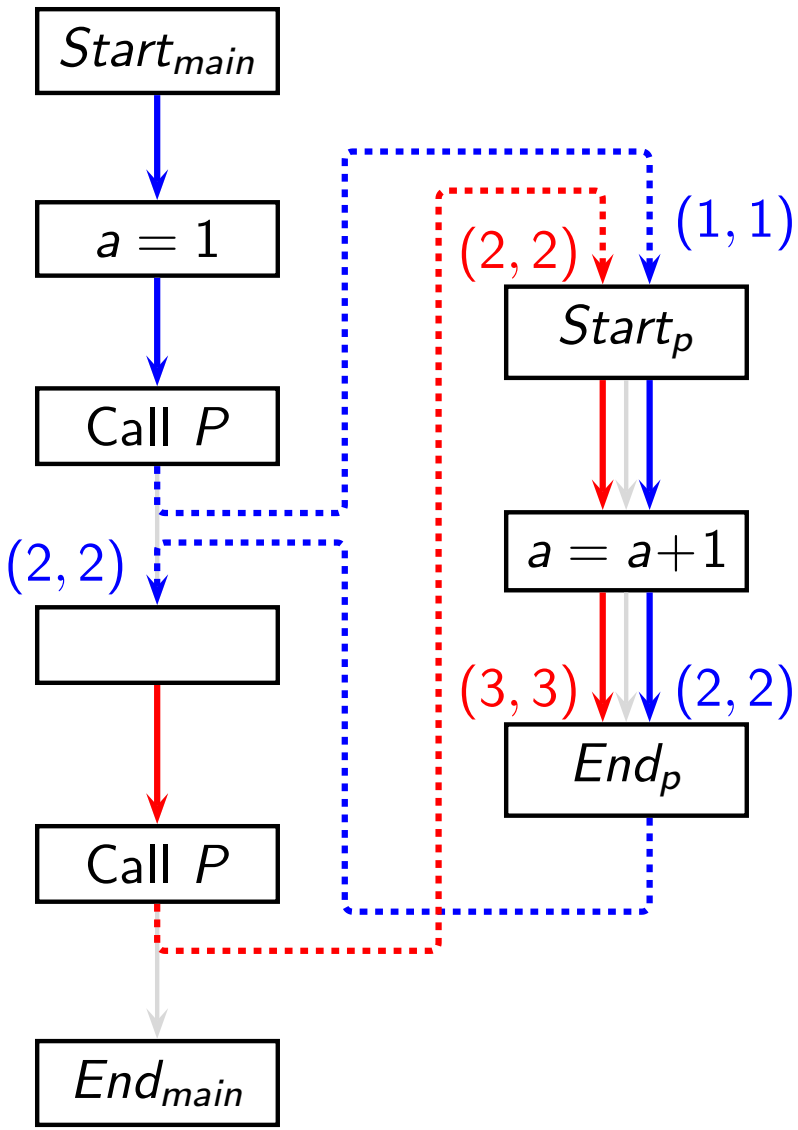
- What is the value range of a?

Context Insensitivity = Imprecision + Potential Inefficiency



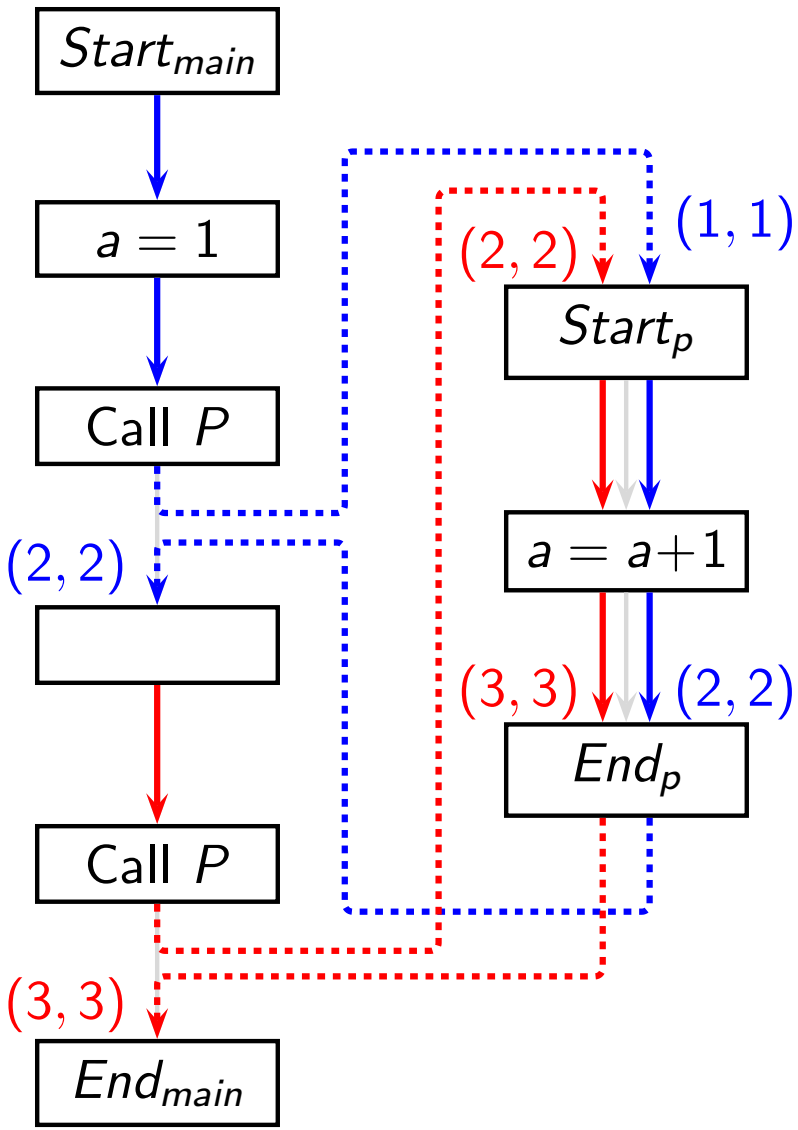
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P

Context Insensitivity = Imprecision + Potential Inefficiency



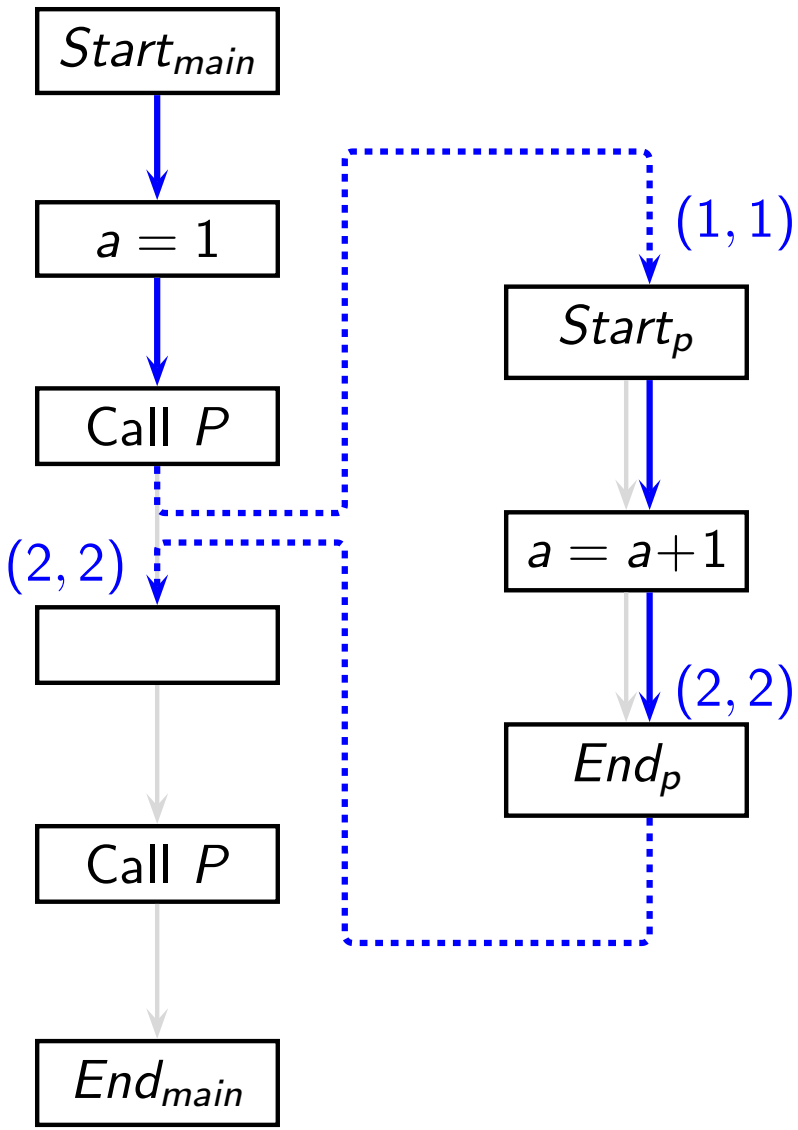
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P

Context Insensitivity = Imprecision + Potential Inefficiency



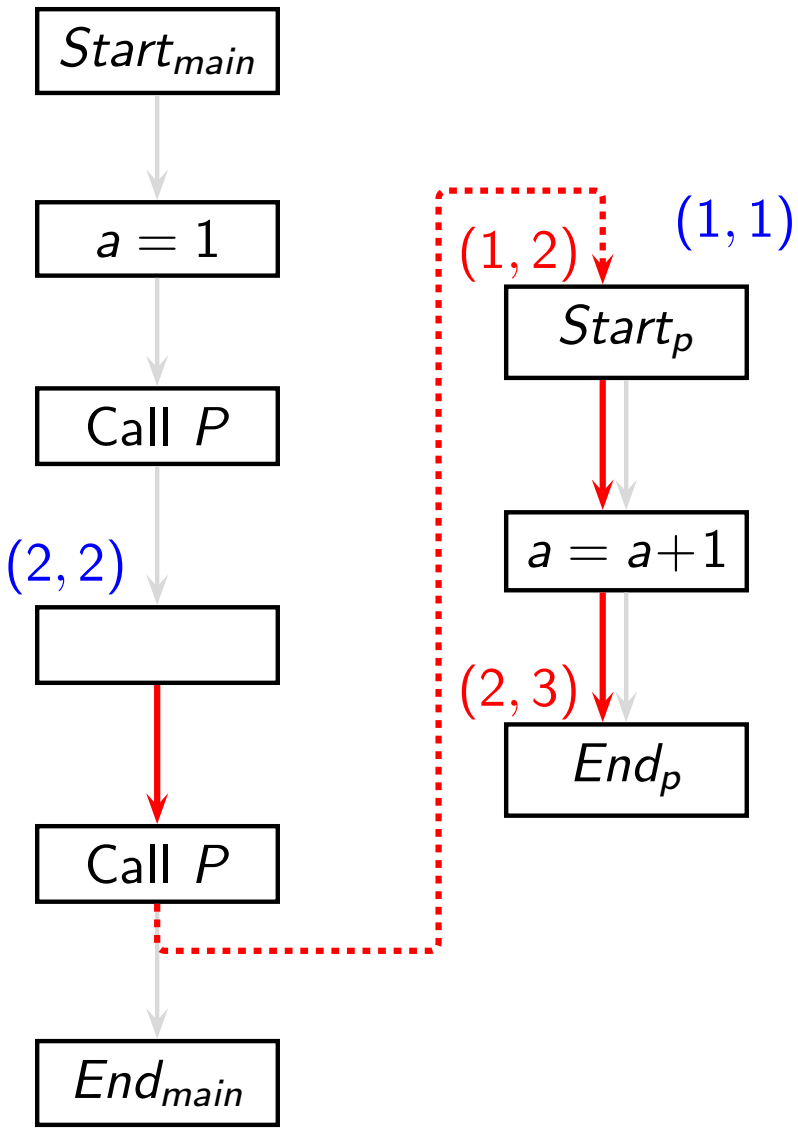
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3, 3)$

Context Insensitivity = Imprecision + Potential Inefficiency



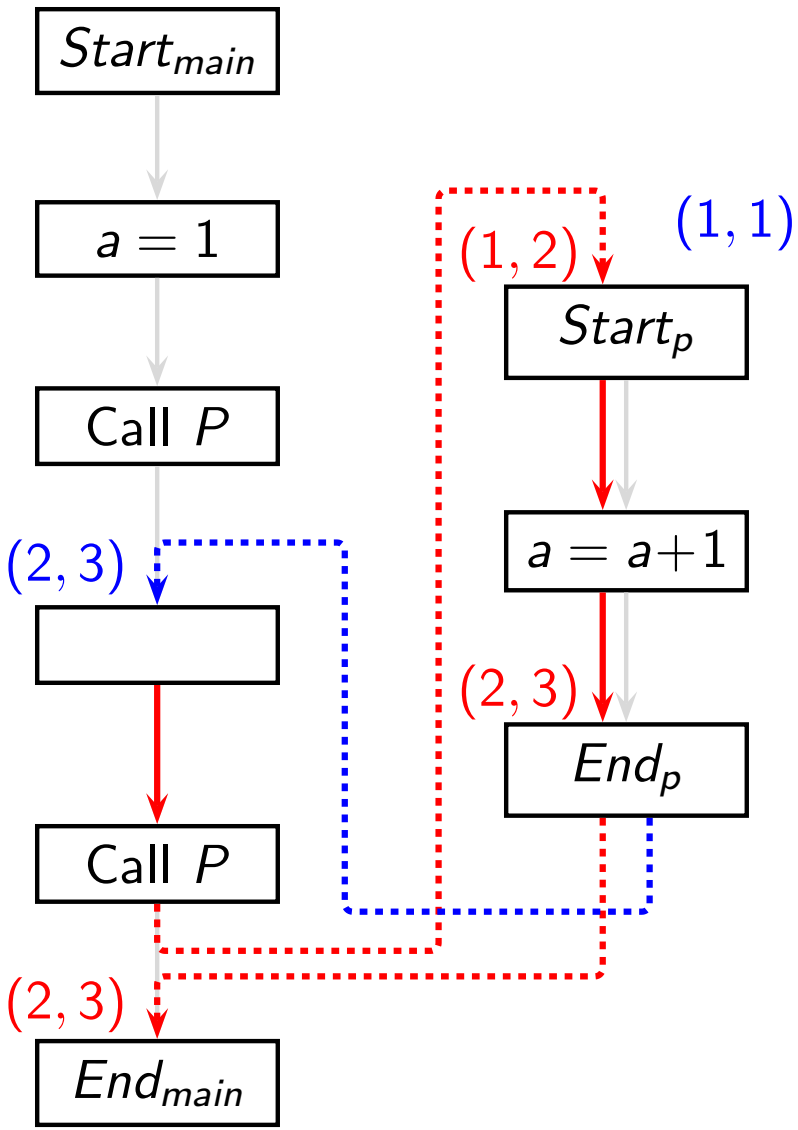
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3,3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller

Context Insensitivity = Imprecision + Potential Inefficiency



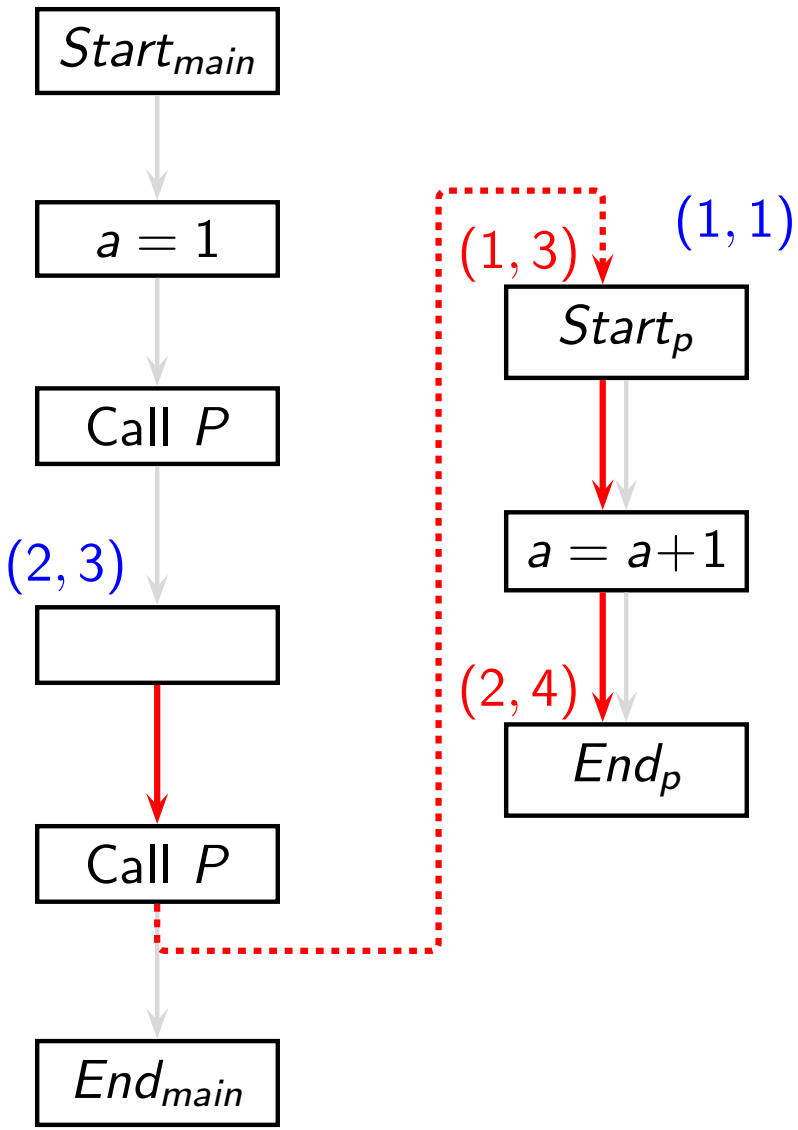
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3,3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller

Context Insensitivity = Imprecision + Potential Inefficiency



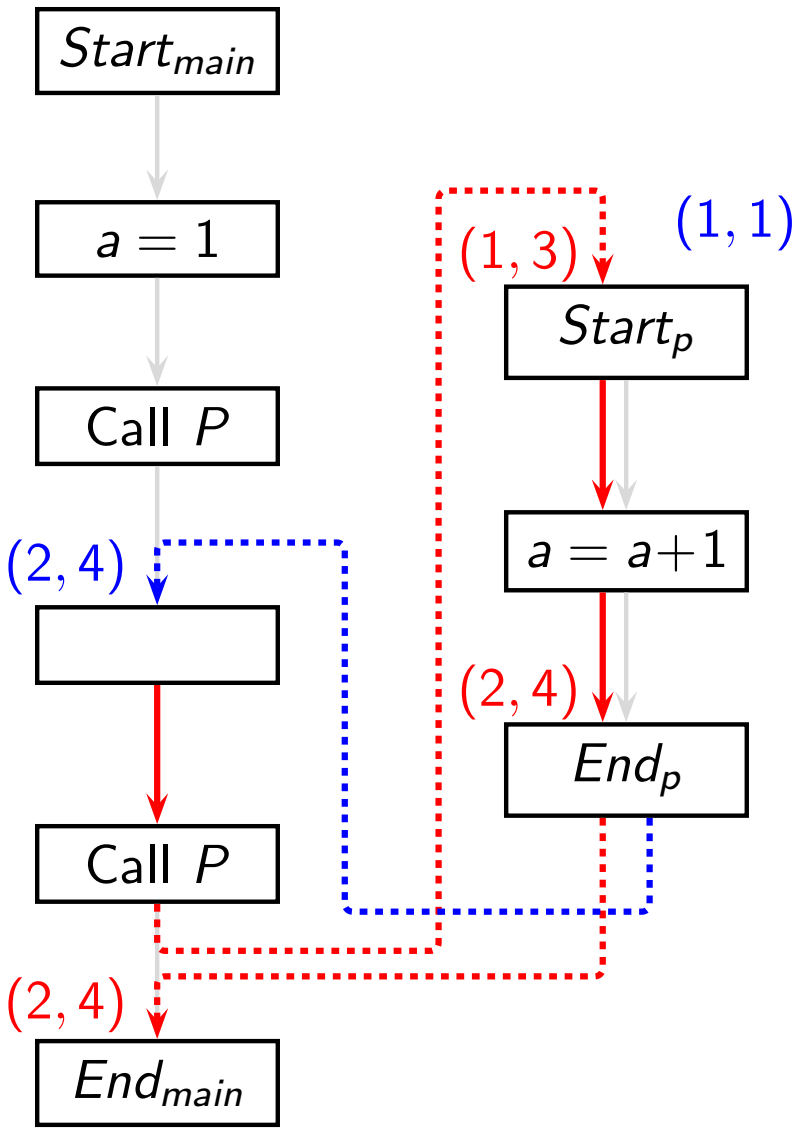
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3,3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller

Context Insensitivity = Imprecision + Potential Inefficiency



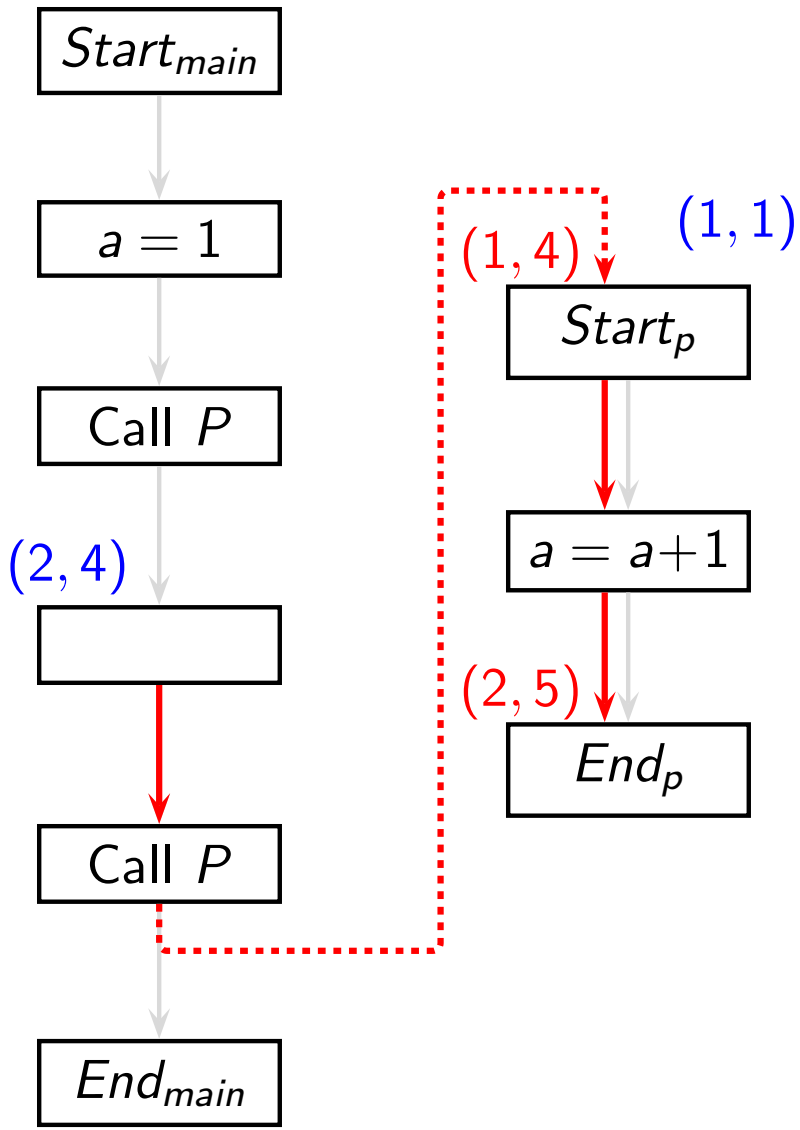
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3,3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller

Context Insensitivity = Imprecision + Potential Inefficiency



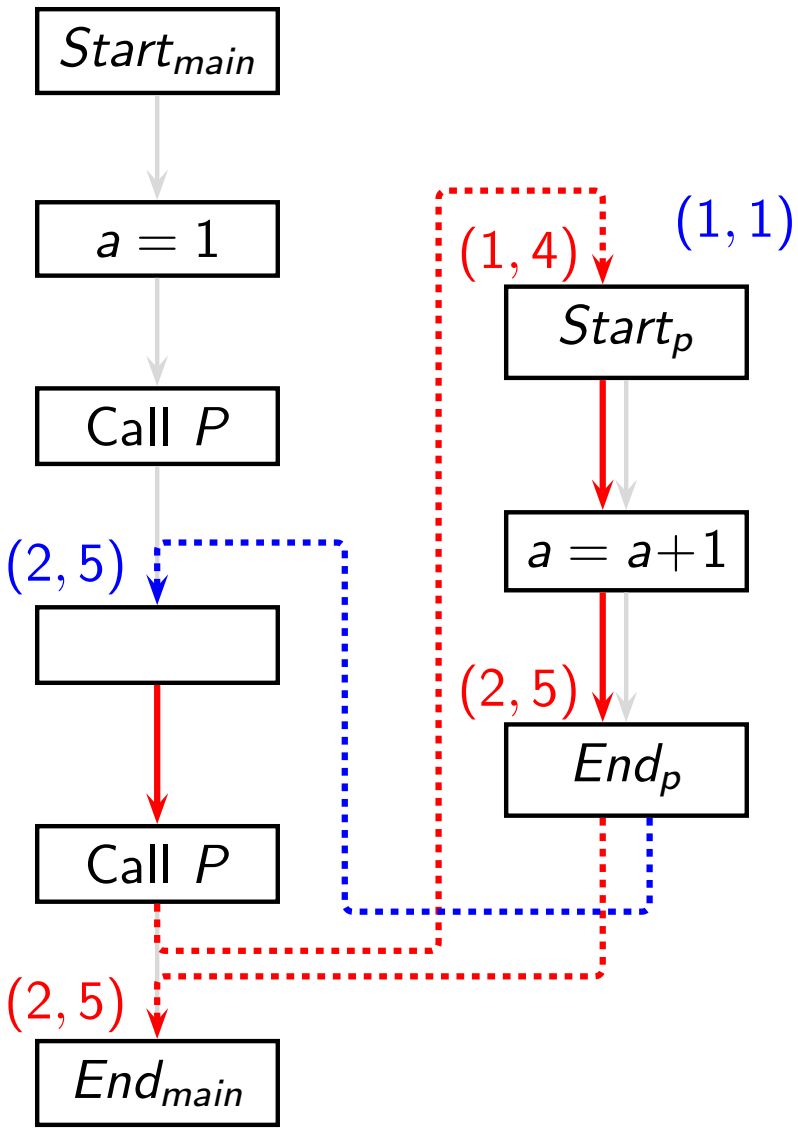
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3,3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller

Context Insensitivity = Imprecision + Potential Inefficiency



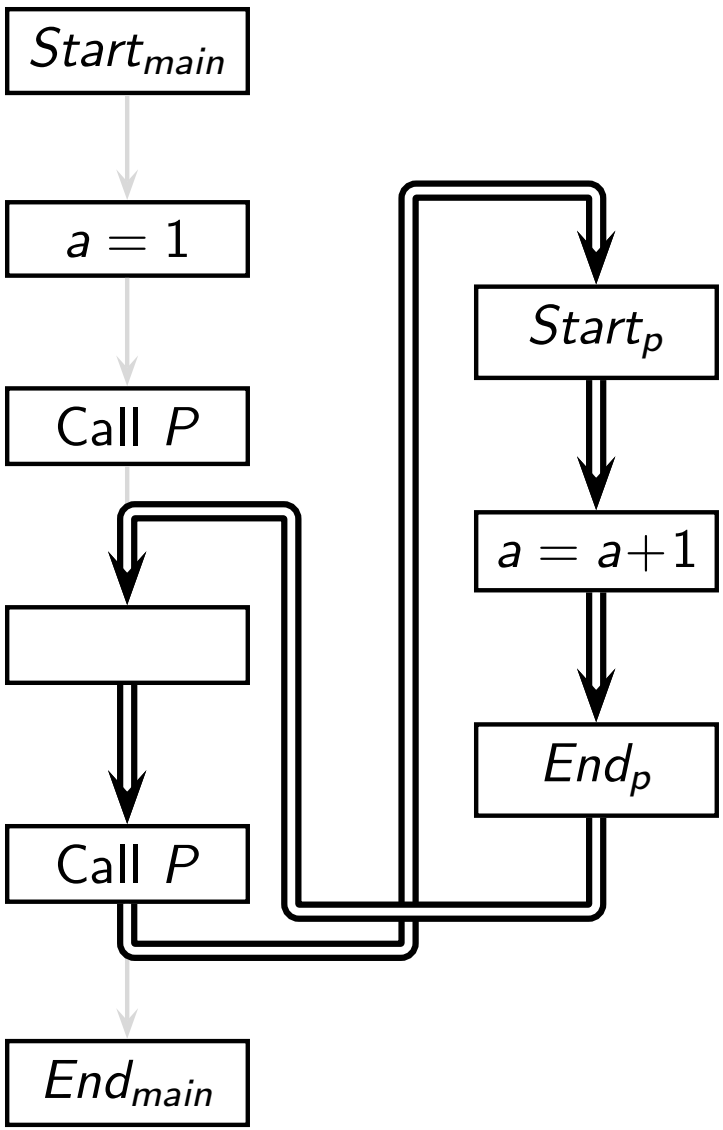
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3,3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller

Context Insensitivity = Imprecision + Potential Inefficiency



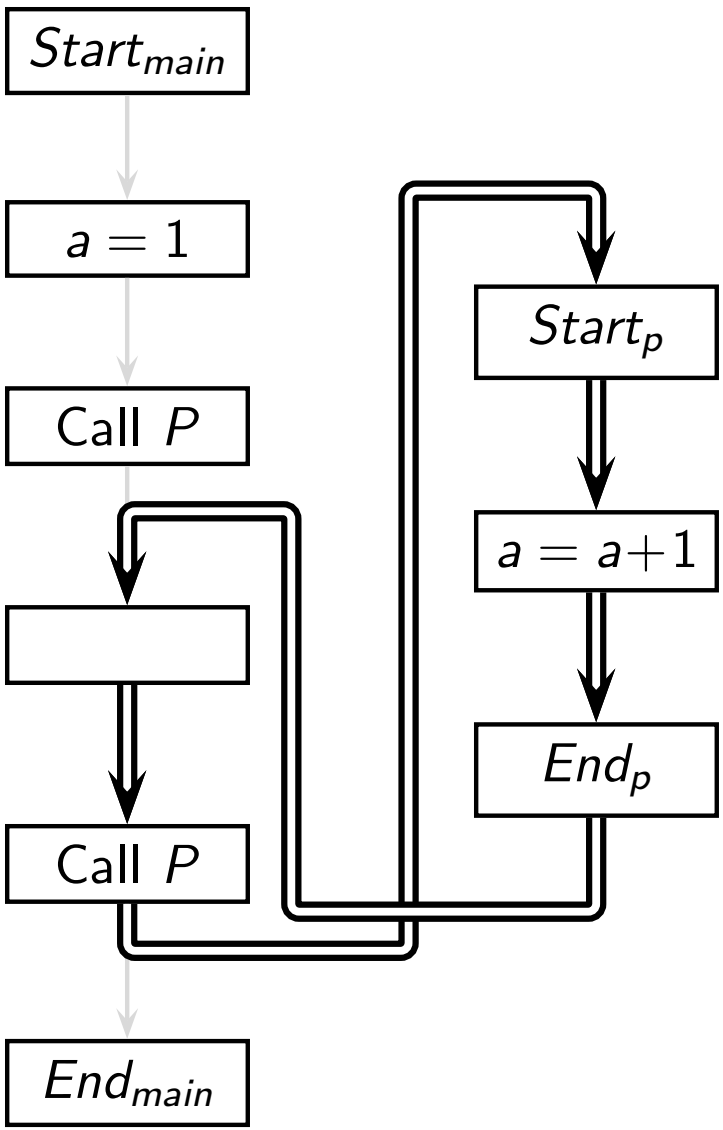
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3,3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller
  - Range of a at End_{main} is $(2,\ldots)$

Context Insensitivity = Imprecision + Potential Inefficiency



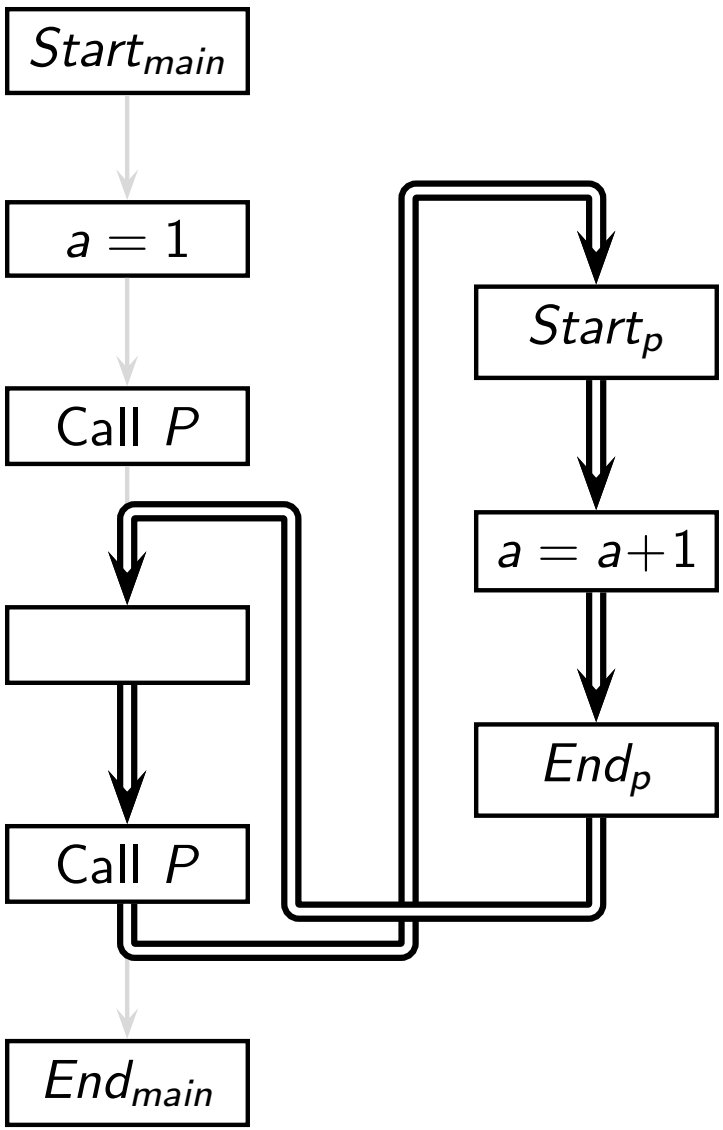
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3, 3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller
  - Range of a at End_{main} is $(2, \ldots)$

Context Insensitivity = Imprecision + Potential Inefficiency



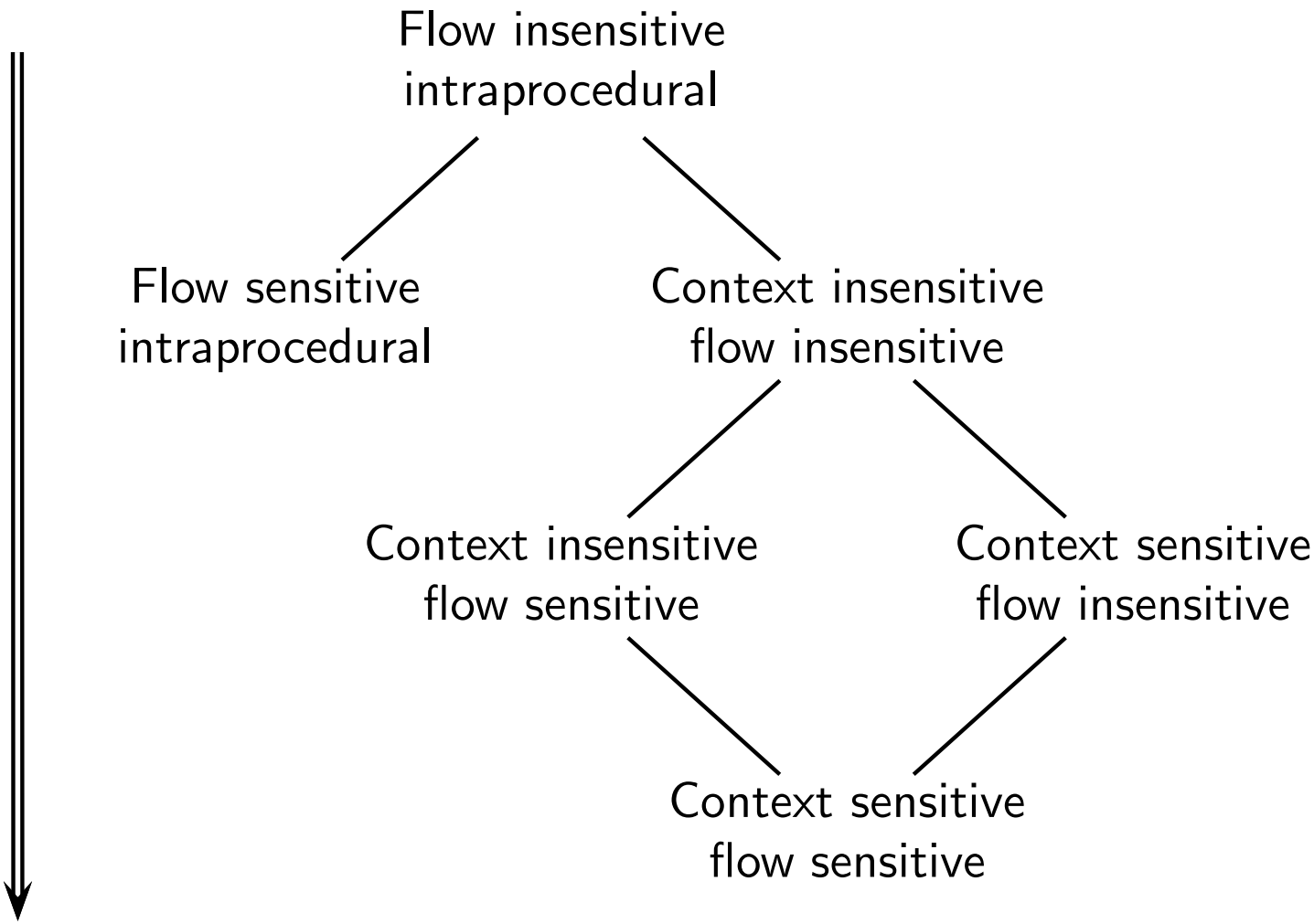
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3, 3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller
  - Range of a at End_{main} is $(2, \ldots)$
- *Spurious interprocedural loops*

Context Insensitivity = Imprecision + Potential Inefficiency

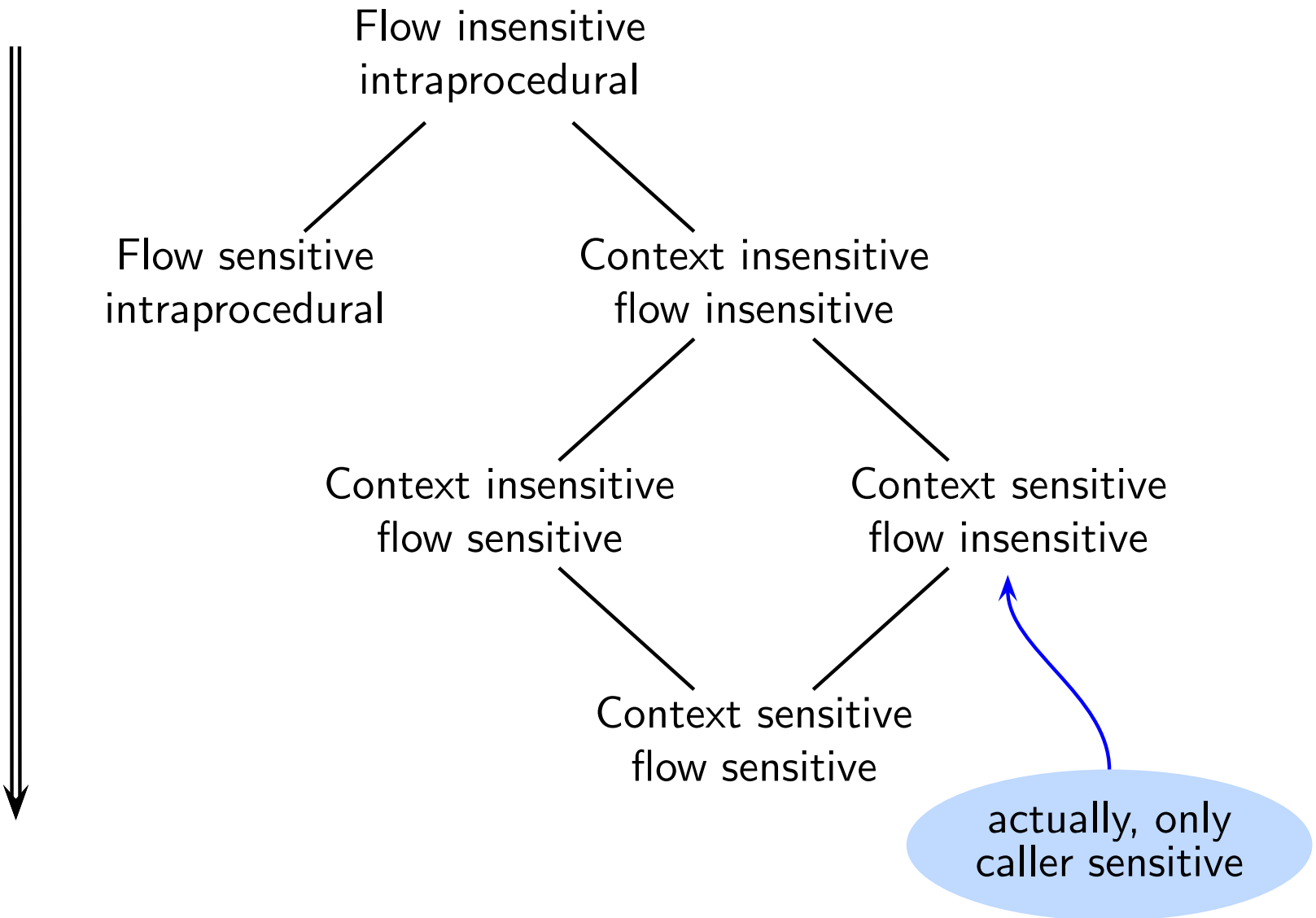


- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3, 3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller
  - Range of a at End_{main} is $(2, \ldots)$
- *Spurious interprocedural loops*
- *Spurious fixed point computations*

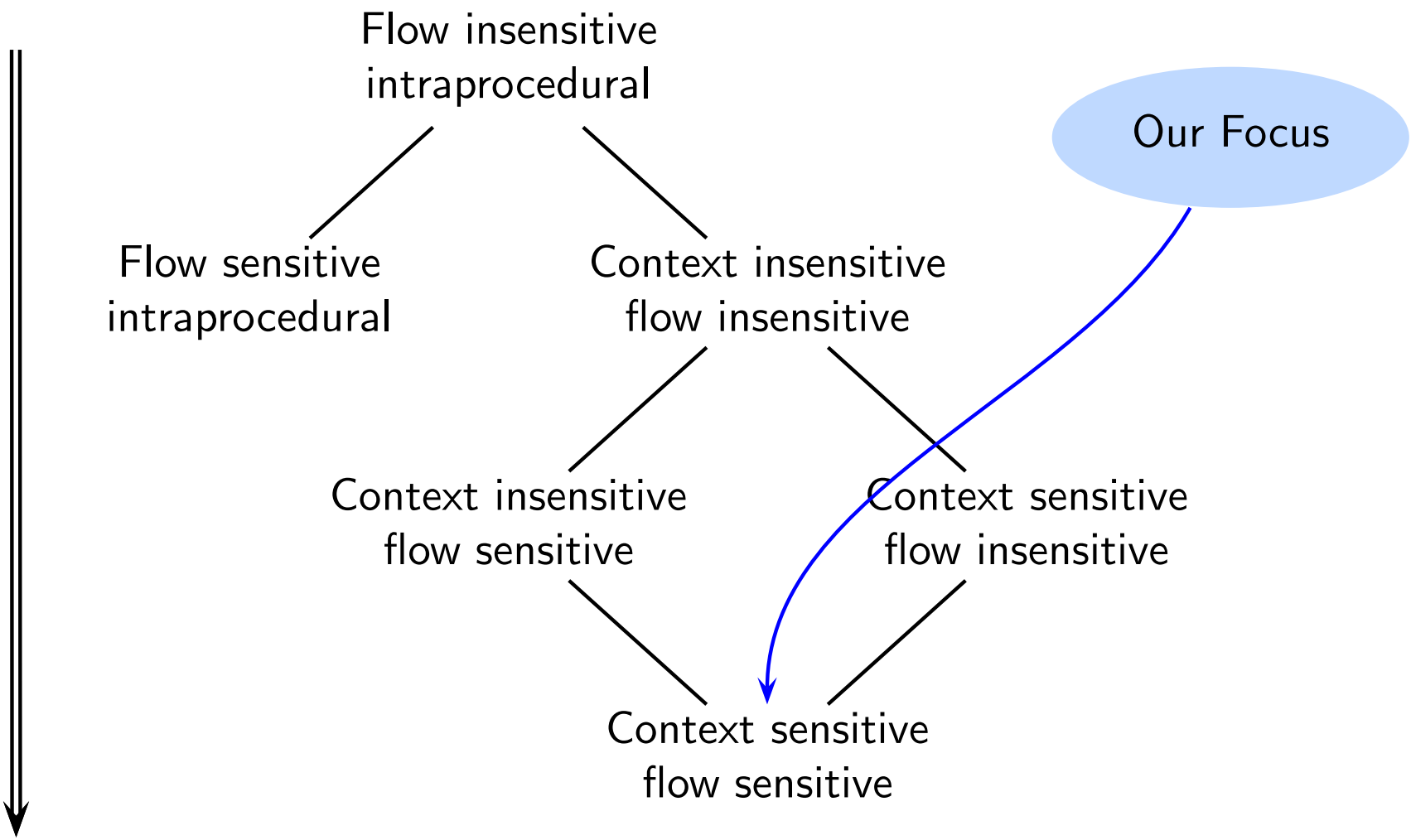
Increasing Precision in Data Flow Analysis



Increasing Precision in Data Flow Analysis



Increasing Precision in Data Flow Analysis



Part 2

Top-Down Approach

Classical Top-Down Methods

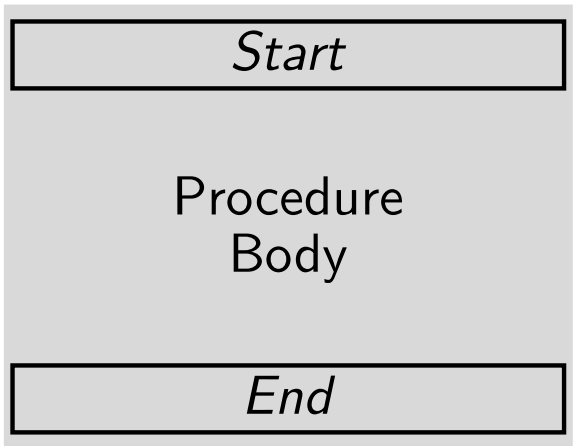
- Call Strings Approach

  Superseded by value contexts (next few slides)

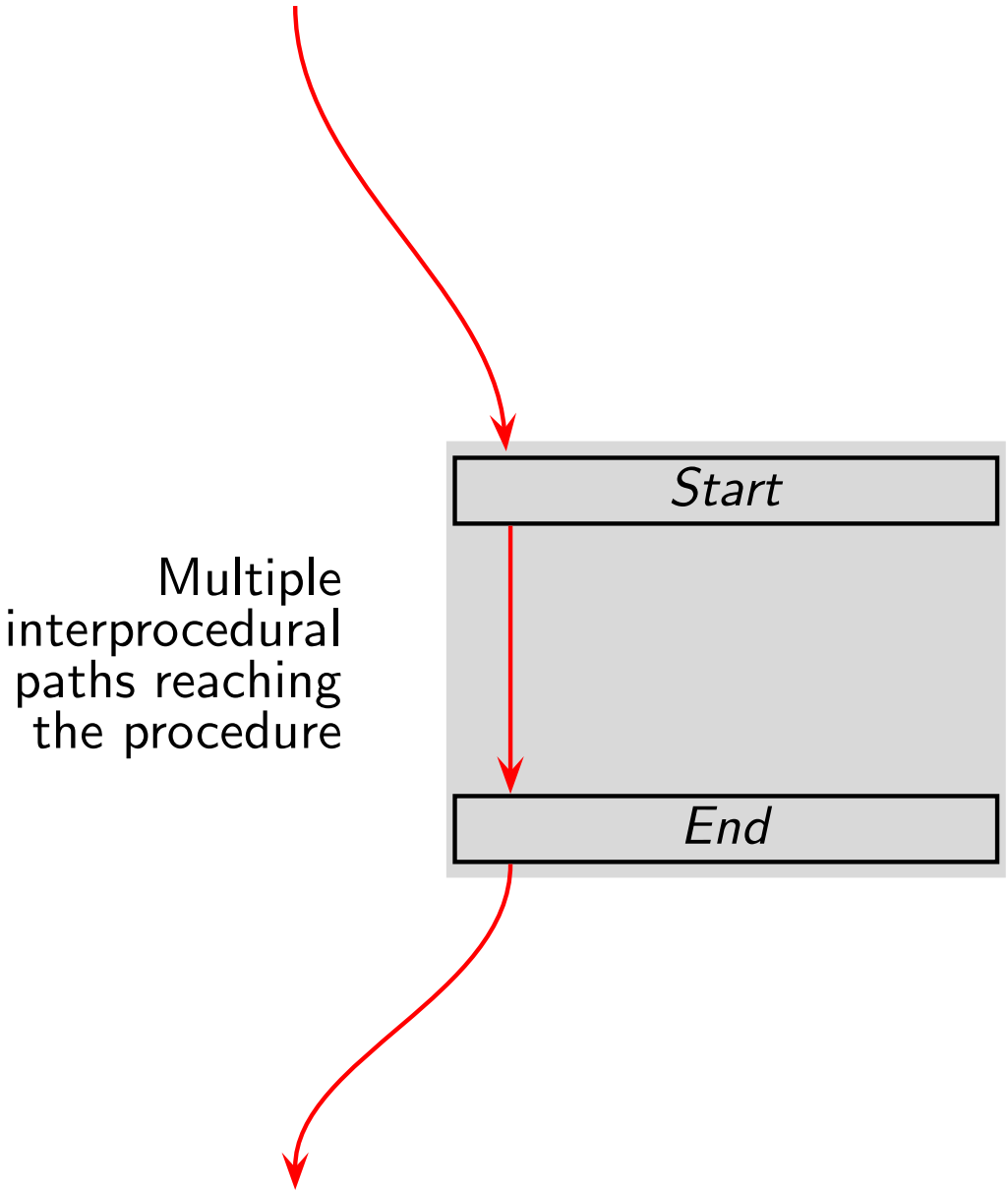
- Graph Reachability
  - Not applicable to points-to analysis
  - Requires flow functions of the form $V \rightarrow V$
    (V is the set of variables)

    Points-to flow functions are of the form $V \rightarrow 2^V$
    (or relations of the form $V \times V$)

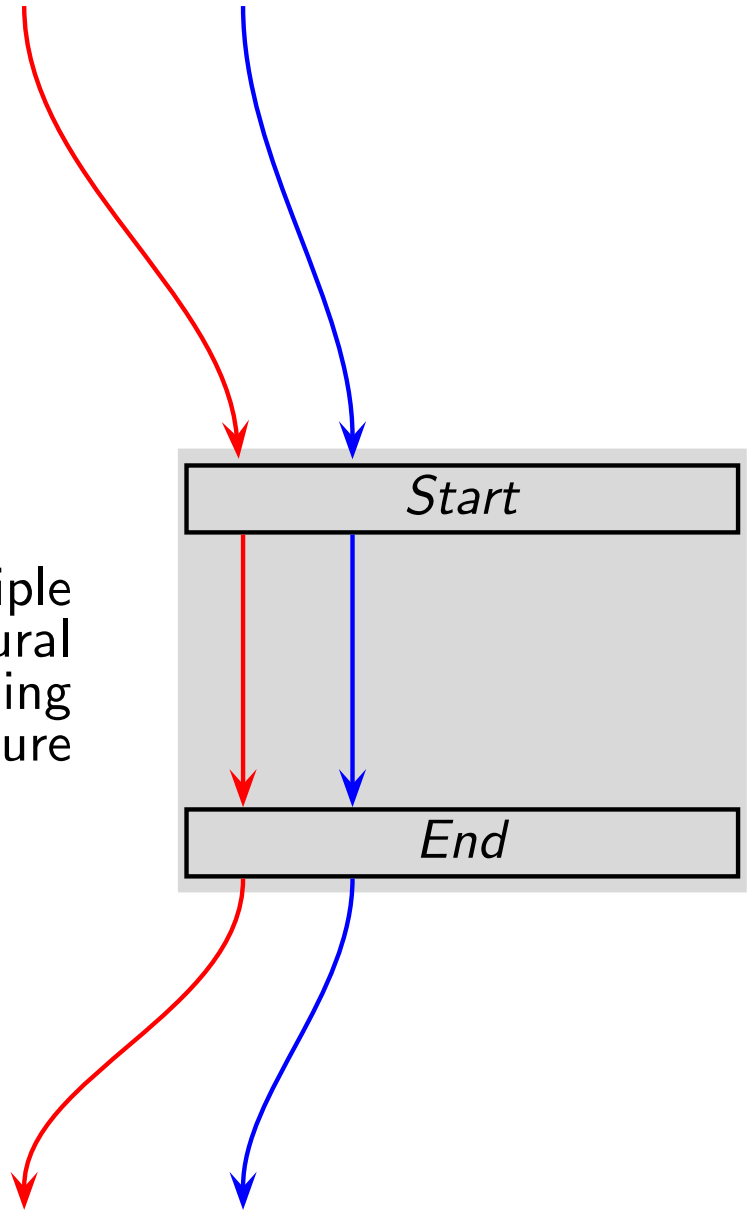
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



Value Contexts (CC-2008, SAS-2012, SOAP-2013)

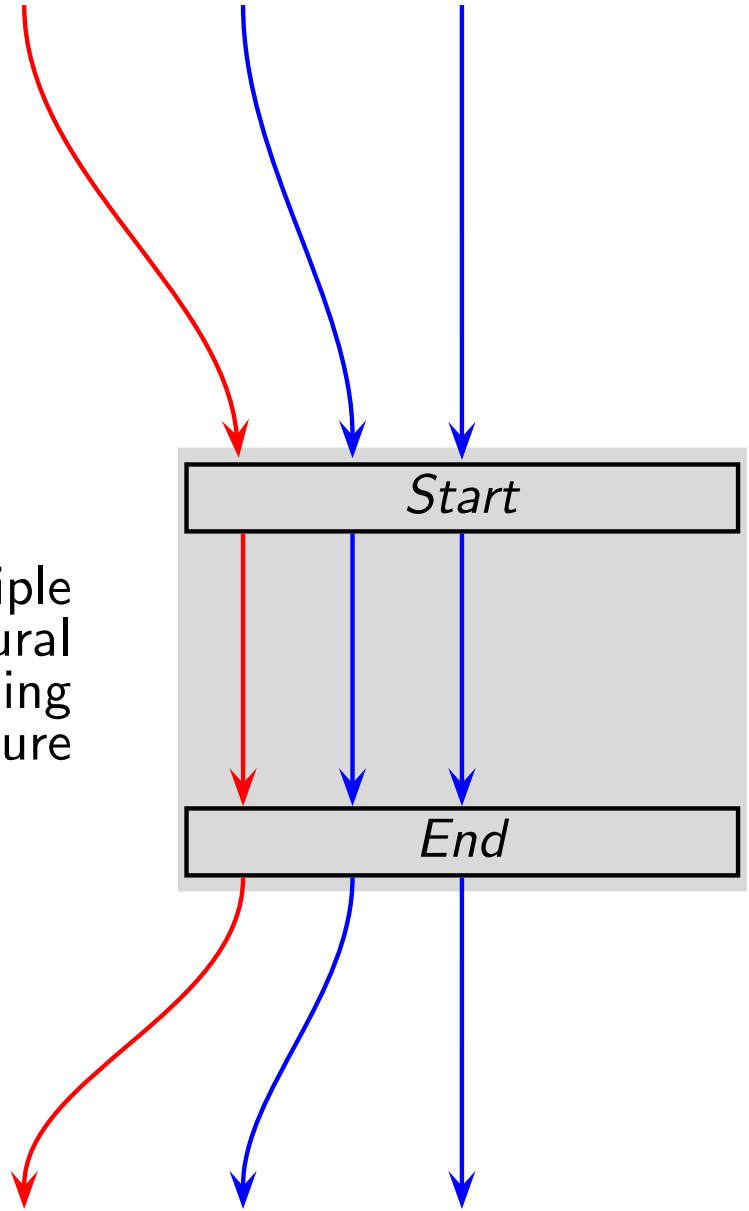


Value Contexts (CC-2008, SAS-2012, SOAP-2013)



Multiple interprocedural paths reaching the procedure

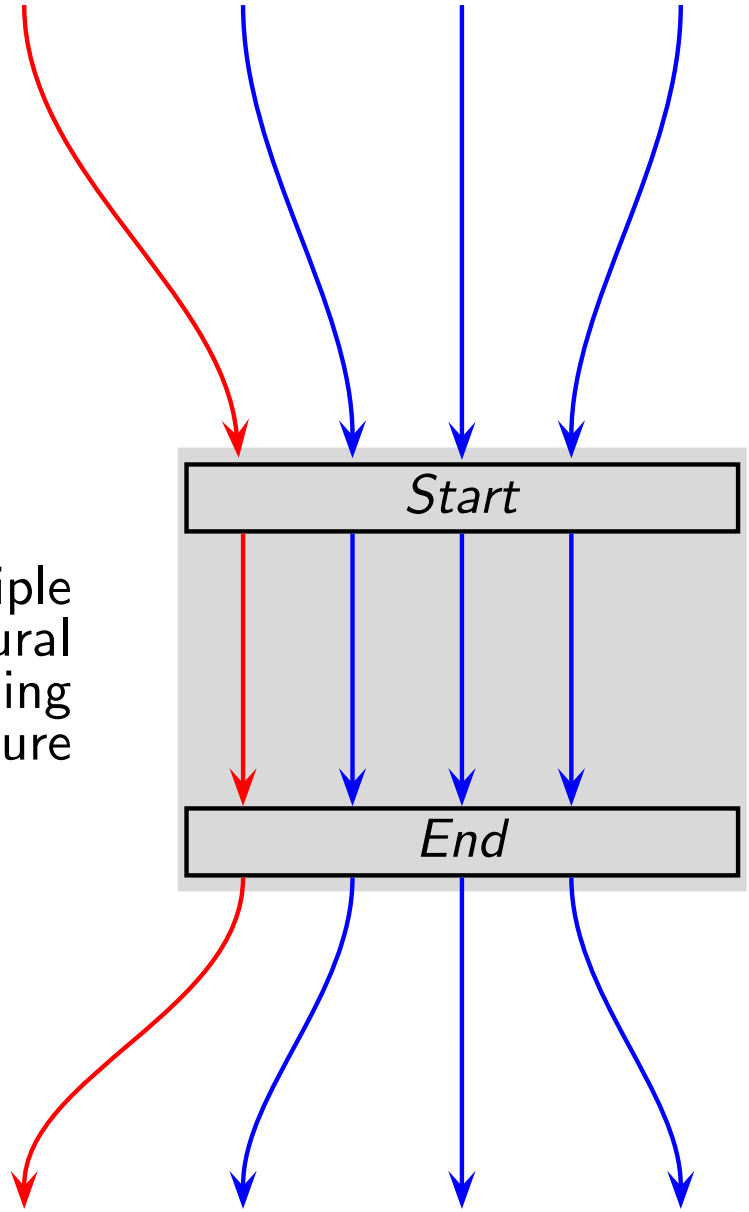
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



Multiple interprocedural paths reaching the procedure

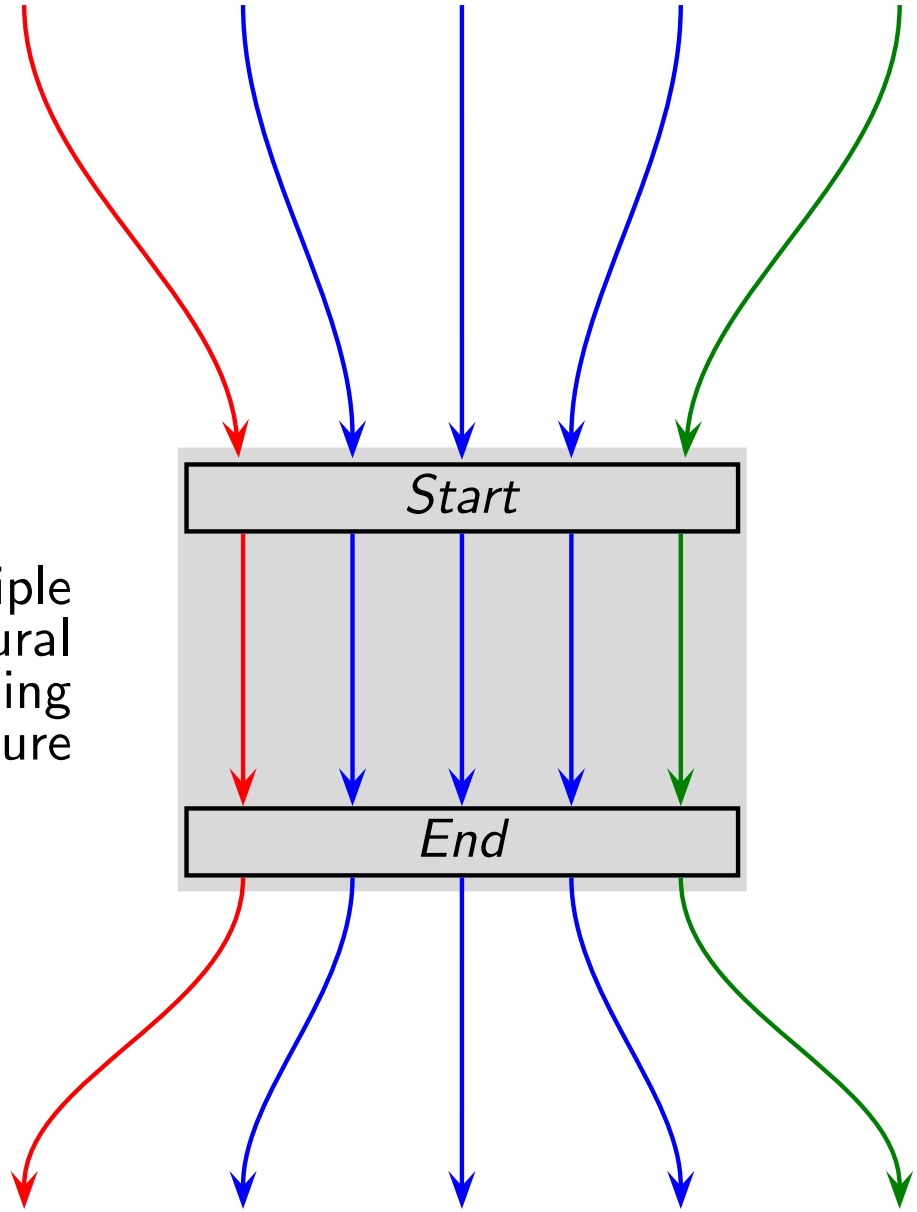
Value Contexts (CC-2008, SAS-2012, SOAP-2013)

Multiple interprocedural paths reaching the procedure

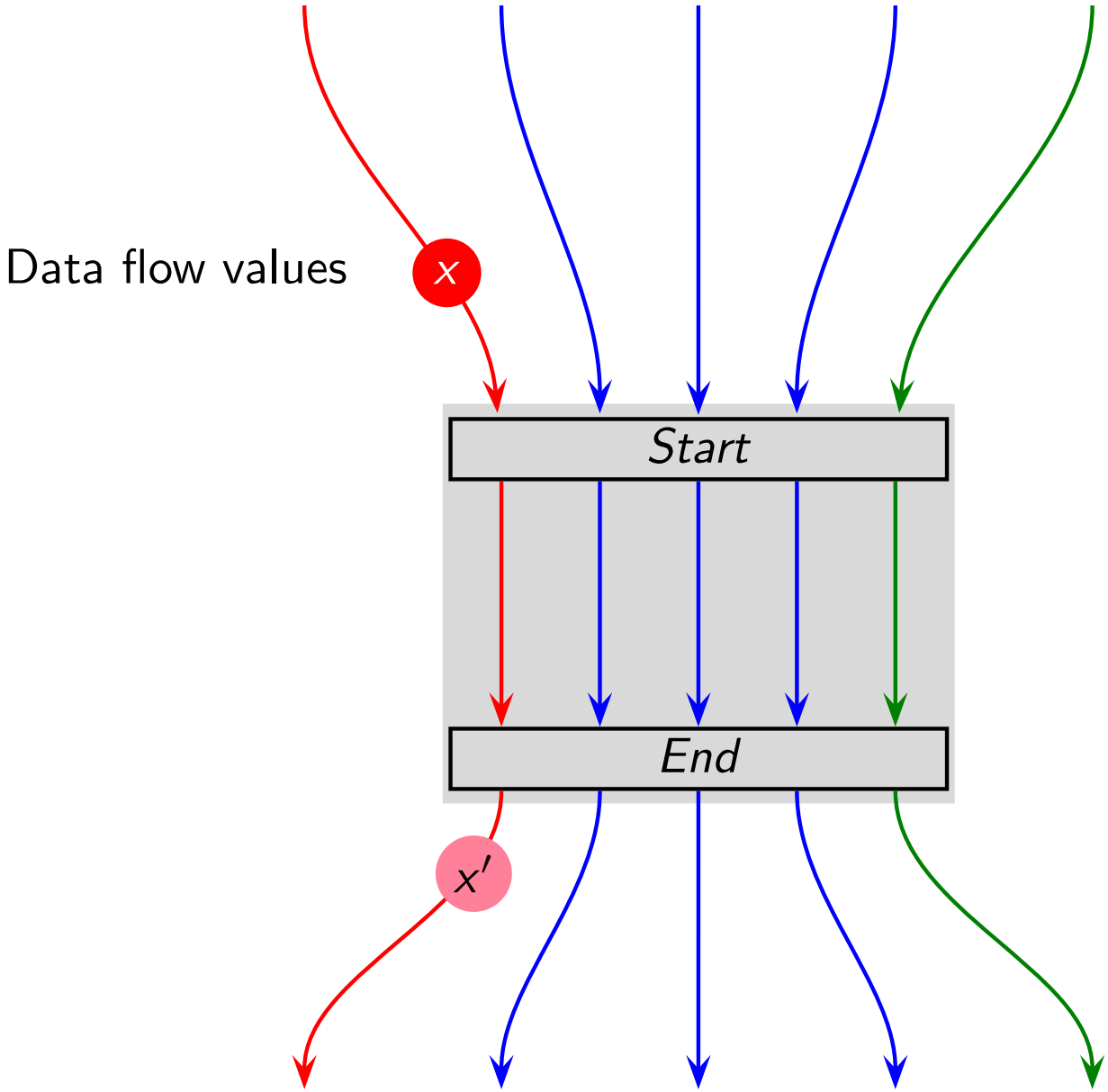


Value Contexts (CC-2008, SAS-2012, SOAP-2013)

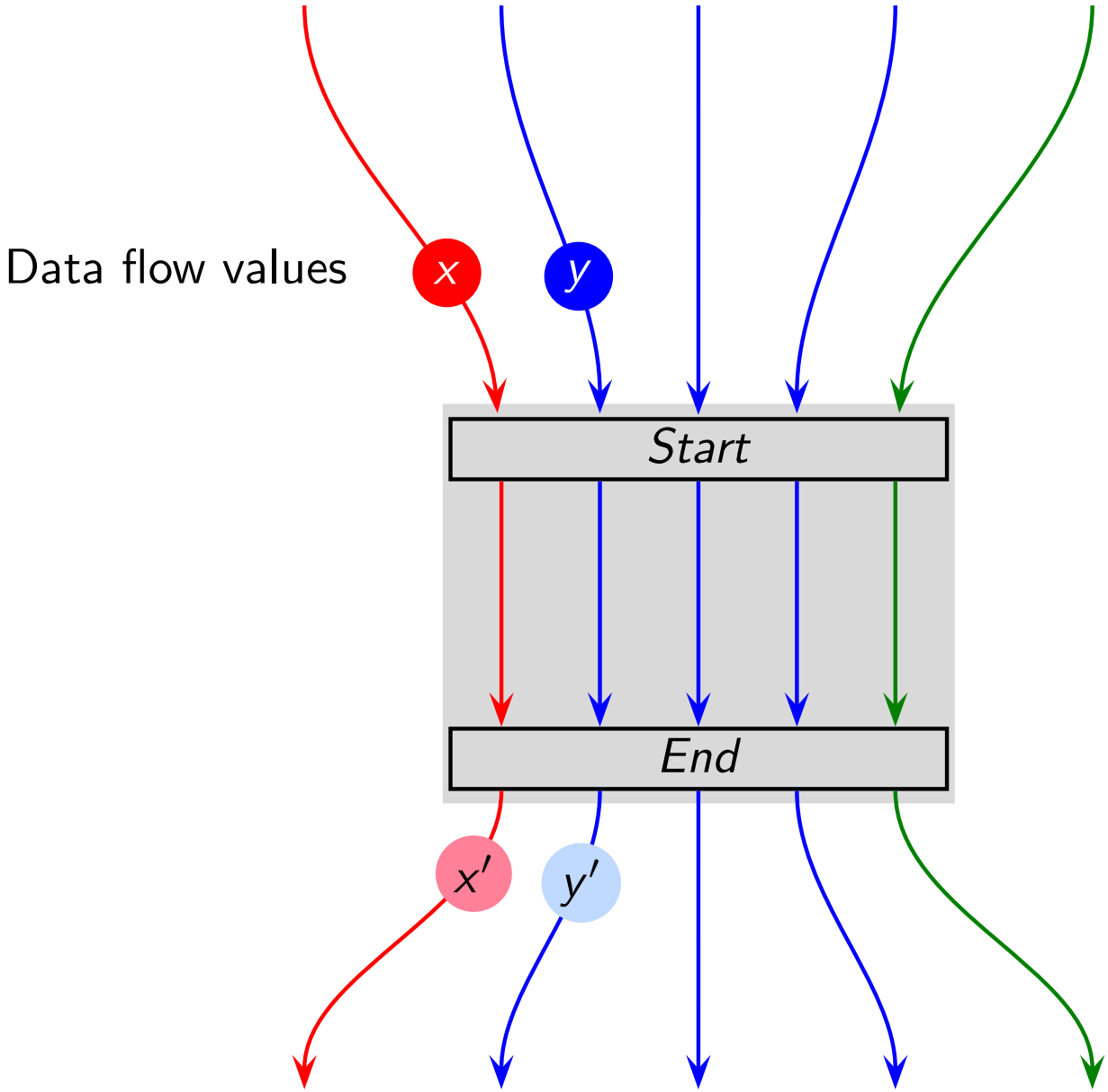
Multiple interprocedural paths reaching the procedure



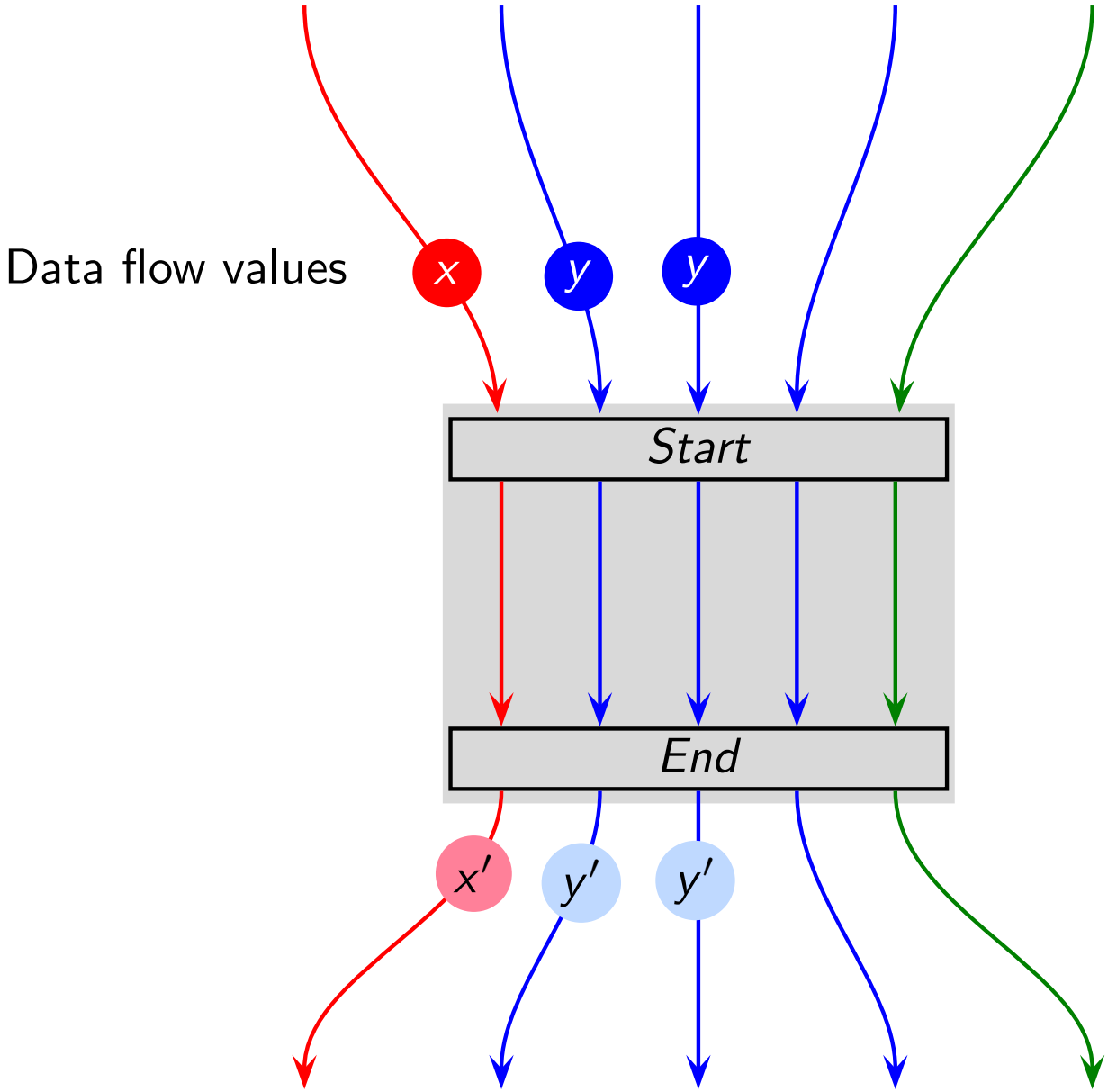
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



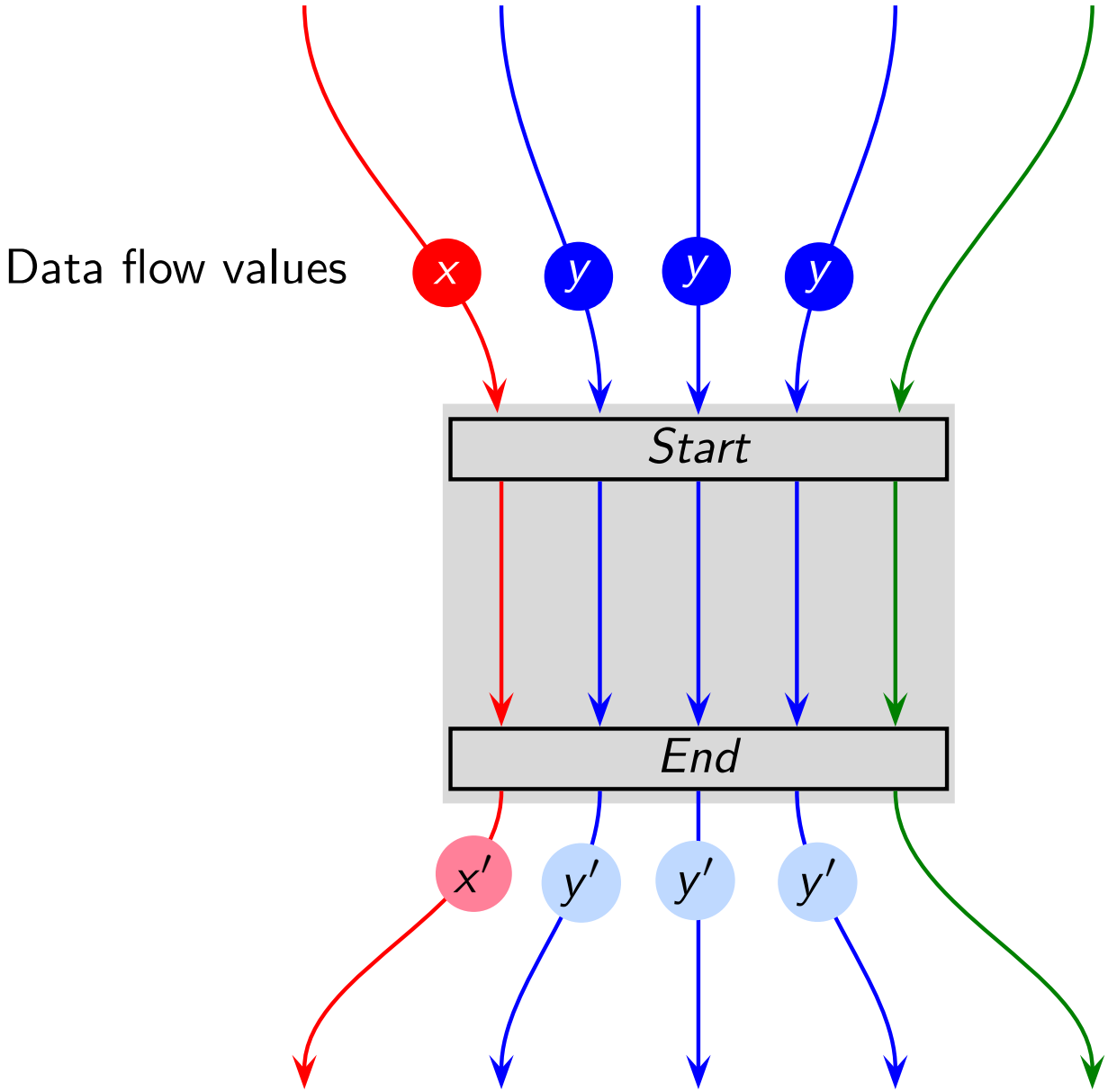
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



Value Contexts (CC-2008, SAS-2012, SOAP-2013)

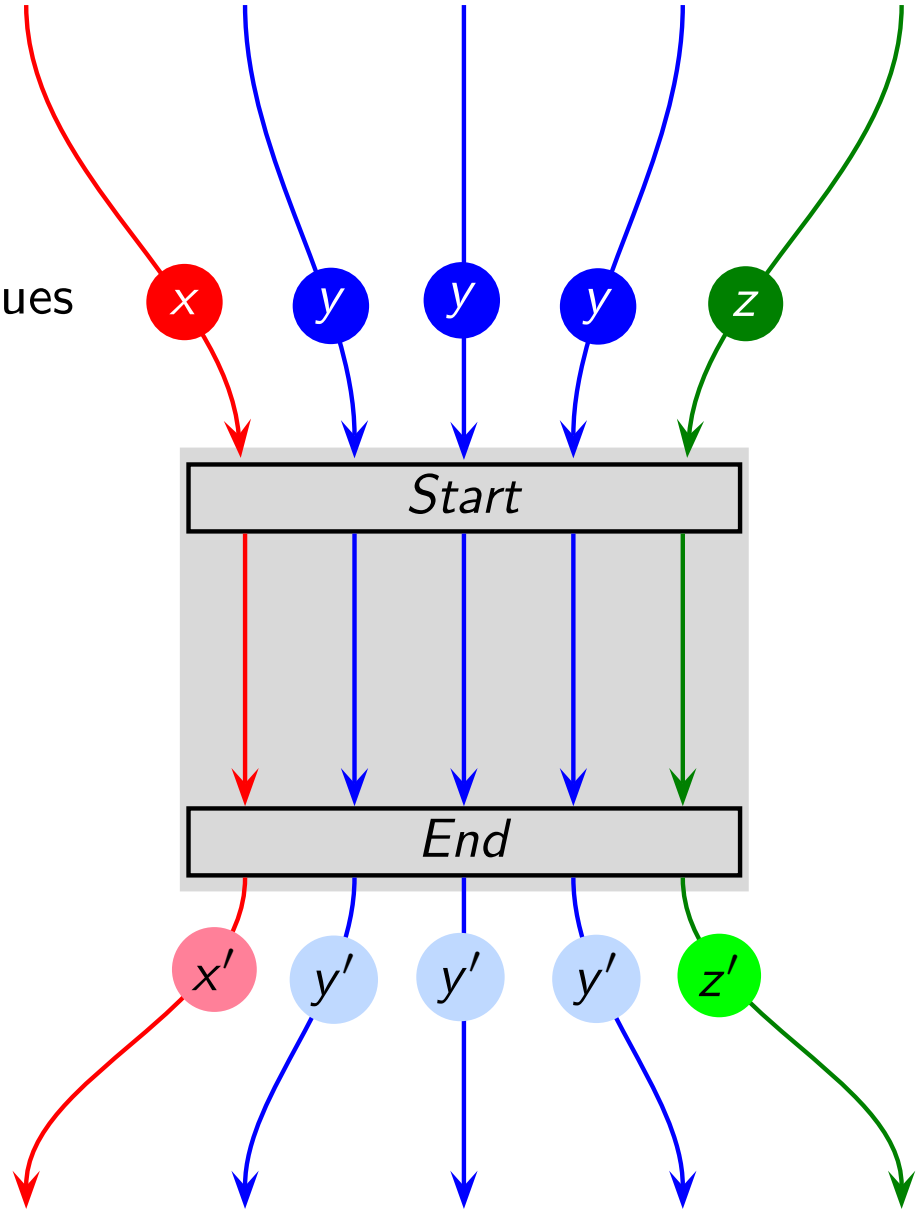


Value Contexts (CC-2008, SAS-2012, SOAP-2013)

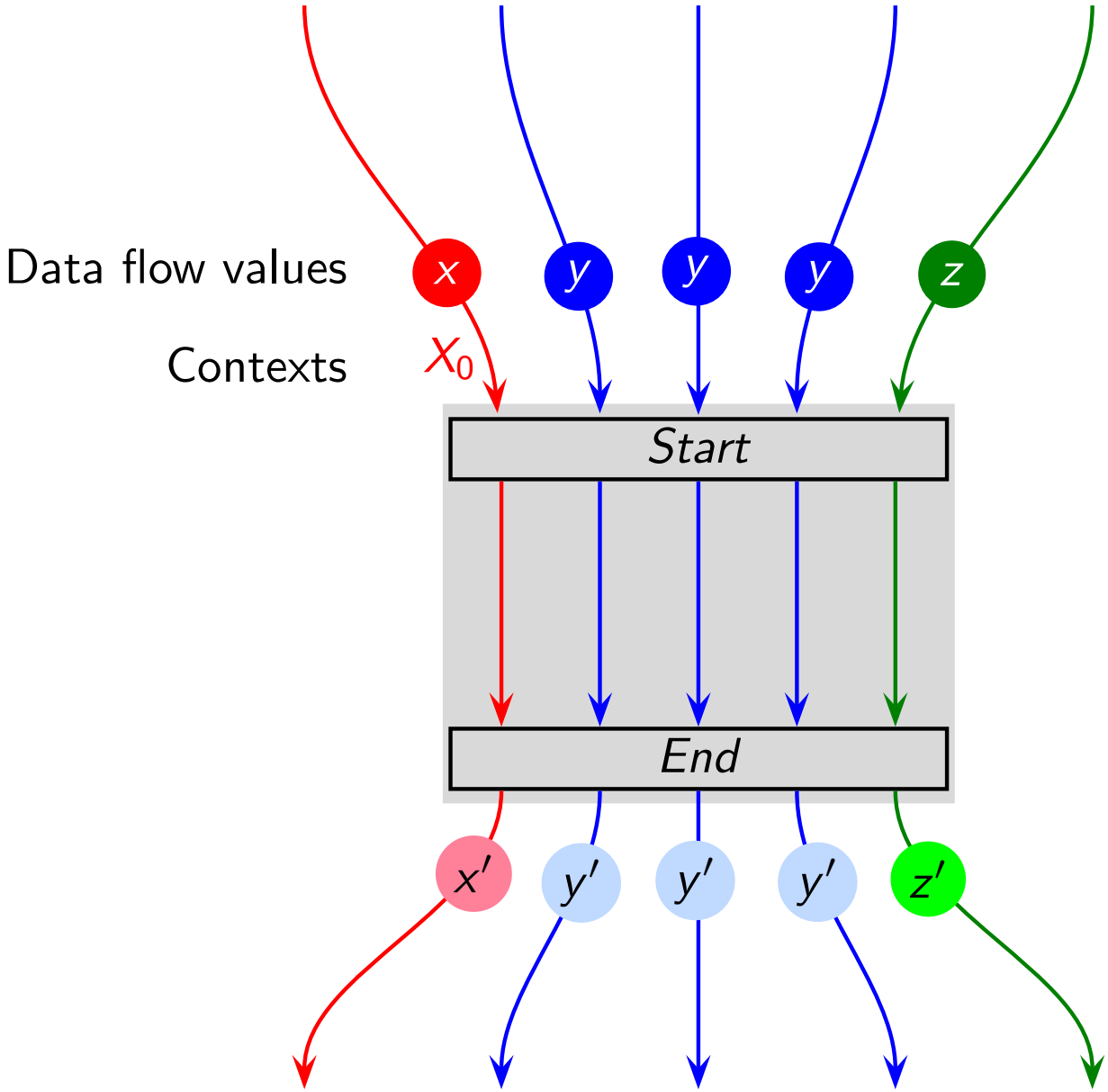


Value Contexts (CC-2008, SAS-2012, SOAP-2013)

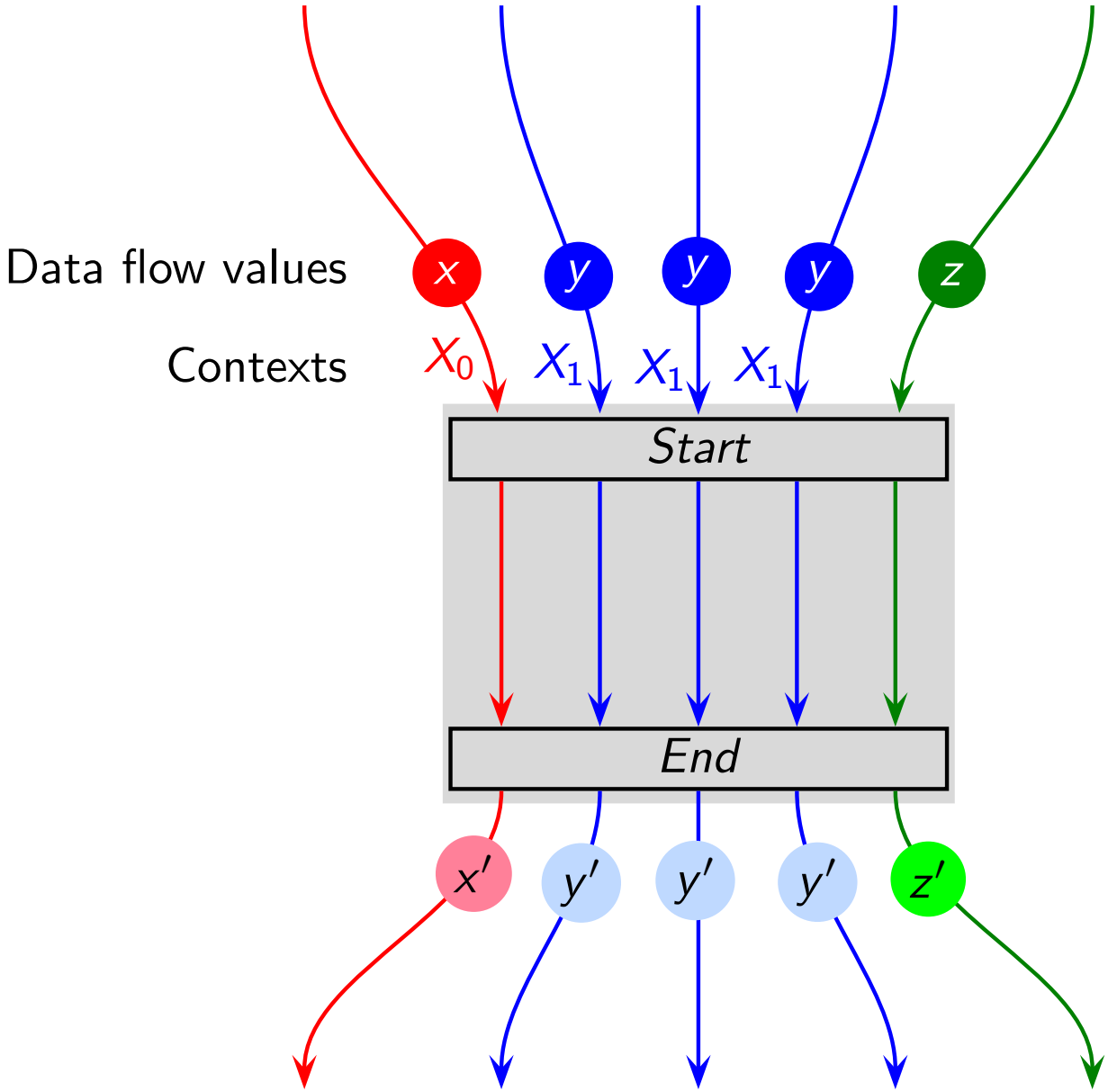
Data flow values



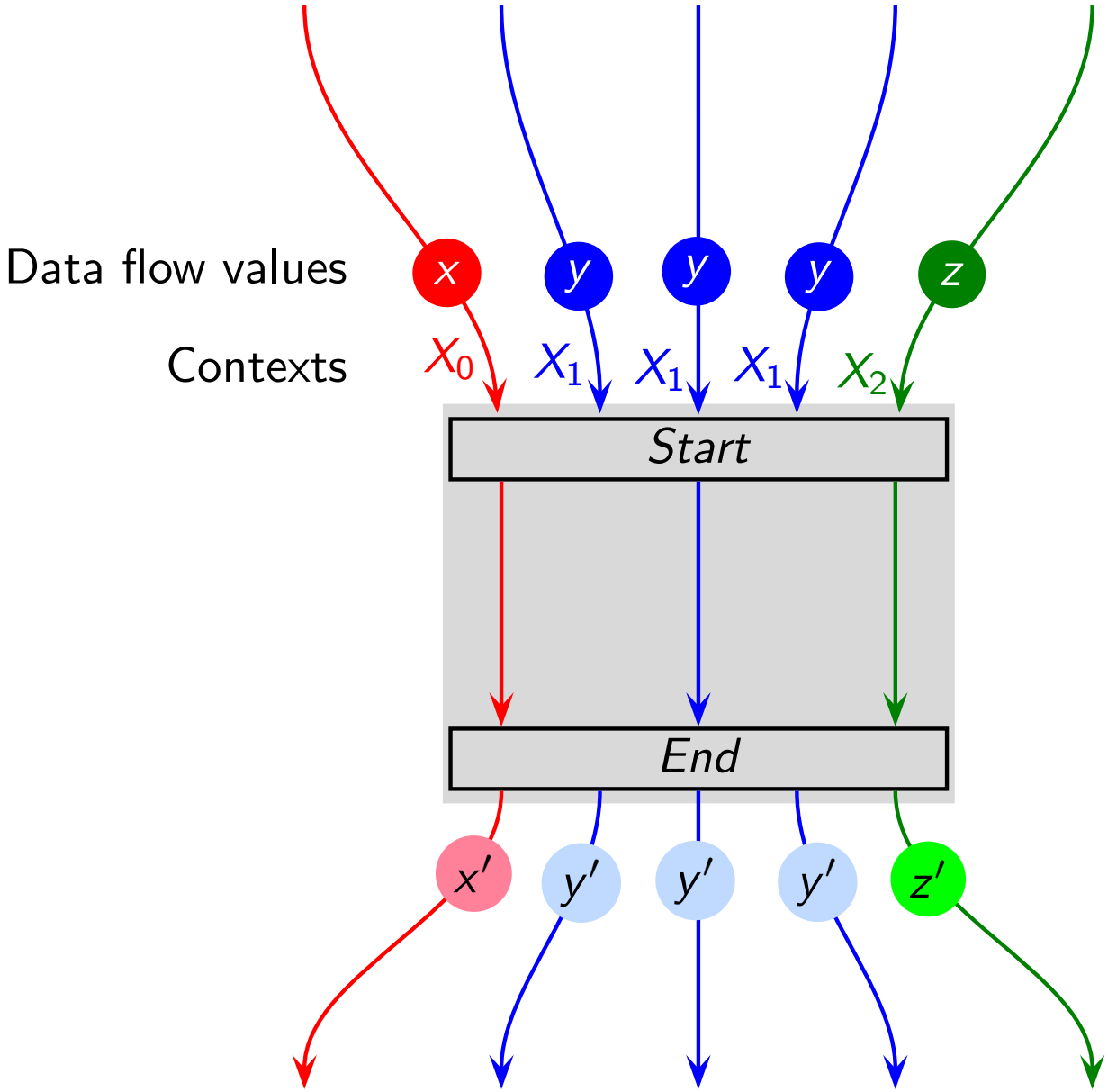
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



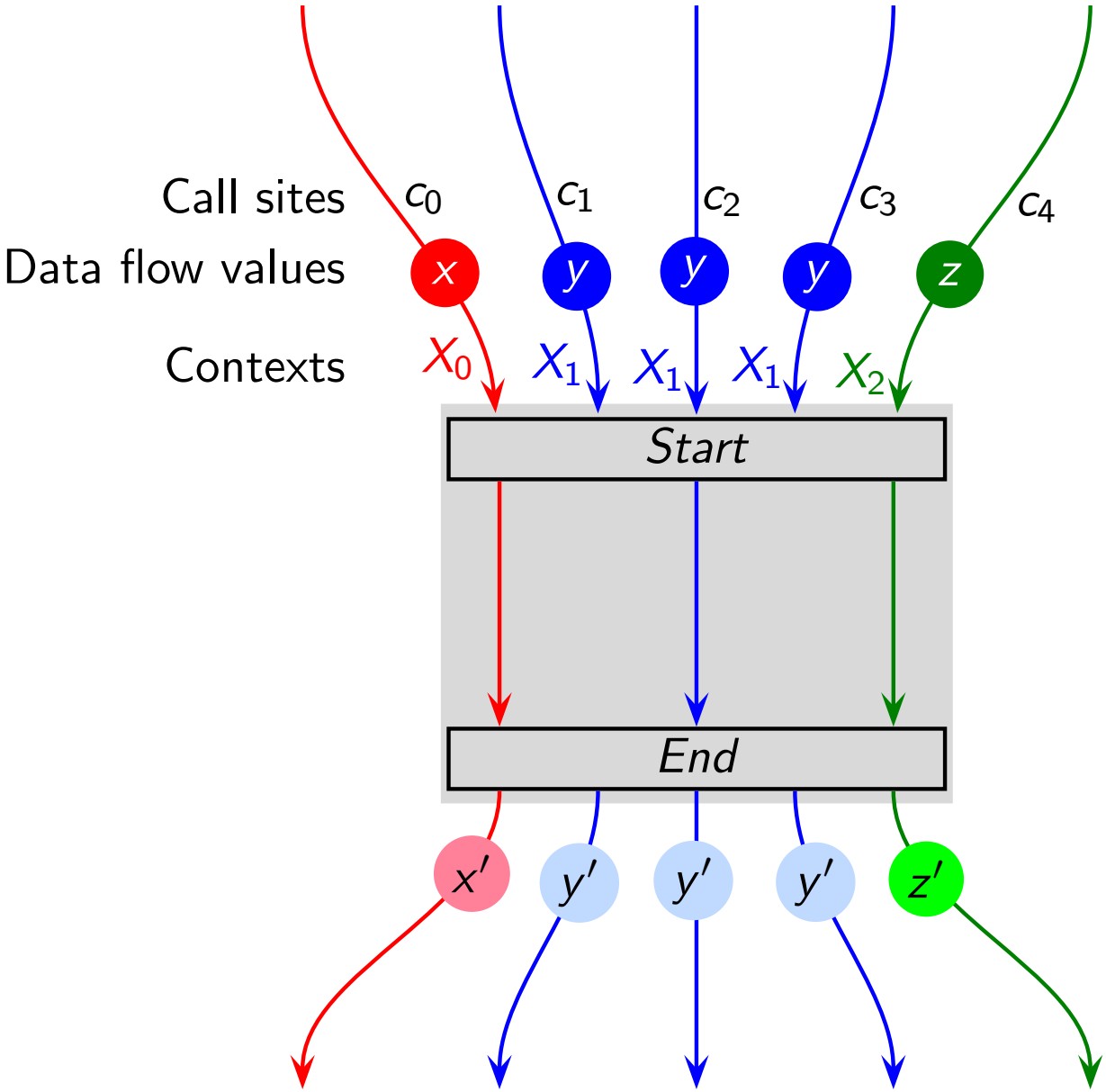
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



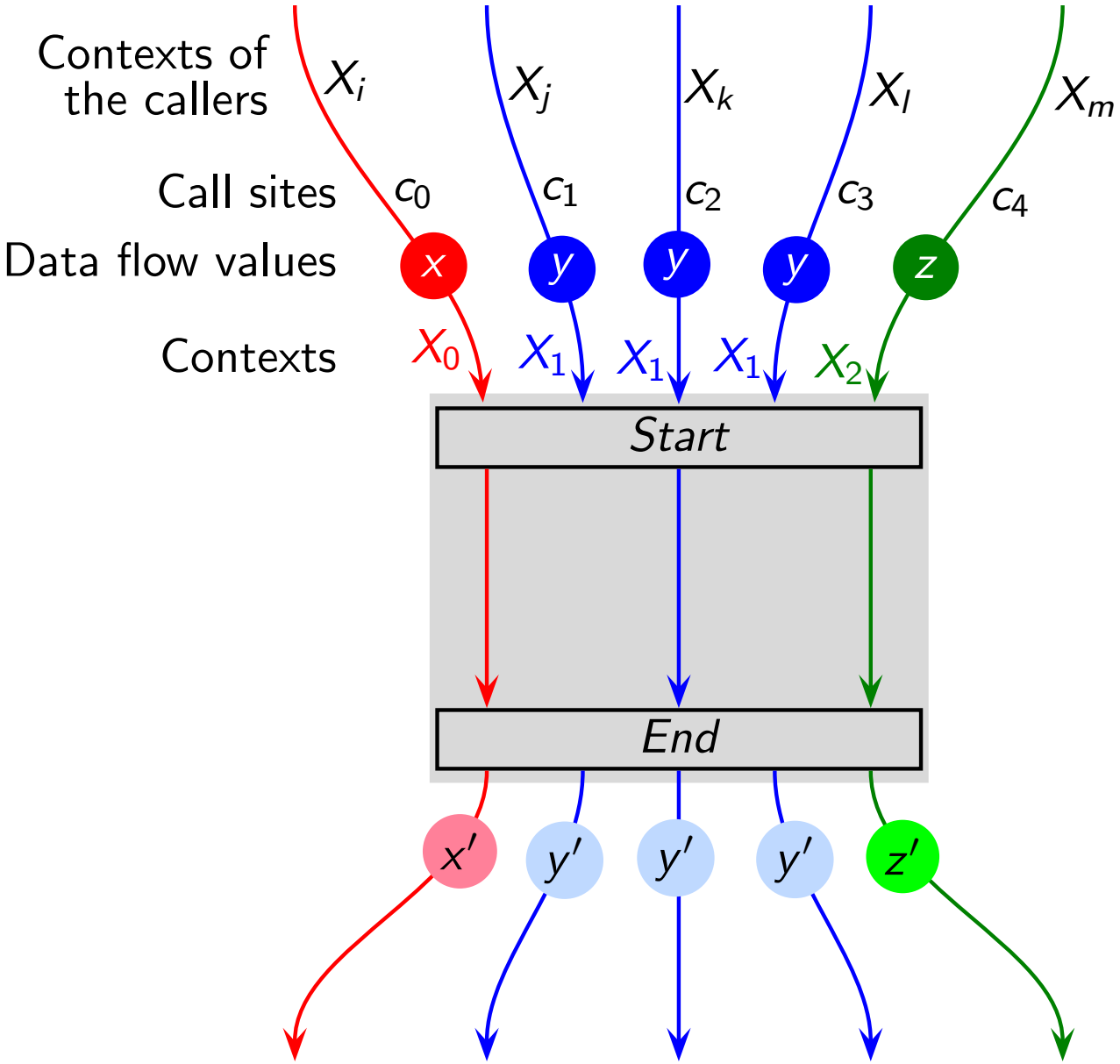
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



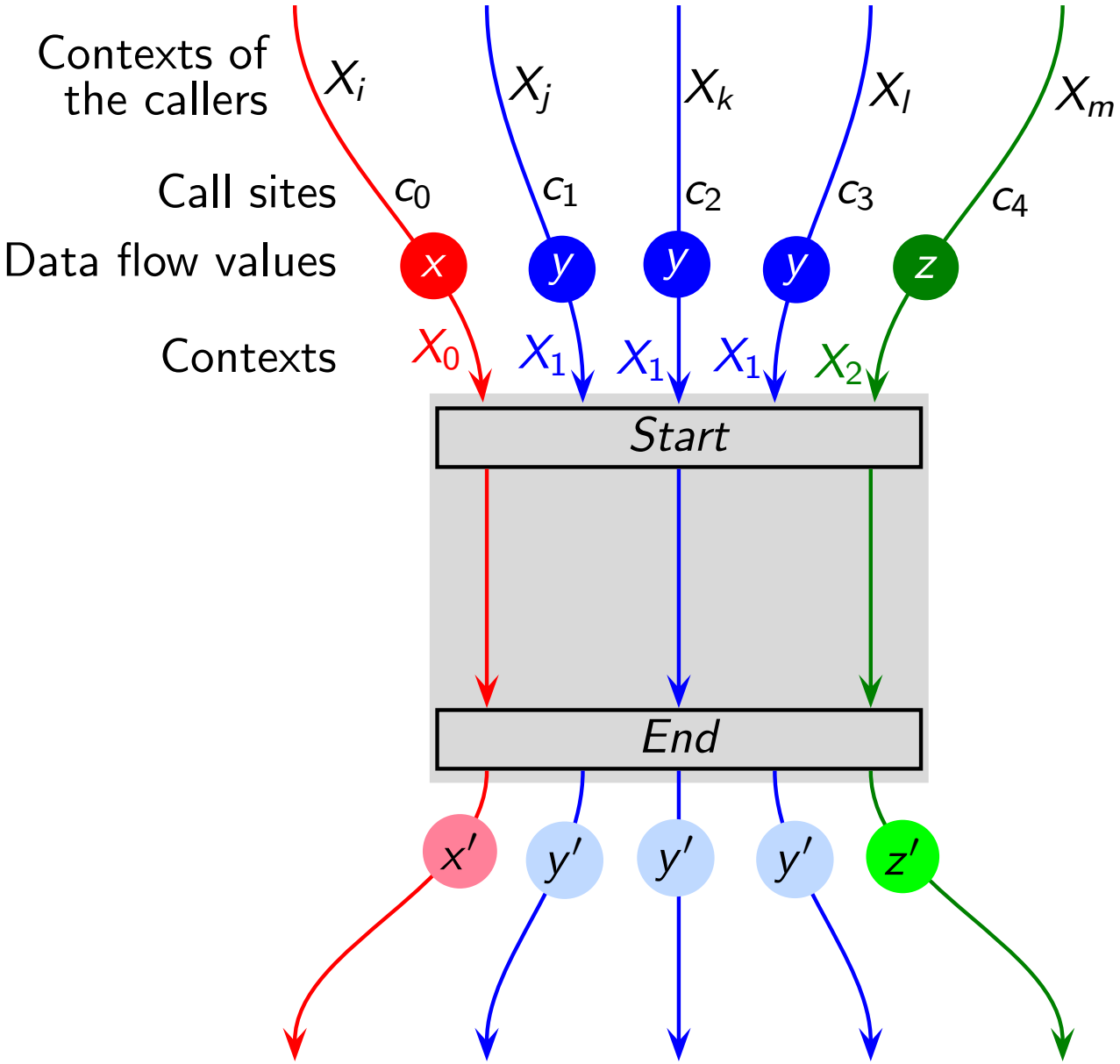
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



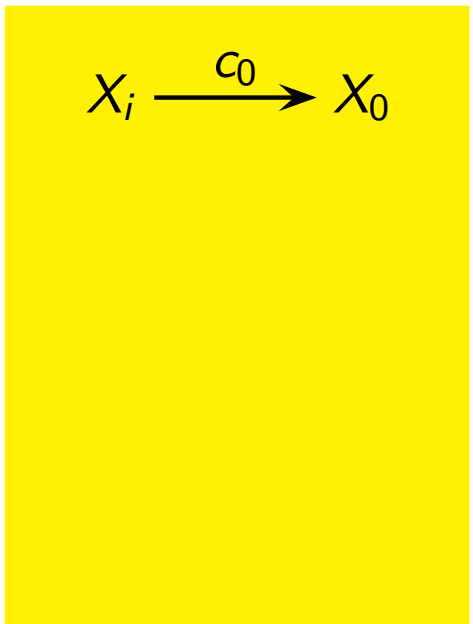
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



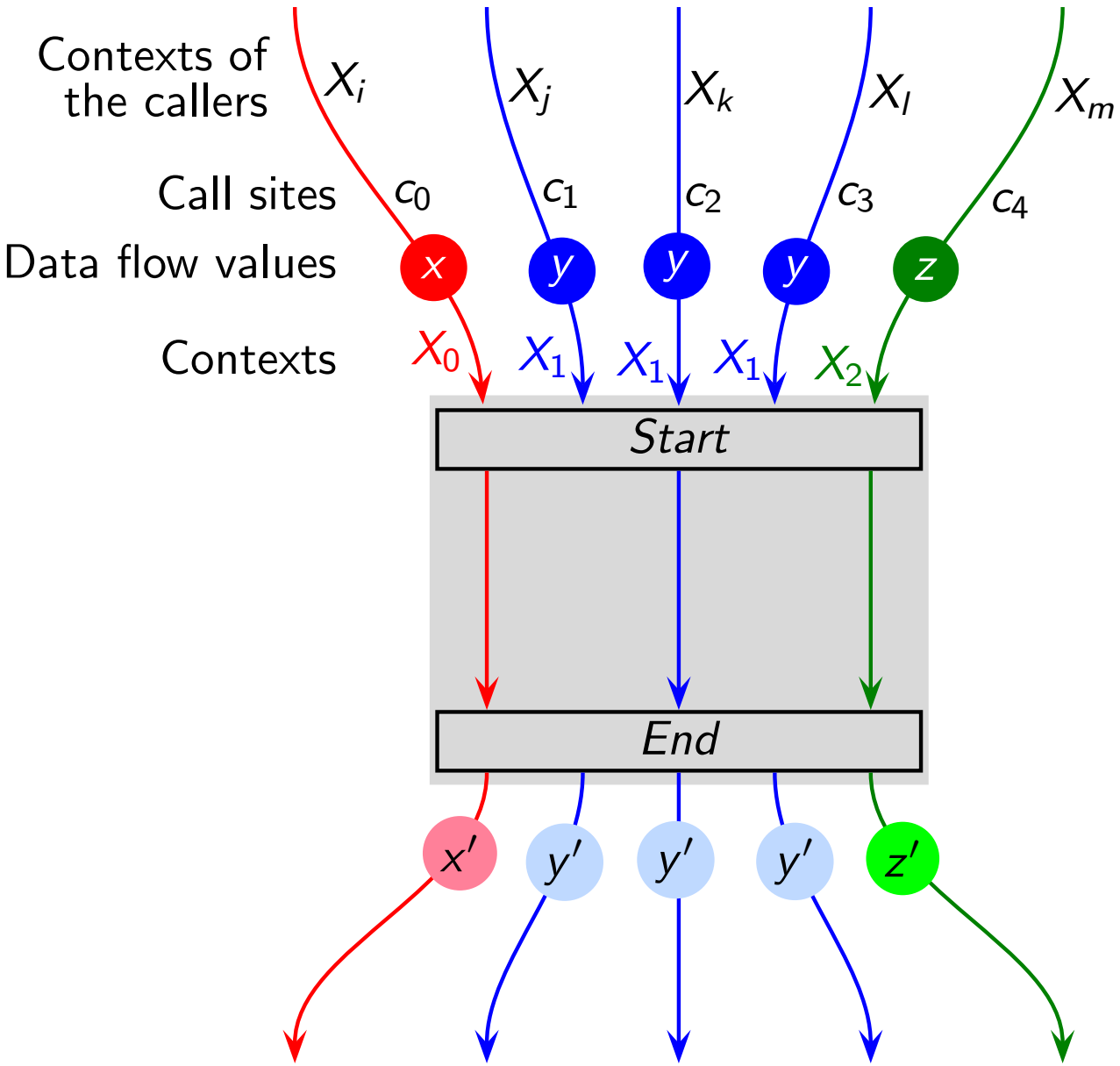
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



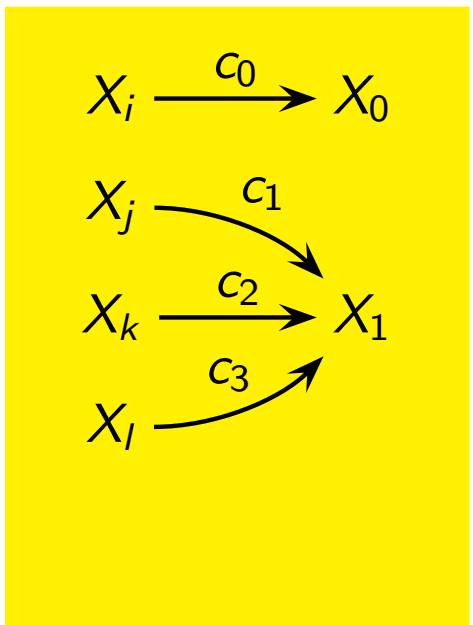
Context transition graph



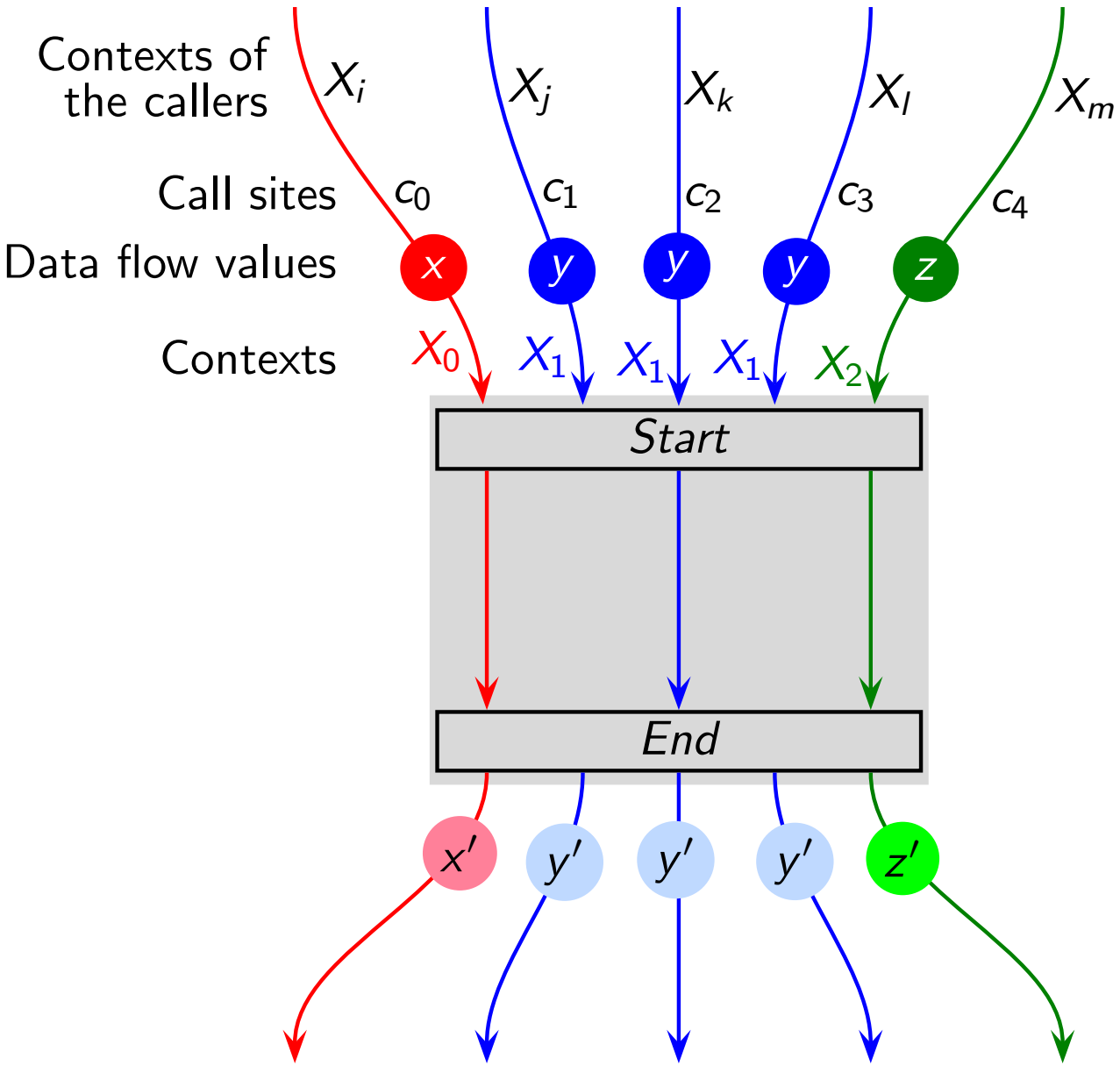
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



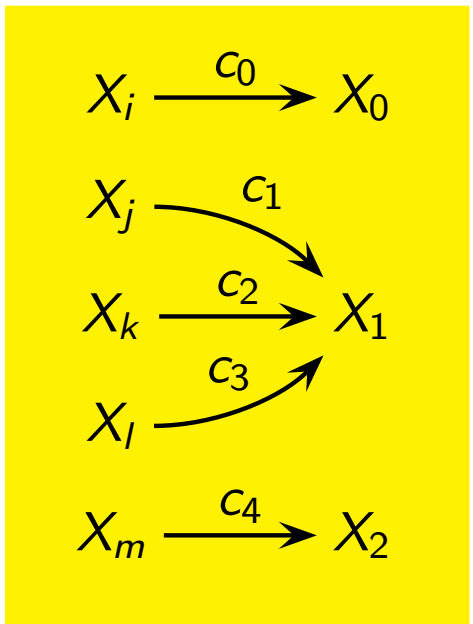
Context transition graph



Value Contexts (CC-2008, SAS-2012, SOAP-2013)



Context transition graph



Value Contexts (CC-2008, SAS-2012, SOAP-2013)

Analyze a procedure once for an input data flow value

- The number of times a procedure is analyzed reduces dramatically
- Similar to the tabulation based method of functional approach [Sharir-Pnueli, 1981]

  However,

  - Value contexts record calling contexts too
    Useful for context matching across program analyses
  - Can avoid some reprocessing even when a new input value is found

Empirical Observations About Value Contexts

- Reaching definitions analysis in GCC 4.2.0 (CC-2008)

  Analysis of Towers of Hanoi

  - Time brought down from 3.973×10^6 ms to 2.37 ms
  - No of call strings brought down from 10^6+ to 8

Empirical Observations About Value Contexts

- Reaching definitions analysis in GCC 4.2.0 (CC-2008)

  Analysis of Towers of Hanoi

  - Time brought down from 3.973×10^6 ms to 2.37 ms
  - No of call strings brought down from 10^6+ to 8

- Generic Interprocedural Analysis Framework in SOOT (SOAP-2013)

  Empirical observations on SPECJVM98 and DaCapo 2006 benchmarks for on-the-fly call graph construction

  - Average number of contexts per procedure lies in the range 4-25
  - Much fewer long call chains than in the default call graph constructed using SPARK
    For length 7, less than 50%
    For length 10, less than 5%

Empirical Observations About Value Contexts

- Reaching definitions analysis in GCC 4.2.0 (CC-2008)

  Analysis of Towers of Hanoi

  - Time brought down from 3.973×10^6 ms to 2.37 ms
  - No of call strings brought down from 10^6+ to 8

- Generic Interprocedural Analysis Framework in SOOT (SOAP-2013)

  Empirical observations on SPECJVM98 and DaCapo 2006 benchmarks for on-the-fly call graph construction

  - Average number of contexts per procedure lies in the range 4-25
  - Much fewer long call chains than in the default call graph constructed using SPARK
    For length 7, less than 50%
    For length 10, less than 5%

- And yet, it is insufficient for scaling points-to analysis even with bypassing of irrelevant data across calls (ISMM-2017)

Part 3

Bottom Up Approaches

Summarizing a Procedure for Points-to Analysis

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

Summarizing a Procedure for Points-to Analysis

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

- If data dependency exists, it can be eliminated and the control flow would be 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 dependency

- If data dependency exists, it can be eliminated and the control flow would be 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 dependency

- If data dependency exists, it can be eliminated and the control flow would be redundant
- If data dependency does not exist, then redundant memory updates can be eliminated

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 dependency

- If data dependency exists, it can be eliminated and the control flow would be redundant
- If data dependency does not exist, then redundant memory updates can be eliminated

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

$\Downarrow$

$y = \&b;$
$x = \&b;$

Summarizing a Procedure for Points-to Analysis

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

- If data dependency exists, it can be eliminated and the control flow would be redundant
- If data dependency does not exist, then redundant memory updates can be eliminated
- If data dependency is unknown, then more information is required which becomes available when inlined at call sites

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

Summarizing a Procedure for Points-to Analysis

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

- If data dependency exists, it can be eliminated and the control flow would be redundant
- If data dependency does not exist, then redundant memory updates can be eliminated
- If data dependency is unknown, then more information is required which becomes available when inlined at call sites
  - Control flow required

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

Summarizing a Procedure for Points-to Analysis

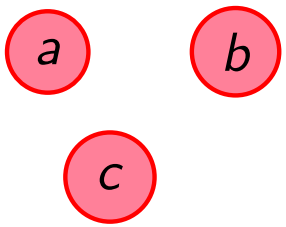
A flow-sensitive analysis requires control flow to be recorded between memory updates that share data dependency

- If data dependency exists, it can be eliminated and the control flow would be redundant
- If data dependency does not exist, then redundant memory updates can be eliminated
- If data dependency is unknown, then more information is required which becomes available when inlined at call sites
  - Control flow required
  - Some accesses of pointees have definitions in the callers

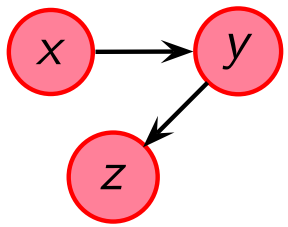
1. $y = \&b;$
2. $*x = \&a;$

Memory and Memory Transformer

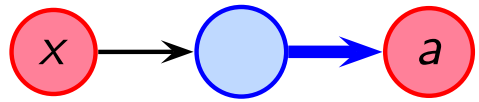
Memory in absence of pointers



Memory in presence of pointers

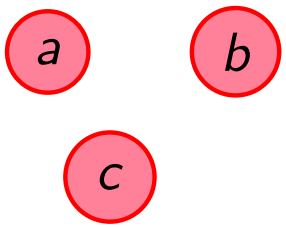


Memory Transformer

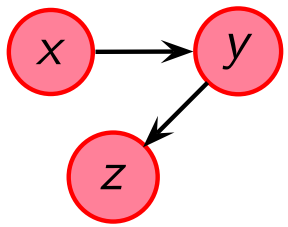


Memory and Memory Transformer

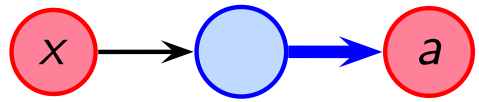
Memory in absence of pointers



Memory in presence of pointers



Memory Transformer

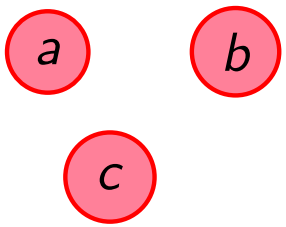


For memory transformer,

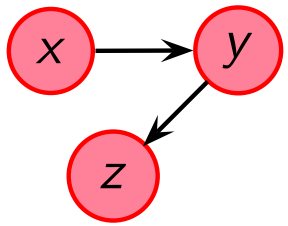
- Blue edges $\Rightarrow$ information generated
- Black edges $\Rightarrow$ carried forward input information

Memory and Memory Transformer

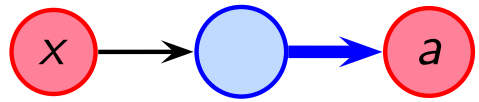
Memory in absence of pointers



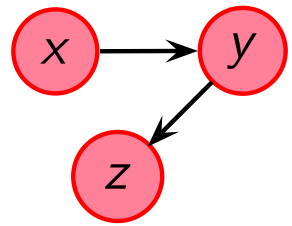
Memory in presence of pointers



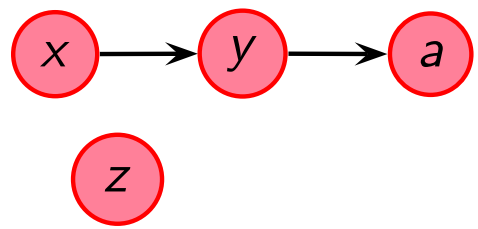
Memory Transformer



Input Memory



Output Memory



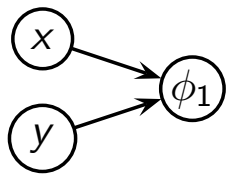
Bottom-up Approaches: The State of the Art

Accesses of pointees that are defined in the callers are represented using placeholders

Bottom-up Approaches: The State of the Art

Accesses of pointees that are defined in the callers are represented using placeholders

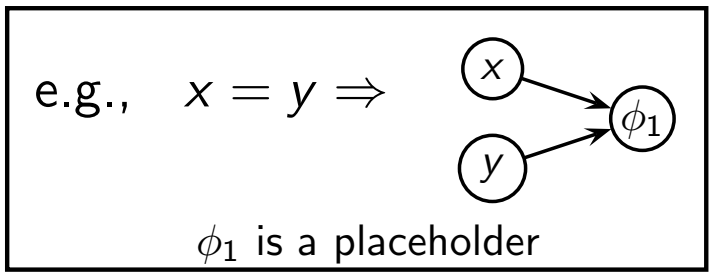
e.g., $x = y \Rightarrow$



ϕ_1 is a placeholder

Bottom-up Approaches: The State of the Art

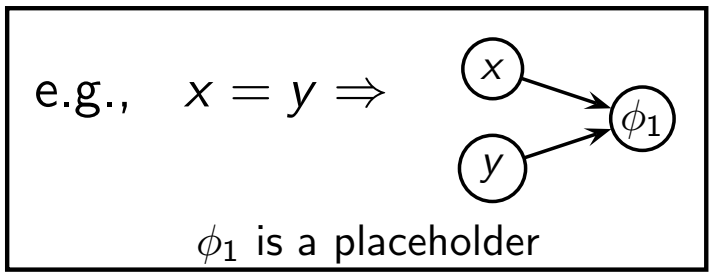
Accesses of pointees that are defined in the callers are represented using placeholders



- Context based analysis [Zhang-PLDI-14, Wilson-PLDI-95, Yu-CGO-10]
  - Use aliases (or points-to relations) present in the caller
  - Construct a collection of partial transfer functions (PTFs)

Bottom-up Approaches: The State of the Art

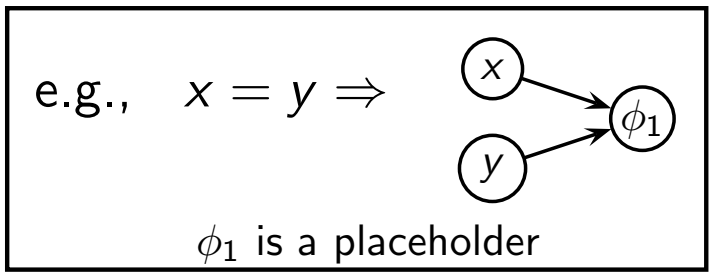
Accesses of pointees that are defined in the callers are represented using placeholders



- Context based analysis [Zhang-PLDI-14, Wilson-PLDI-95, Yu-CGO-10]
  - Use aliases (or points-to relations) present in the caller
  - Construct a collection of partial transfer functions (PTFs)
- Context independent analysis [Sălcianu-VMCAI-05, Madhavan-SAS-12]
  - No aliases assumed in the calling contexts
  - Construct a single procedure summary

Bottom-up Approaches: The State of the Art

Accesses of pointees that are defined in the callers are represented using placeholders



- Context based analysis [Zhang-PLDI-14, Wilson-PLDI-95, Yu-CGO-10]
  - Use aliases (or points-to relations) present in the caller
  - Construct a collection of partial transfer functions (PTFs)
- Context independent analysis [Sălcianu-VMCAI-05, Madhavan-SAS-12]
  - No aliases assumed in the calling contexts
  - Construct a single procedure summary

These approaches are context sensitive

Context Based Bottom-up Approach

The need of multiple partial transfer functions (PTFs)

Example:

1. $x = *y;$
2. $*z = q;$
3. $p = *y;$

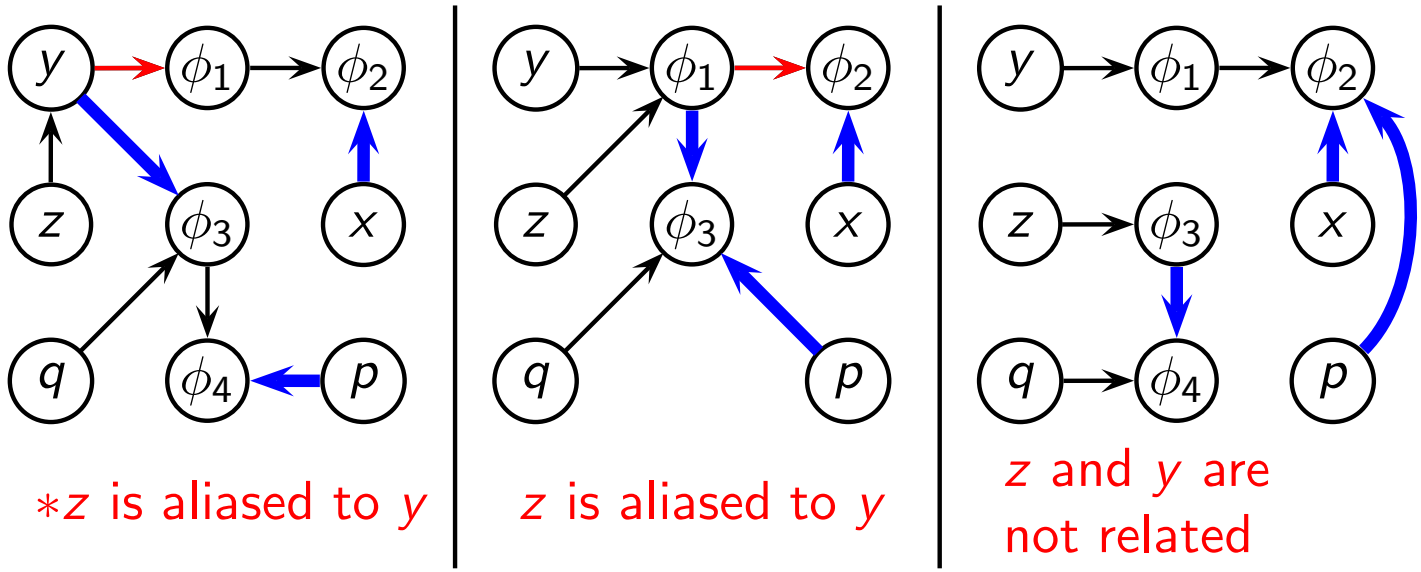
Two dereferences of y are separated by a possibly side-effect causing statement through z

Context Based Bottom-up Approach

The need of multiple partial transfer functions (PTFs)

Example:

1. $x = *y;$
2. $*z = q;$
3. $p = *y;$



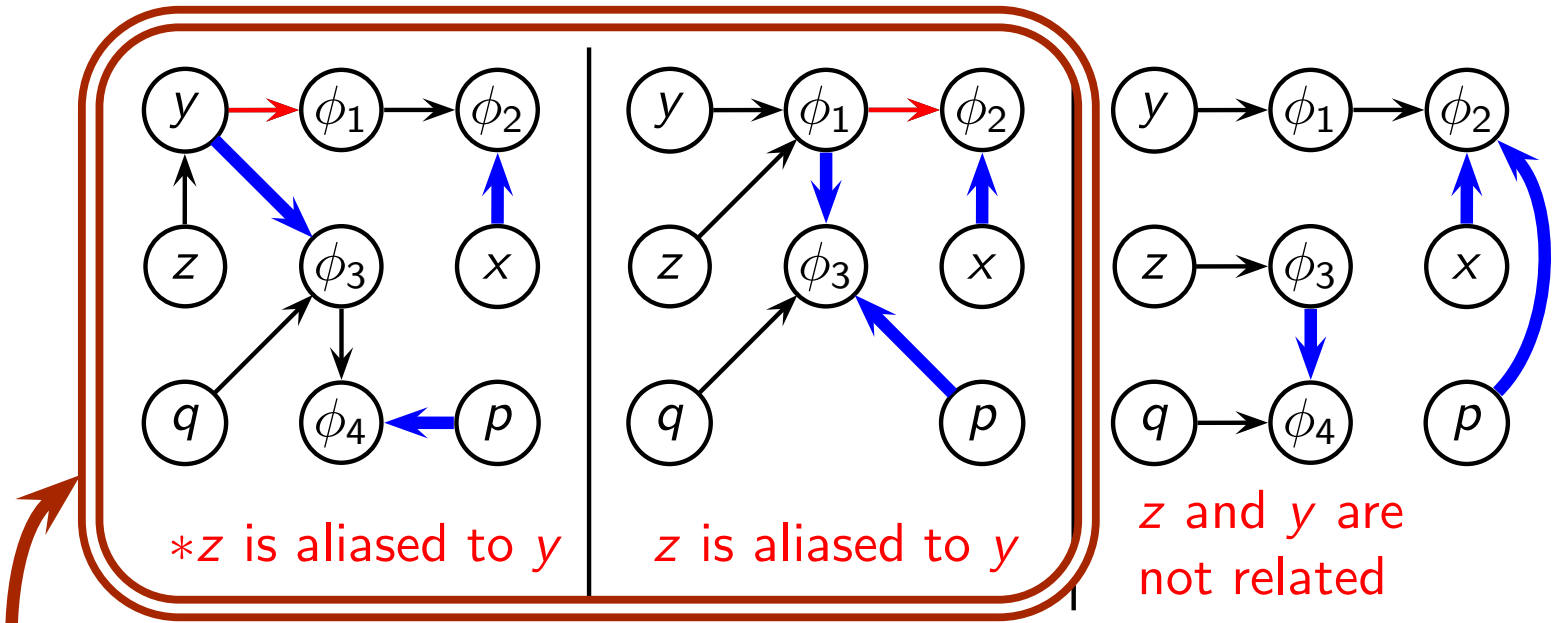
Red edges	$\Rightarrow$	killed information
Blue edges	$\Rightarrow$	information generated
Black edges	$\Rightarrow$	carried forward input information

Context Based Bottom-up Approach

The need of multiple partial transfer functions (PTFs)

Example:

1. $x = *y;$
2. $*z = q;$
3. $p = *y;$



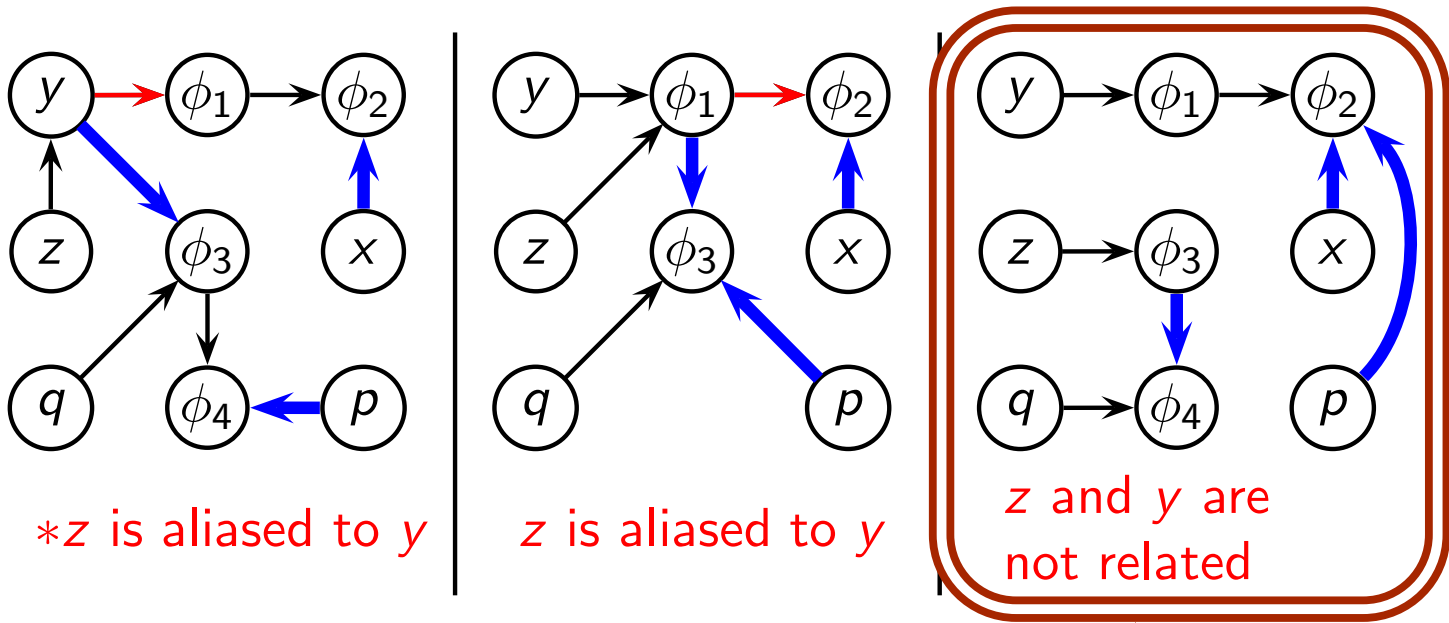
Statement 2 will cause a side effect and p will point to what is related to q and not what is related to x

Context Based Bottom-up Approach

The need of multiple partial transfer functions (PTFs)

Example:

1. $x = *y;$
2. $*z = q;$
3. $p = *y;$



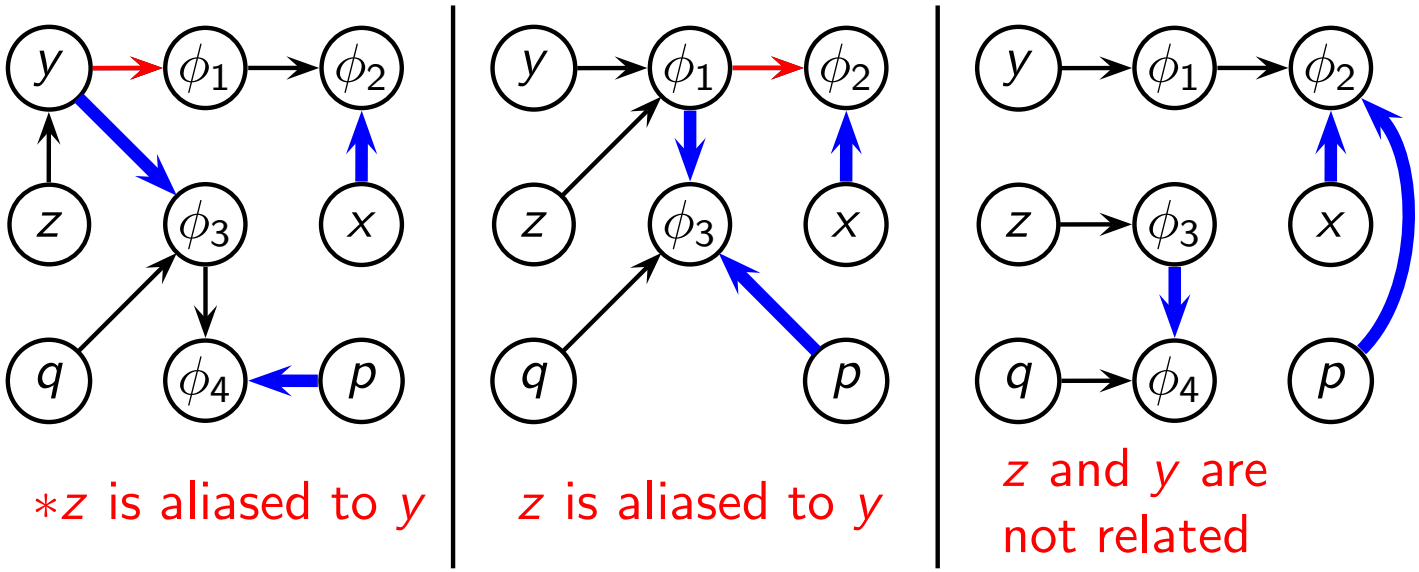
Statement 2 will NOT cause a side effect and p will point to what is related to x and not what is related to q

Context Based Bottom-up Approach

The need of multiple partial transfer functions (PTFs)

Example:

1. $x = *y;$
2. $*z = q;$
3. $p = *y;$

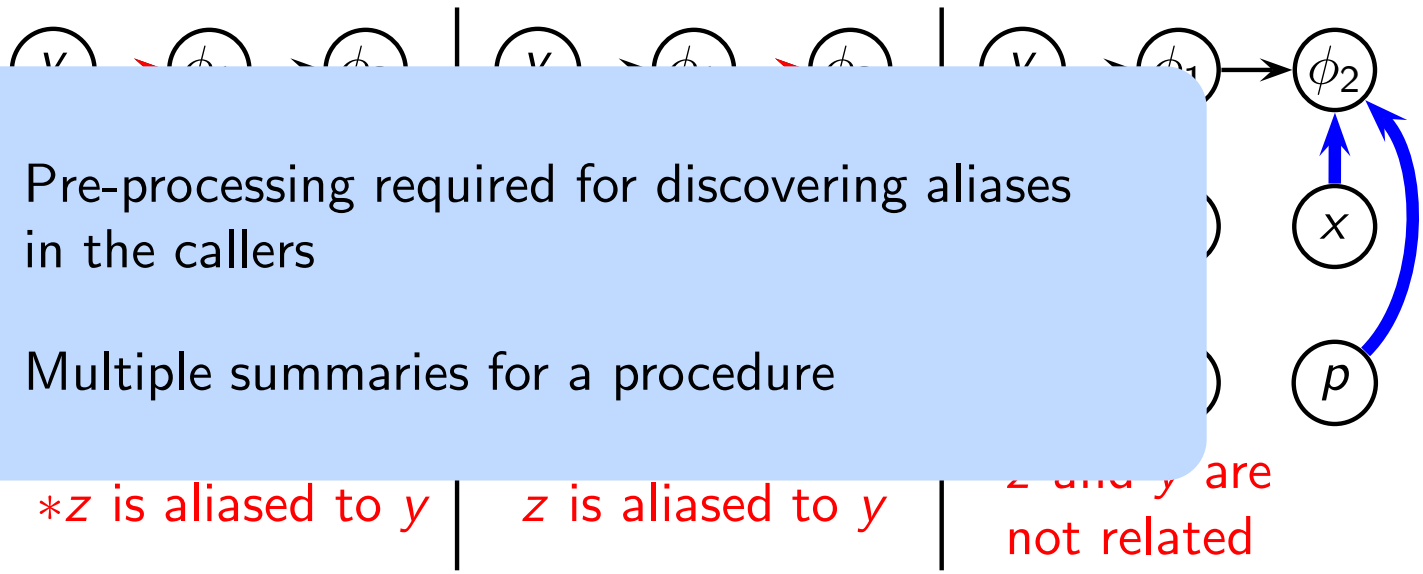


Alias information eliminates data dependency, hence no control flow required

Only relevant aliases are considered

Context Based Bottom-up Approach

The need of multiple partial transfer functions (PTFs)



Examp

1. $x =$
2. $*z =$
3. $p =$

- Pre-processing required for discovering aliases in the callers
- Multiple summaries for a procedure

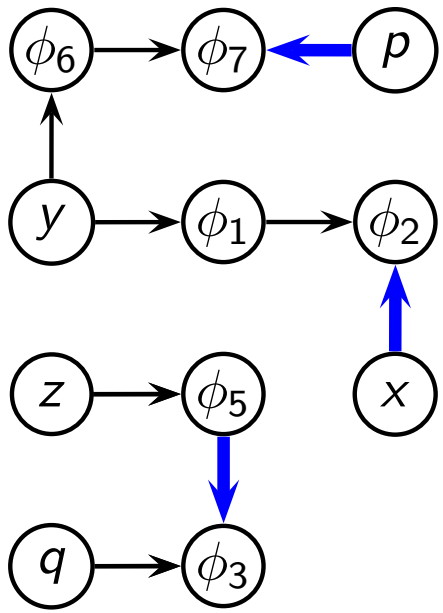
Only relevant aliases are considered

Context Independent Bottom-up Approach

Construction of a single flow-sensitive procedure summary

Example:

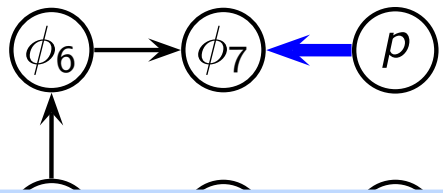
1. $x = *y;$
2. $*z = q;$
3. $p = *y;$



Different accesses of the same variable may require different placeholders

Context Independent Bottom-up Approach

Construction of a single flow-sensitive procedure summary



- Large number of placeholders
  $\Rightarrow$ size of procedure summary may be proportional to the # of statements
- A flow-insensitive version may require fewer placeholders $\Rightarrow$ affects precision

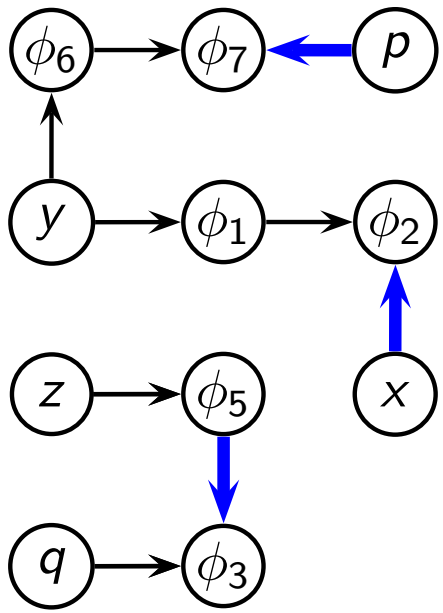
Different accesses of the same variable may require different placeholders

Context Independent Bottom-up Approach

Construction of a single flow-sensitive procedure summary

Example:

1. $x = *y;$
2. $*z = q;$
3. $p = *y;$



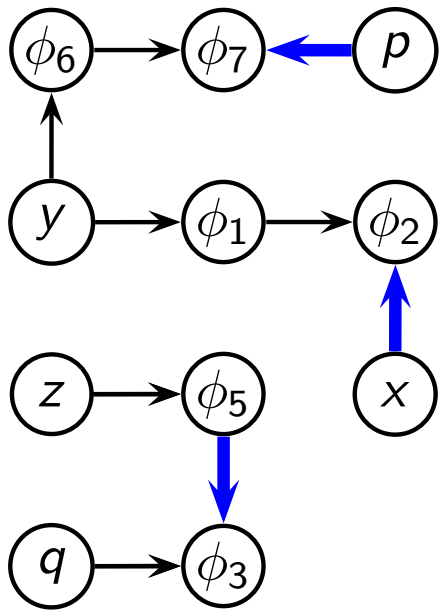
Ordering of generated edges is important

Context Independent Bottom-up Approach

Construction of a single flow-sensitive procedure summary

Example:

1. $x = *y;$
2. $*z = q;$
3. $p = *y;$



Ordering of generated edges is important

If $\phi_5 \rightarrow \phi_3$ is considered before $x \rightarrow \phi_2$, it will amount to swapping statements 1 and 2

Hence, x and p will be always be aliased ignoring the possible side-effect of statement 2

Limitation of Placeholders

- The problem with placeholders is that they explicate the pointees defined in the caller

Limitation of Placeholders

- The problem with placeholders is that they explicate the pointees defined in the caller
- This results in
  - either multiple call-specific procedure summaries, or

Reuse of a placeholder for a flow sensitive summary flow function depends on the aliases in the calling contexts

Limitation of Placeholders

- The problem with placeholders is that they explicate the pointees defined in the caller
- This results in
  - either multiple call-specific procedure summaries, or
  - large number of placeholders

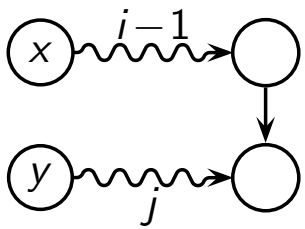
In absence of aliases in the calling context, every access is represented by a separate placeholder. Control flow is also required

Reuse of a placeholder for a flow sensitive summary flow function depends on the aliases in the calling contexts

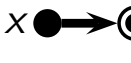
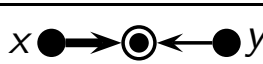
Representing Basic Pointer Assignments using the Generalized Points-to Updates

General Case

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



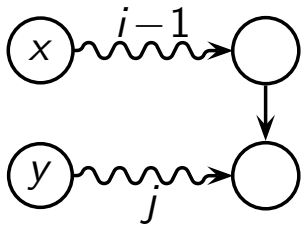
Specific Examples

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

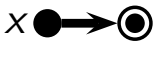
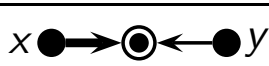
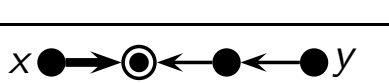
Representing Basic Pointer Assignments using the Generalized Points-to Updates

General Case

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



Specific Examples

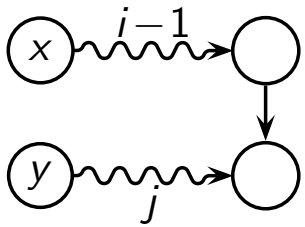
Pointer assignment	GPU	Relevant memory graph after the assignment
$s: x = \&y$	$x \xrightarrow{1\|0}_{s} y$	
$s: x = y$	$x \xrightarrow{1\|1}_{s} y$	
$s: x = *y$	$x \xrightarrow{1\|2}_{s} y$	
$s: *x = y$	$x \xrightarrow{2\|1}_{s} y$	

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

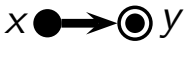
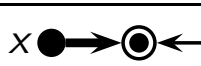
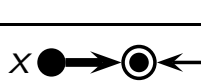
Representing Basic Pointer Assignments using the Generalized Points-to Updates

General Case

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



Specific Examples

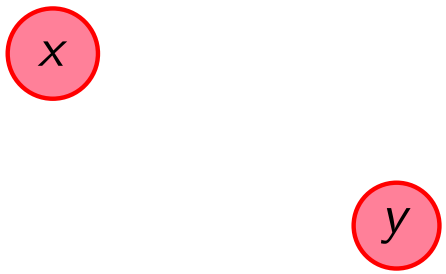
Pointer assignment	GPU	Relevant memory graph after the assignment
$s: x = \&y$	$x \xrightarrow{1\|0}_{s} y$	
$s: x = y$	$x \xrightarrow{1\|1}_{s} y$	
$s: x = *y$	$x \xrightarrow{1\|2}_{s} y$	
$s: *x = y$	$x \xrightarrow{2\|1}_{s} y$	

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

GPU represents both memory and memory transformer

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

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

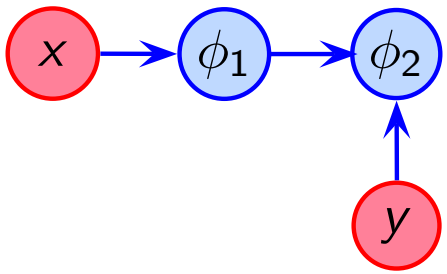


All variables are global

Red nodes are known named locations

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

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



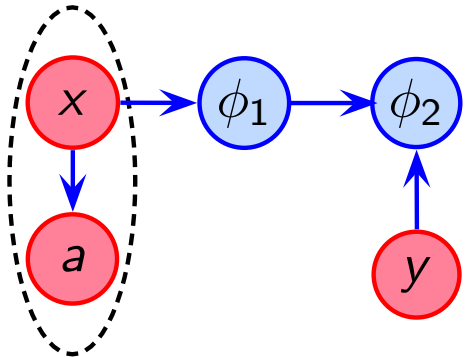
All variables are global

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
}
```



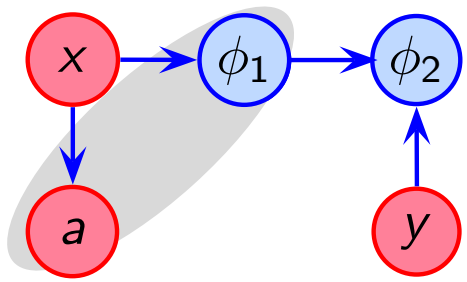
All variables are global

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
}
```



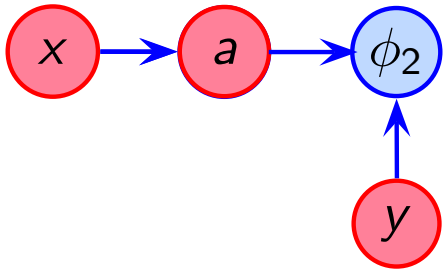
All variables are global

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
}
```



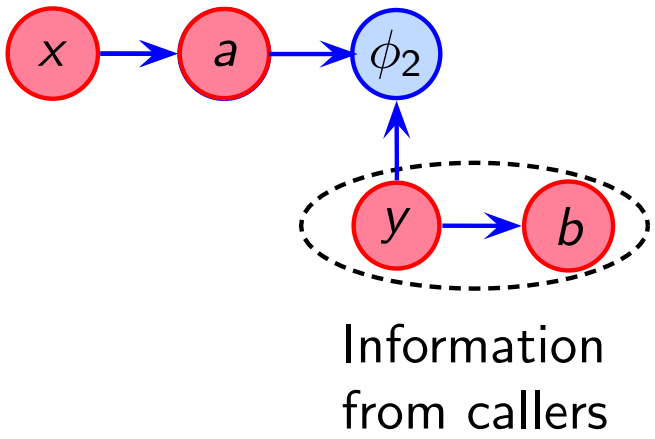
All variables are global

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
}
```



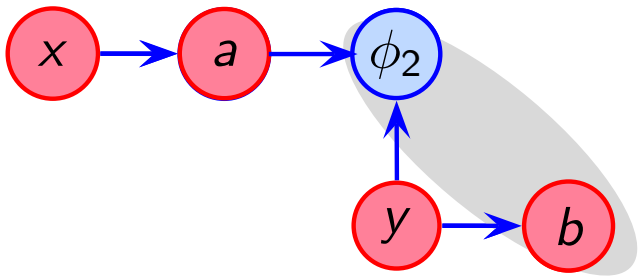
All variables are global

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
}
```



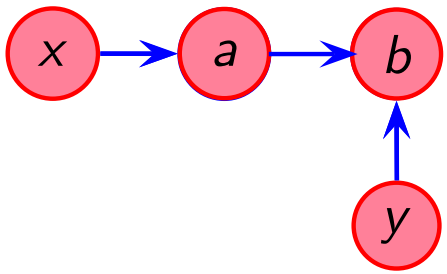
All variables are global

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
}
```



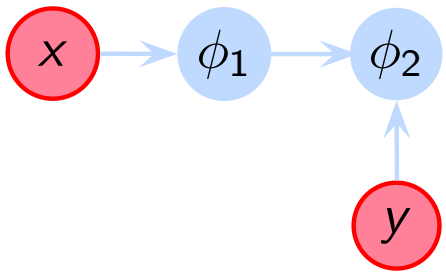
All variables are global

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

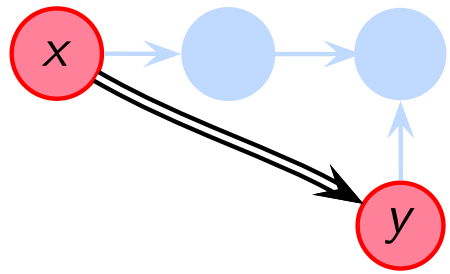
```
f()
{
   *x = y
}
```



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

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

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

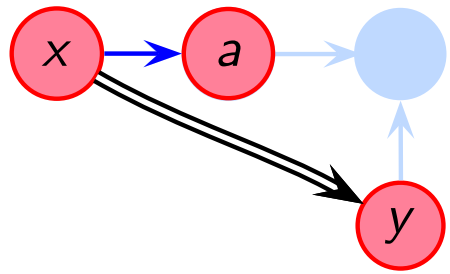


Blue arrows are low level view of memory in terms of classical points-to updates

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

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

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

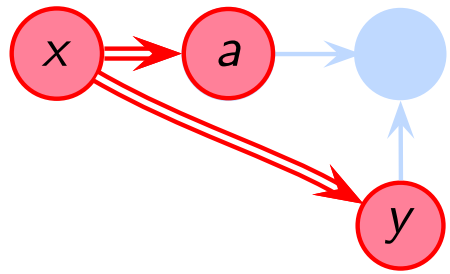


Blue arrows are low level view of memory in terms of classical points-to updates

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

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

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

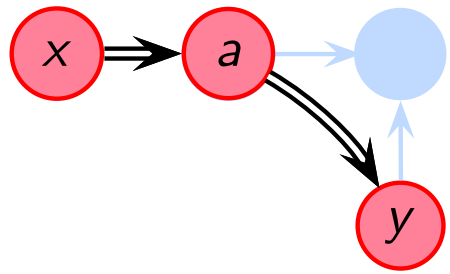


Blue arrows are low level view of memory in terms of classical points-to updates

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

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

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

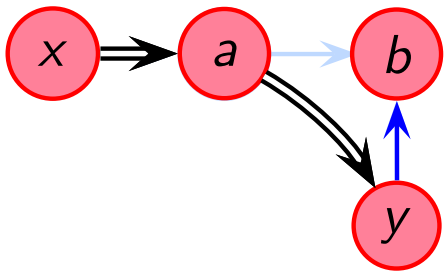


Blue arrows are low level view of memory in terms of classical points-to updates

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

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

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

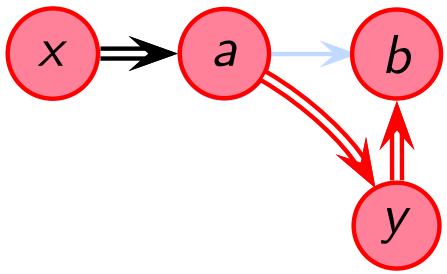


Blue arrows are low level view of memory in terms of classical points-to updates

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

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

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

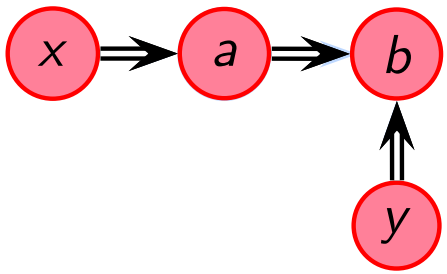


Blue arrows are low level view of memory in terms of classical points-to updates

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

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

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

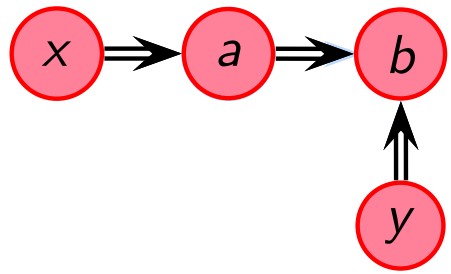


Blue arrows are low level view of memory in terms of classical points-to updates

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

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

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

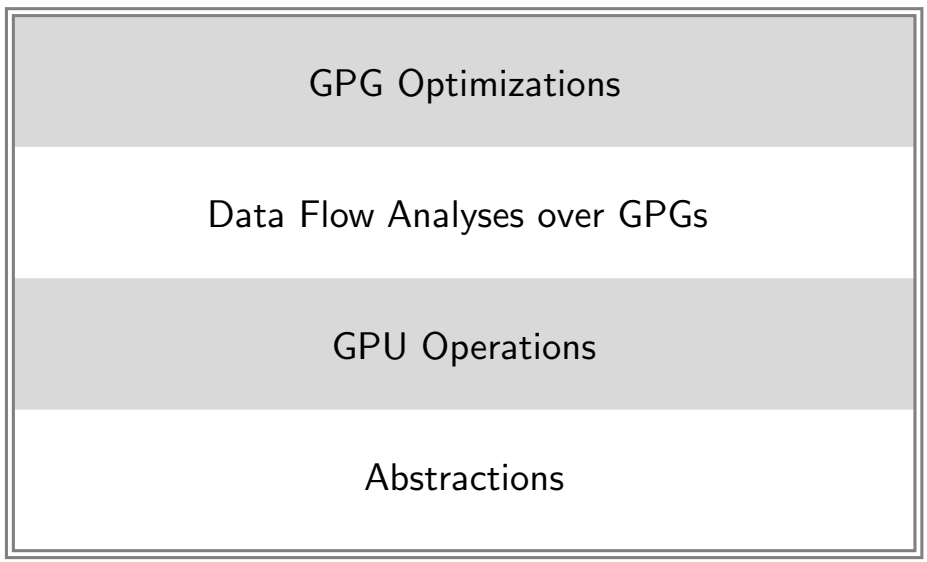


Blue arrows are low level view of memory in terms of classical points-to updates

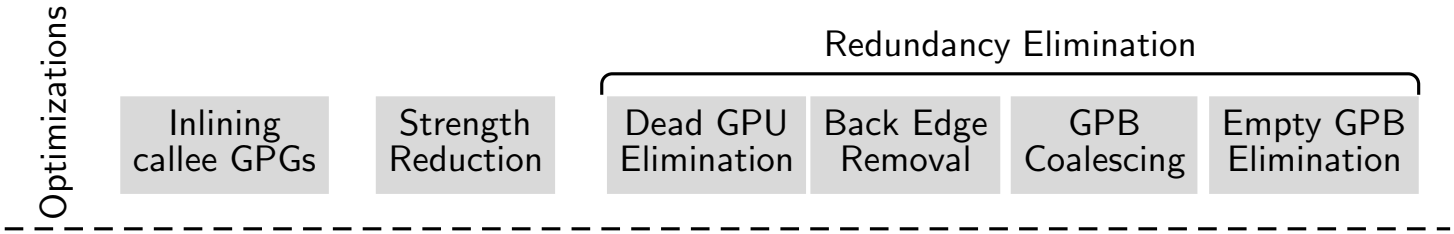
Black arrows are high level view of memory in terms of generalized points-to updates

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

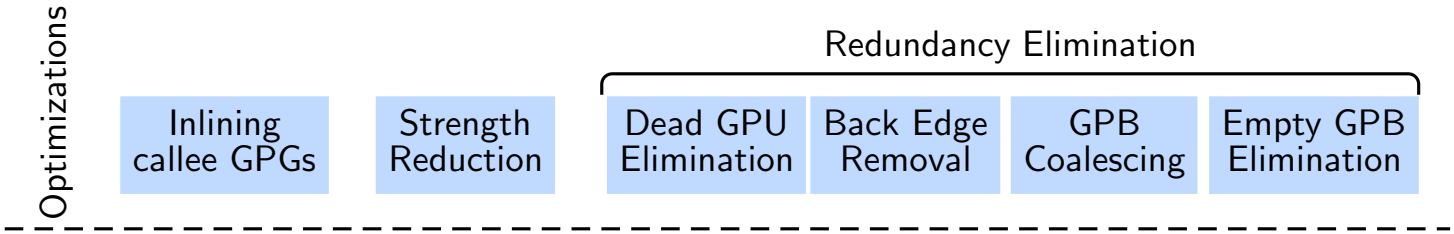
Generalized Points-to Graphs: A Work in Progress



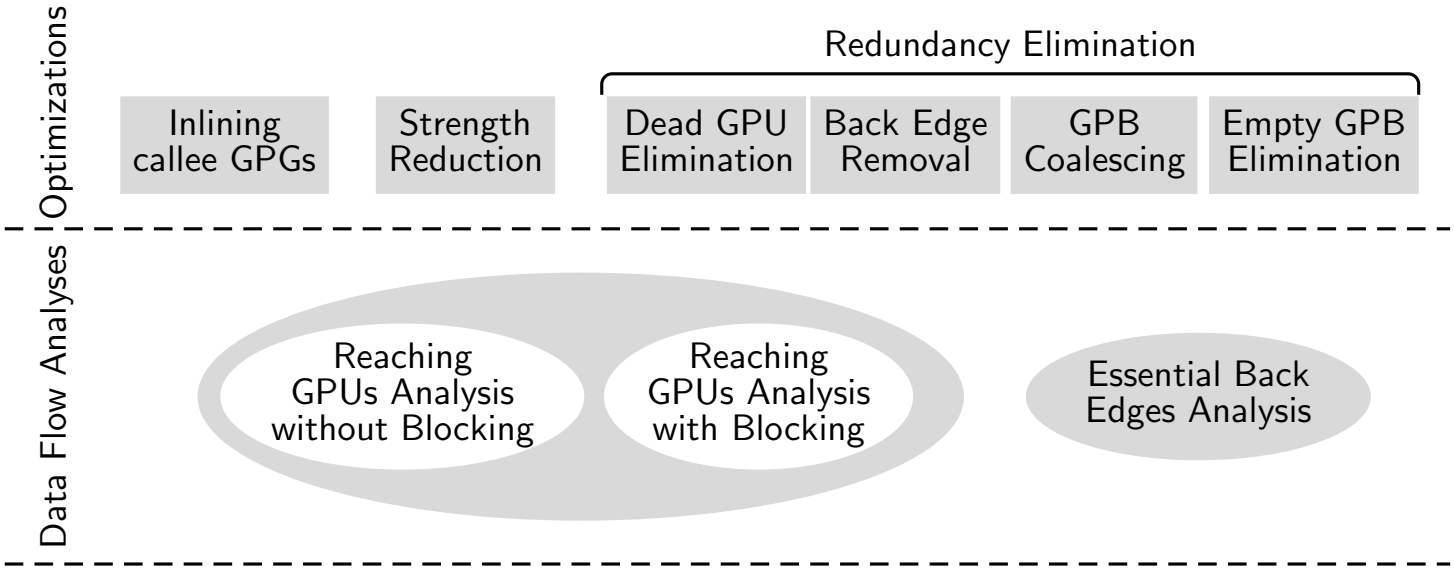
Generalized Points-to Graphs: A Work in Progress



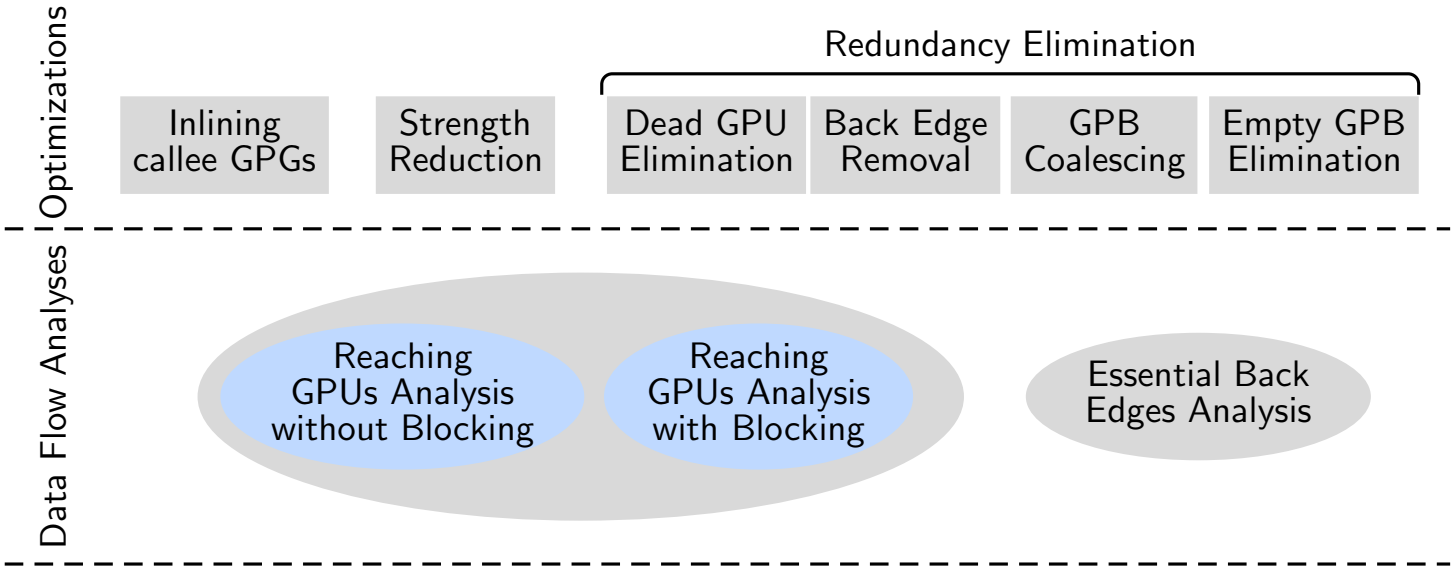
Generalized Points-to Graphs: A Work in Progress



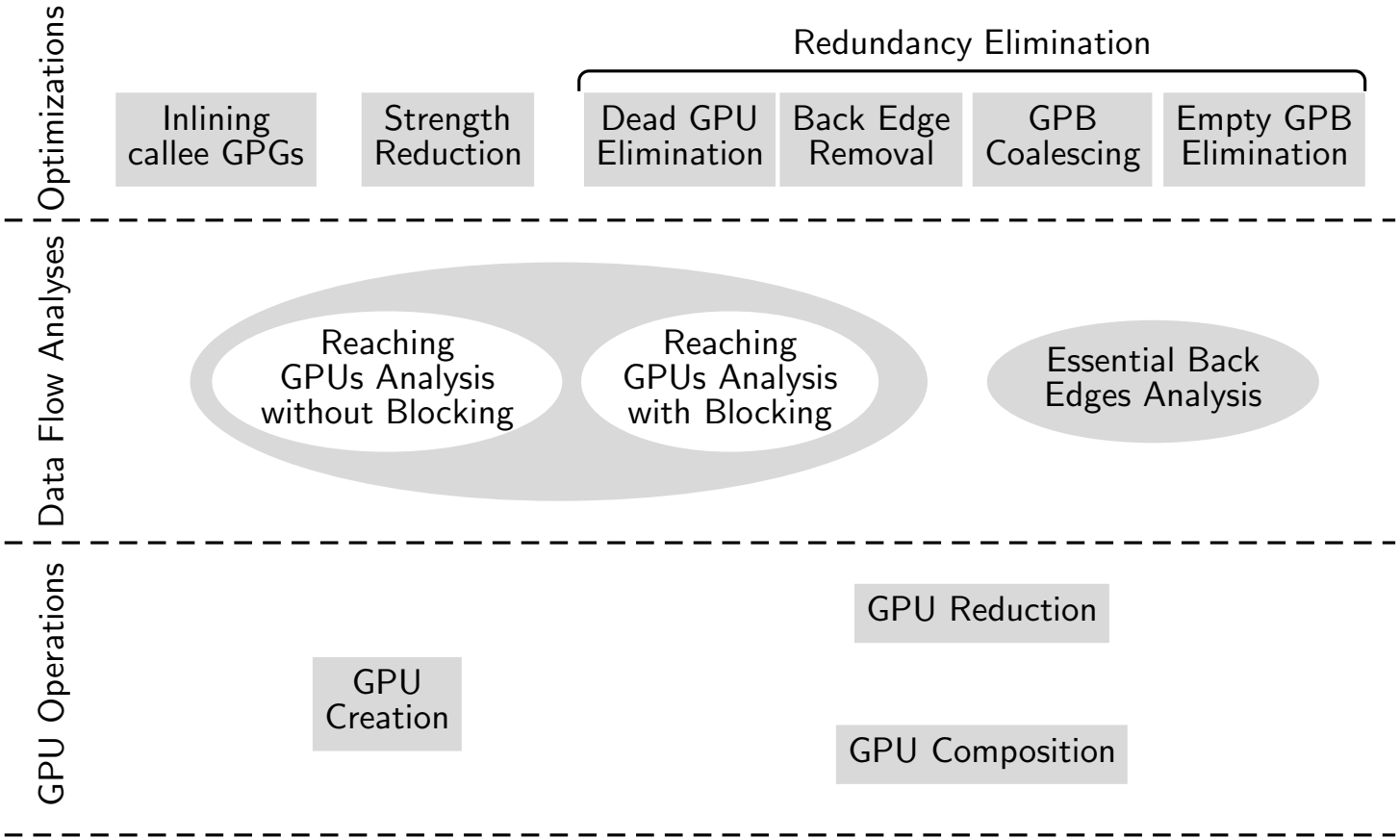
Generalized Points-to Graphs: A Work in Progress



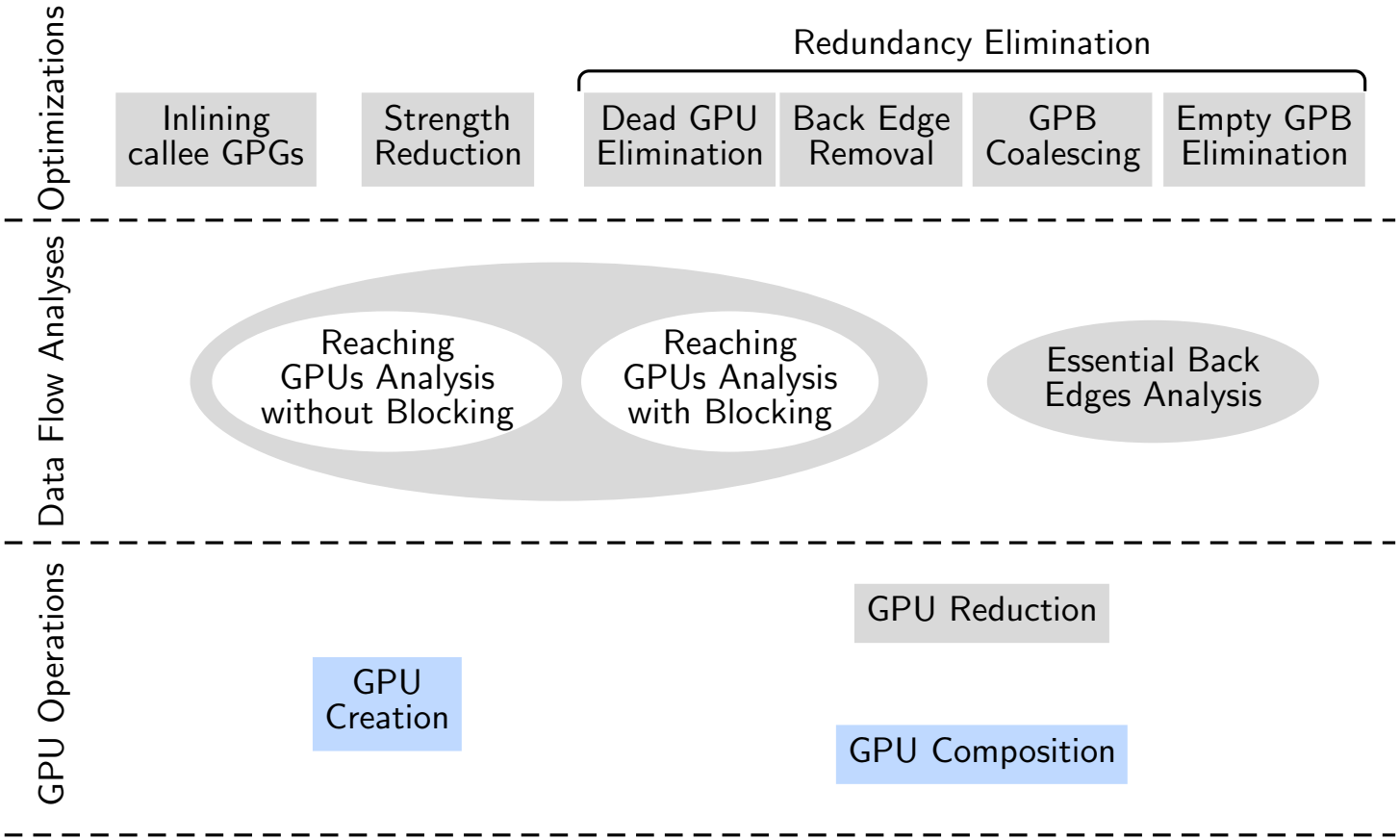
Generalized Points-to Graphs: A Work in Progress



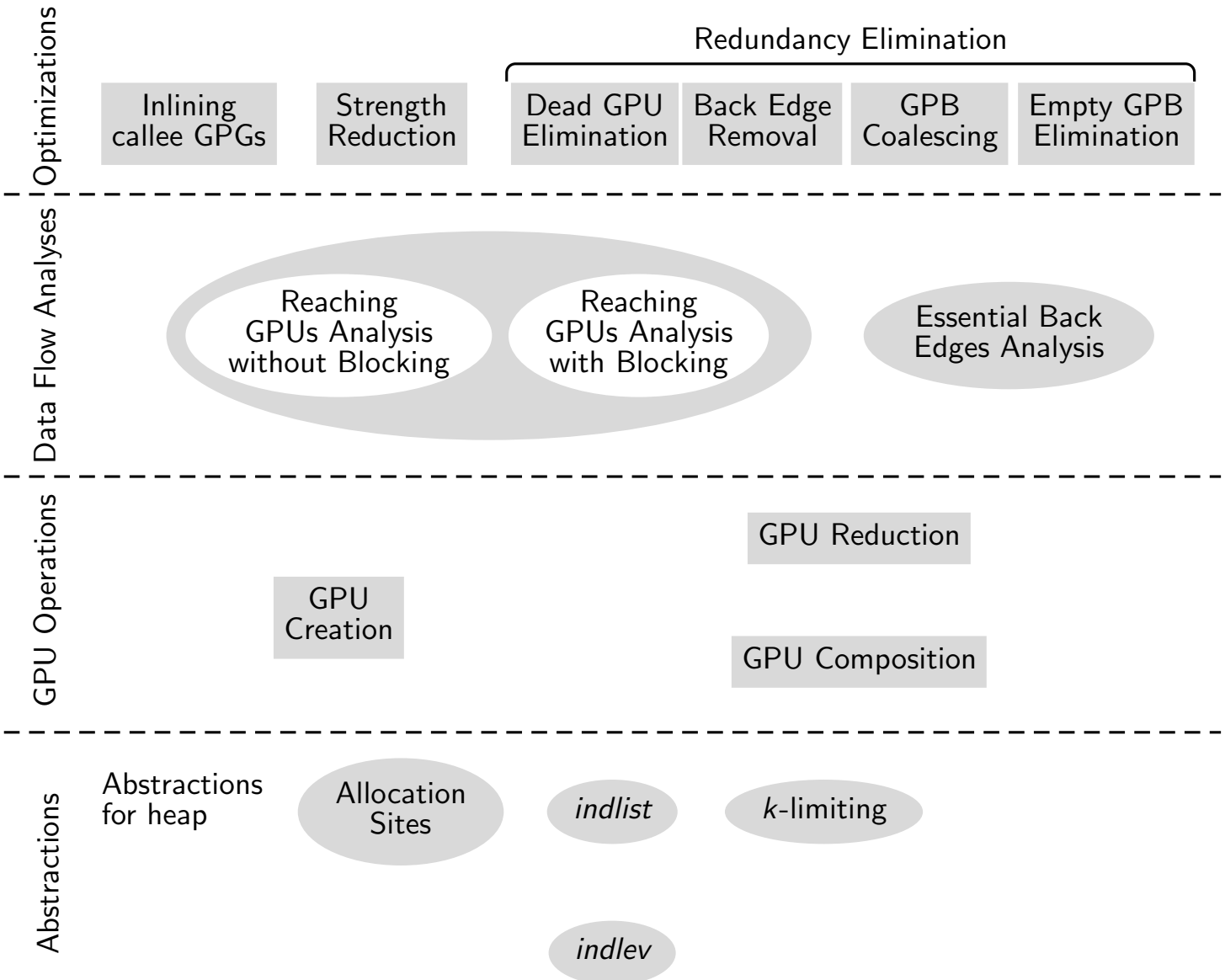
Generalized Points-to Graphs: A Work in Progress



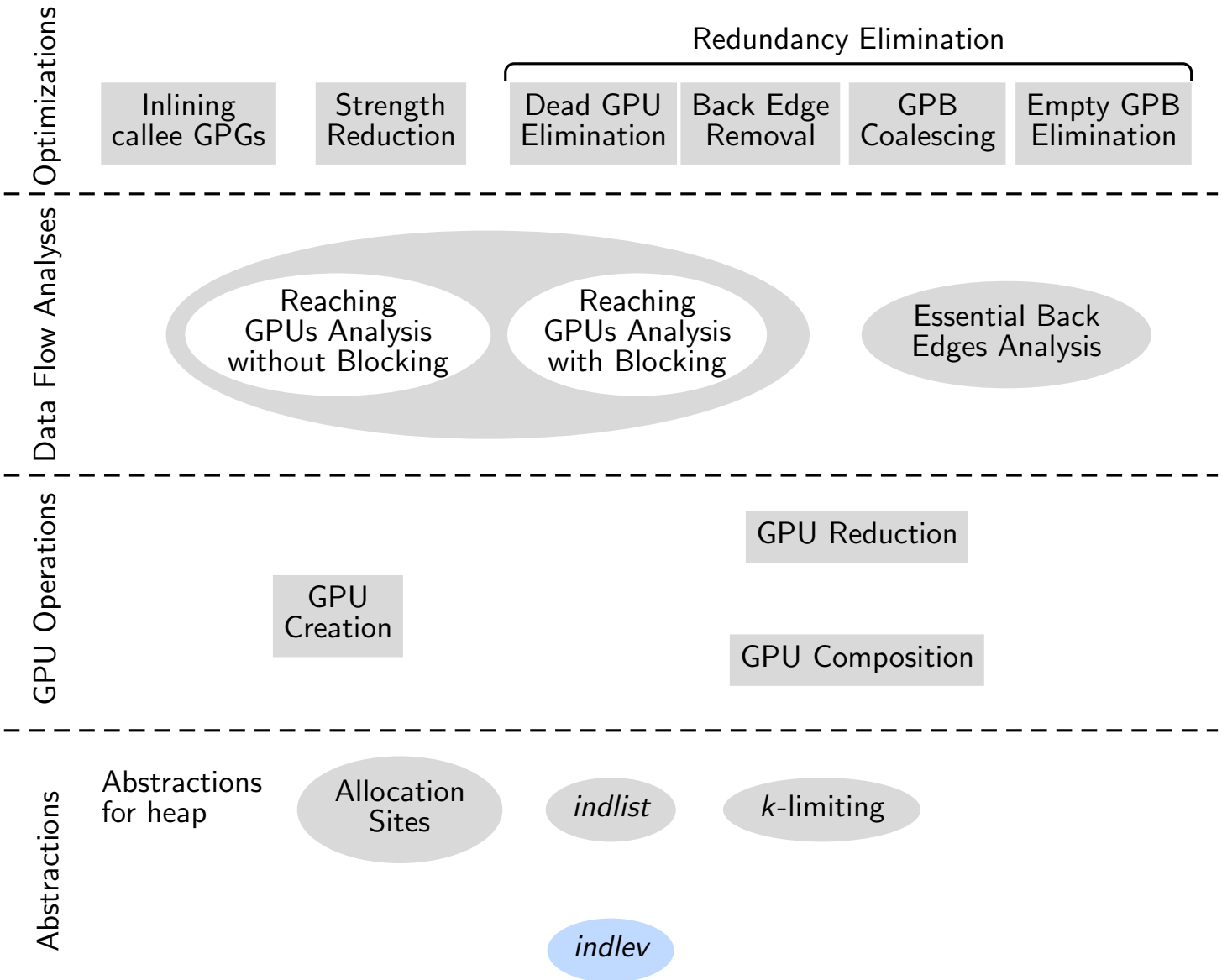
Generalized Points-to Graphs: A Work in Progress



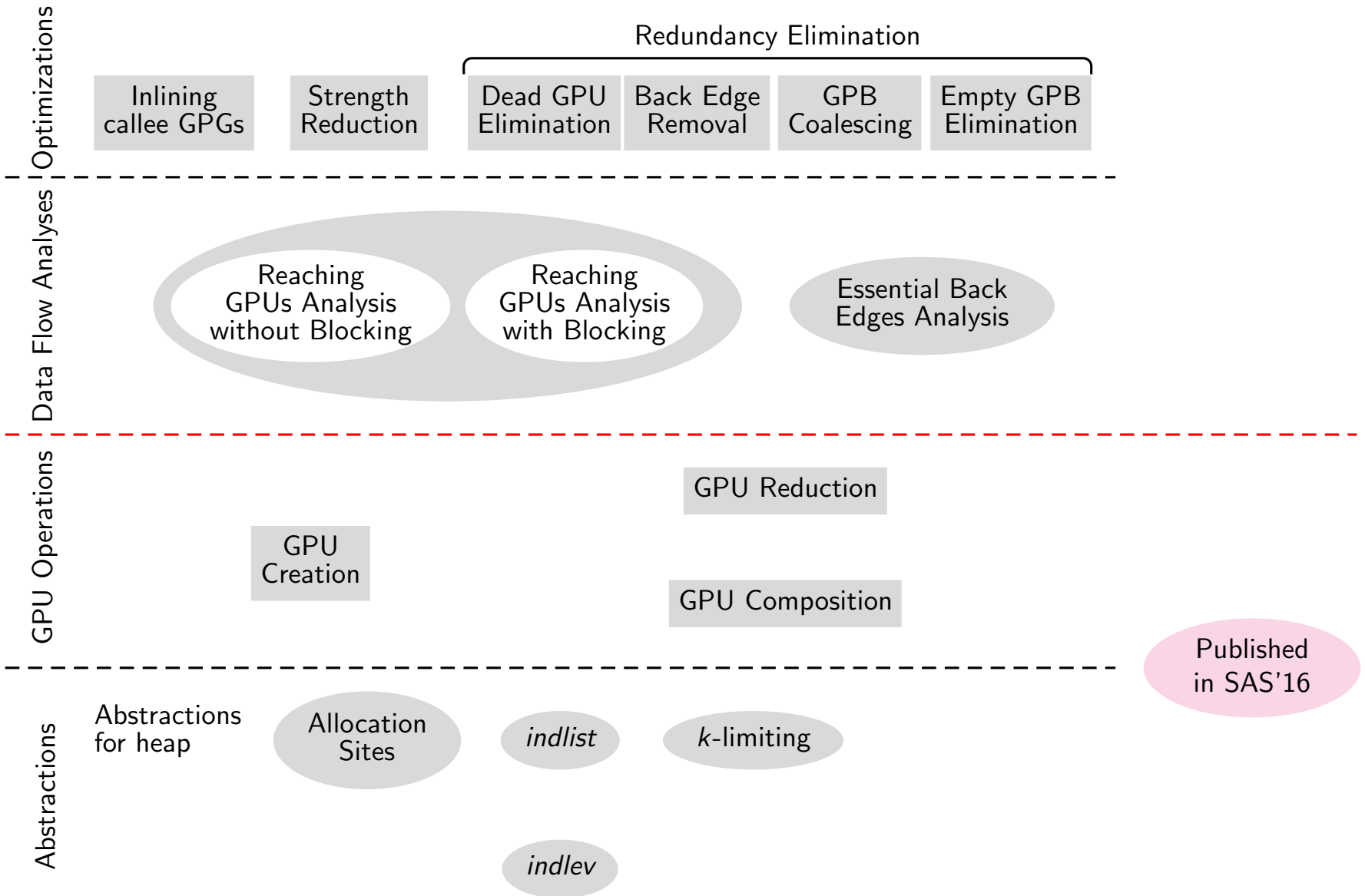
Generalized Points-to Graphs: A Work in Progress



Generalized Points-to Graphs: A Work in Progress



Generalized Points-to Graphs: A Work in Progress



Part 4

Conclusions

Observations

- Data flow propagation in real programs seems to involve a much smaller subset of all possible data flow values

  In large programs that work properly, pointer usage is very disciplined and the core information is very small!

- Earlier approaches reported inefficiency and non-scalability because they computed far more information than required because they

  - did not separate the usable information from unusable information, and
  - used low level abstractions of memory

  Their focus was on

  - approximating information to reduce the size, or
  - storing and accessing the information more efficiently

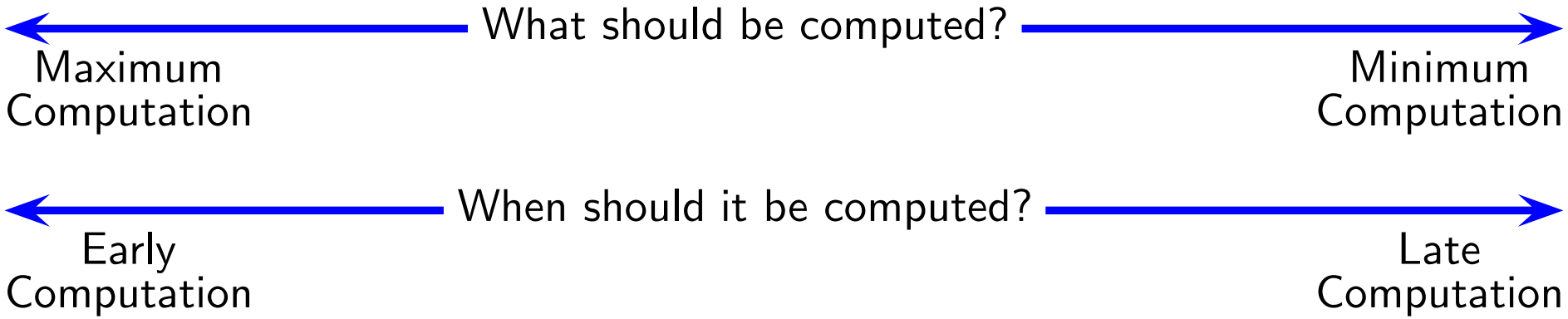
A Spectrum of Possible Ways of Performing Computation

exhaustive computation

computation restricted to usable information

avoiding redundant computation

demand driven computation



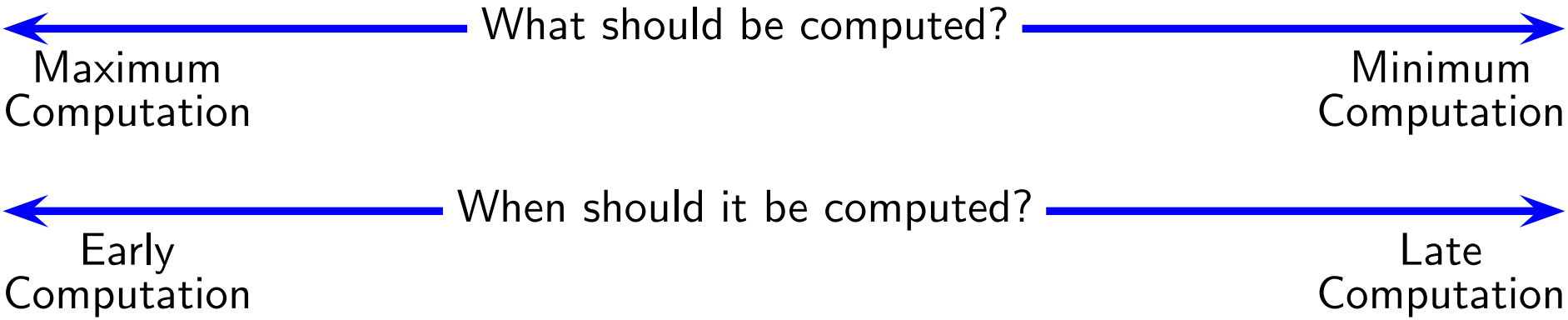
A Spectrum of Possible Ways of Performing Computation

exhaustive computation

computation restricted to usable information

avoiding redundant computation

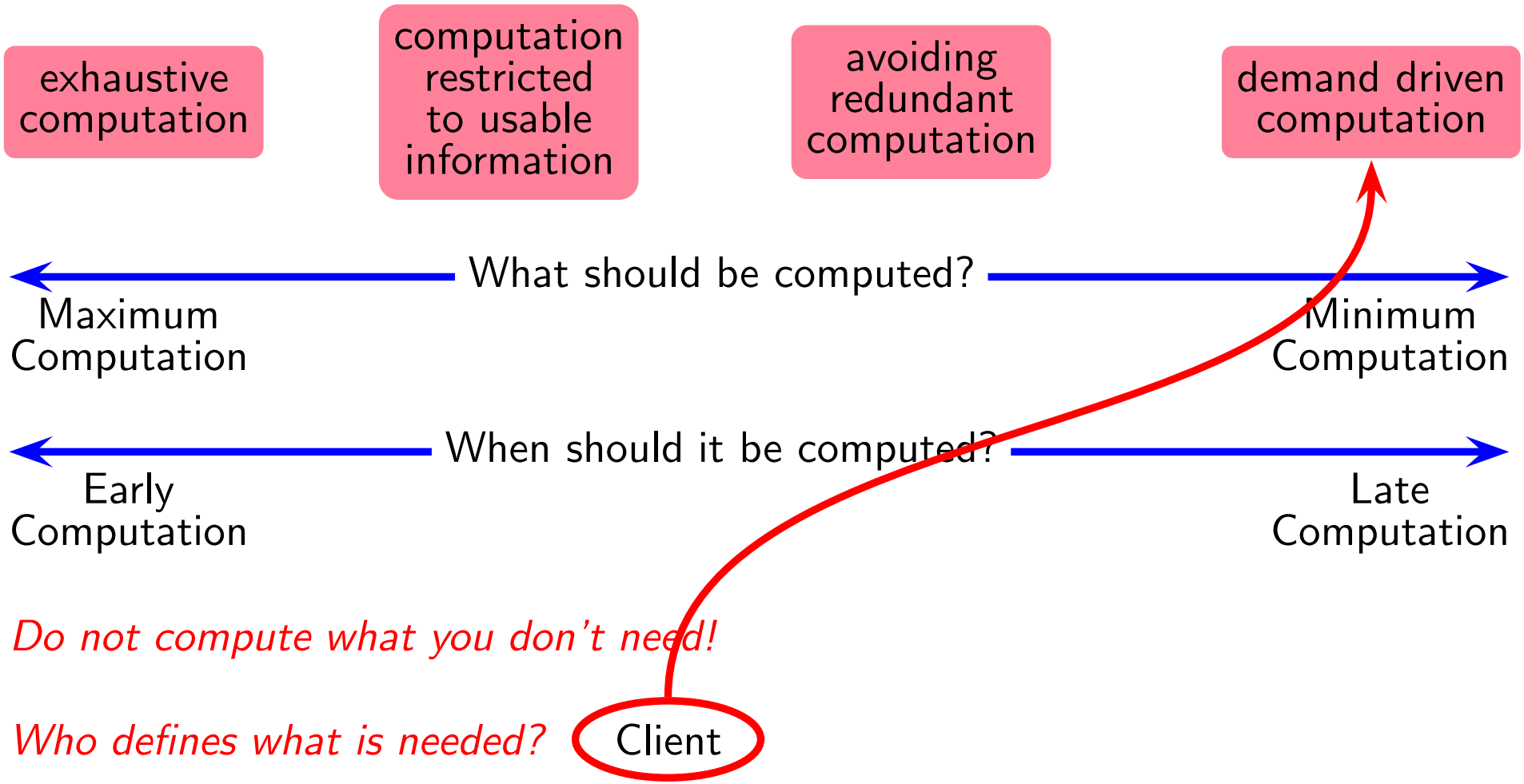
demand driven computation



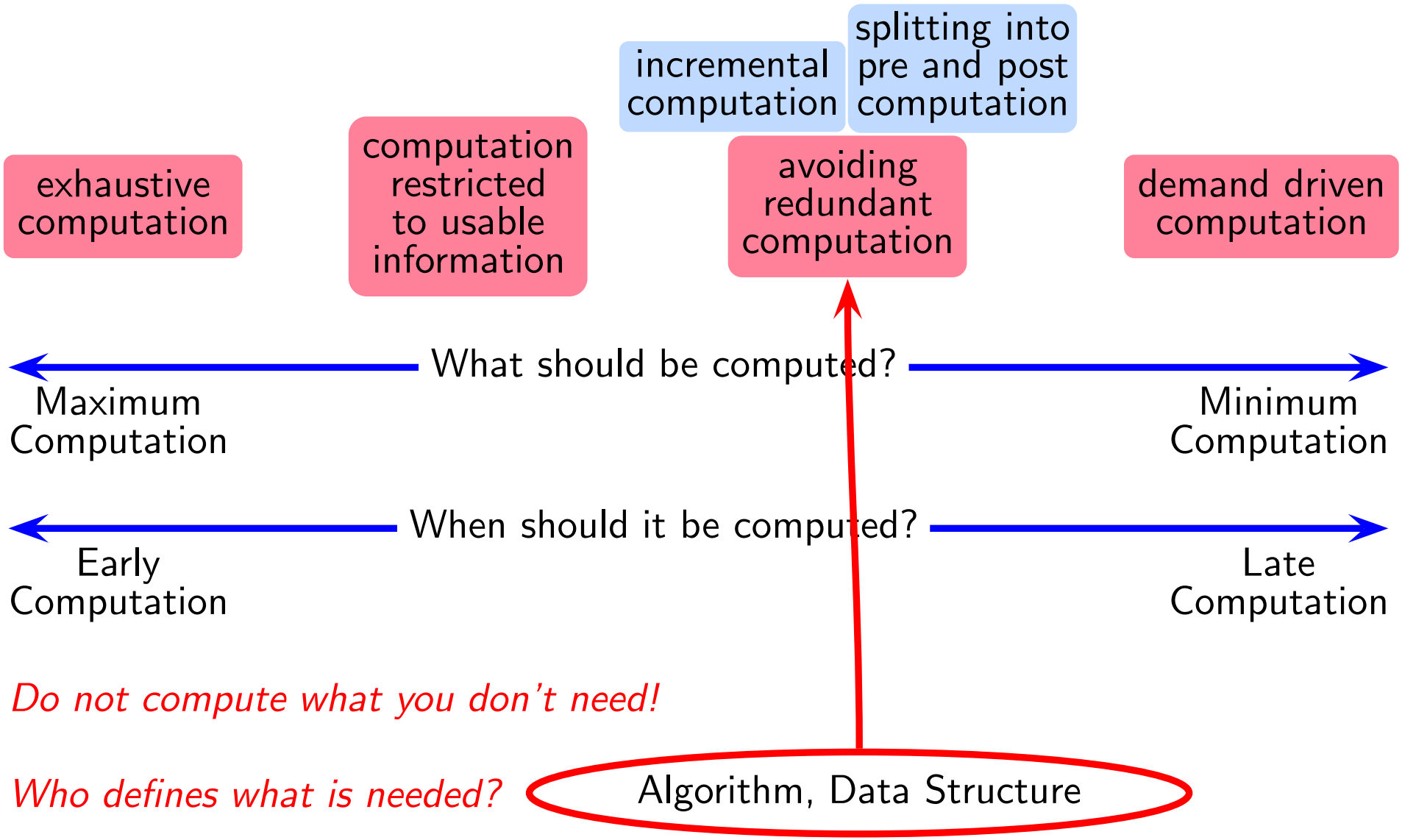
Do not compute what you don't need!

Who defines what is needed?

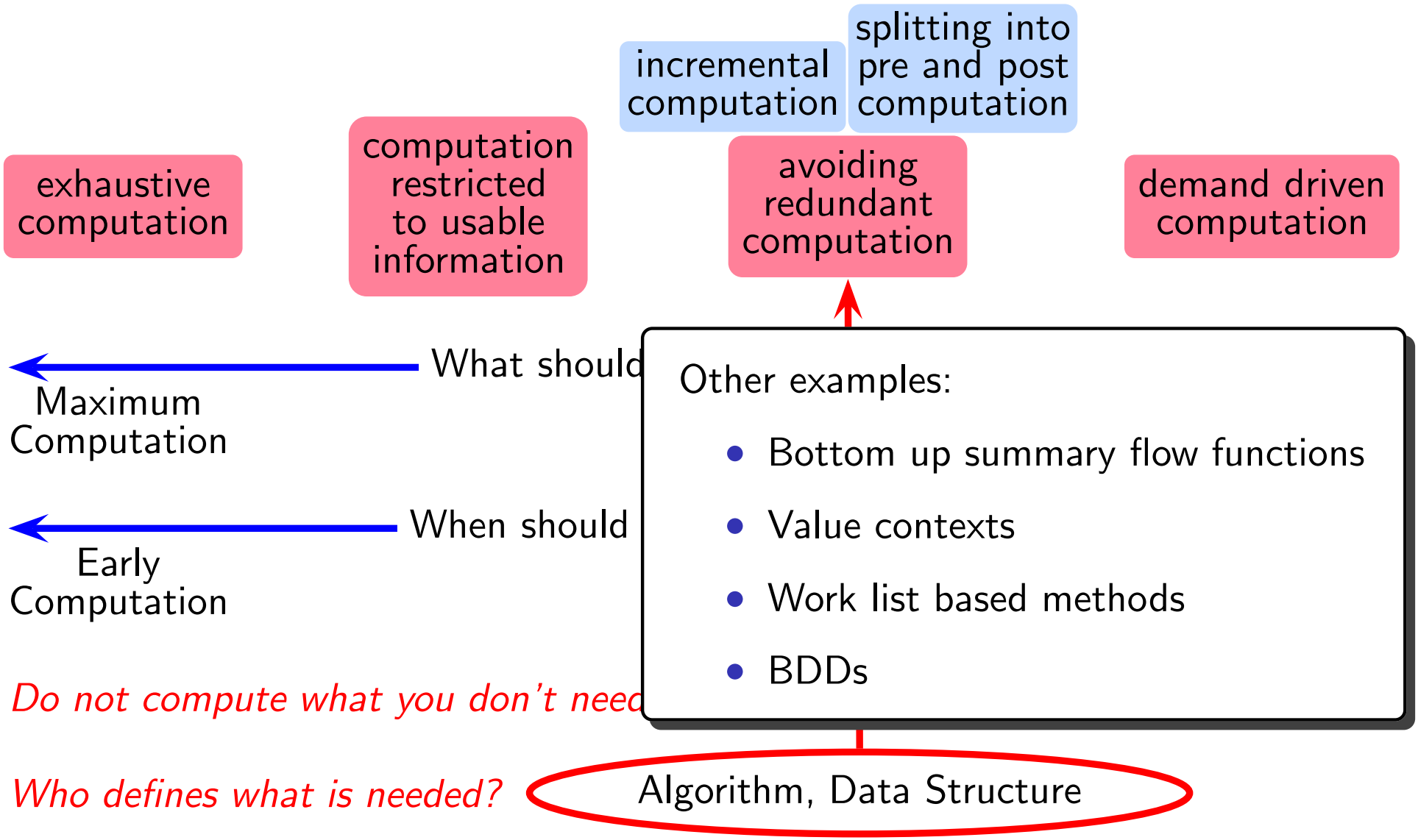
A Spectrum of Possible Ways of Performing Computation



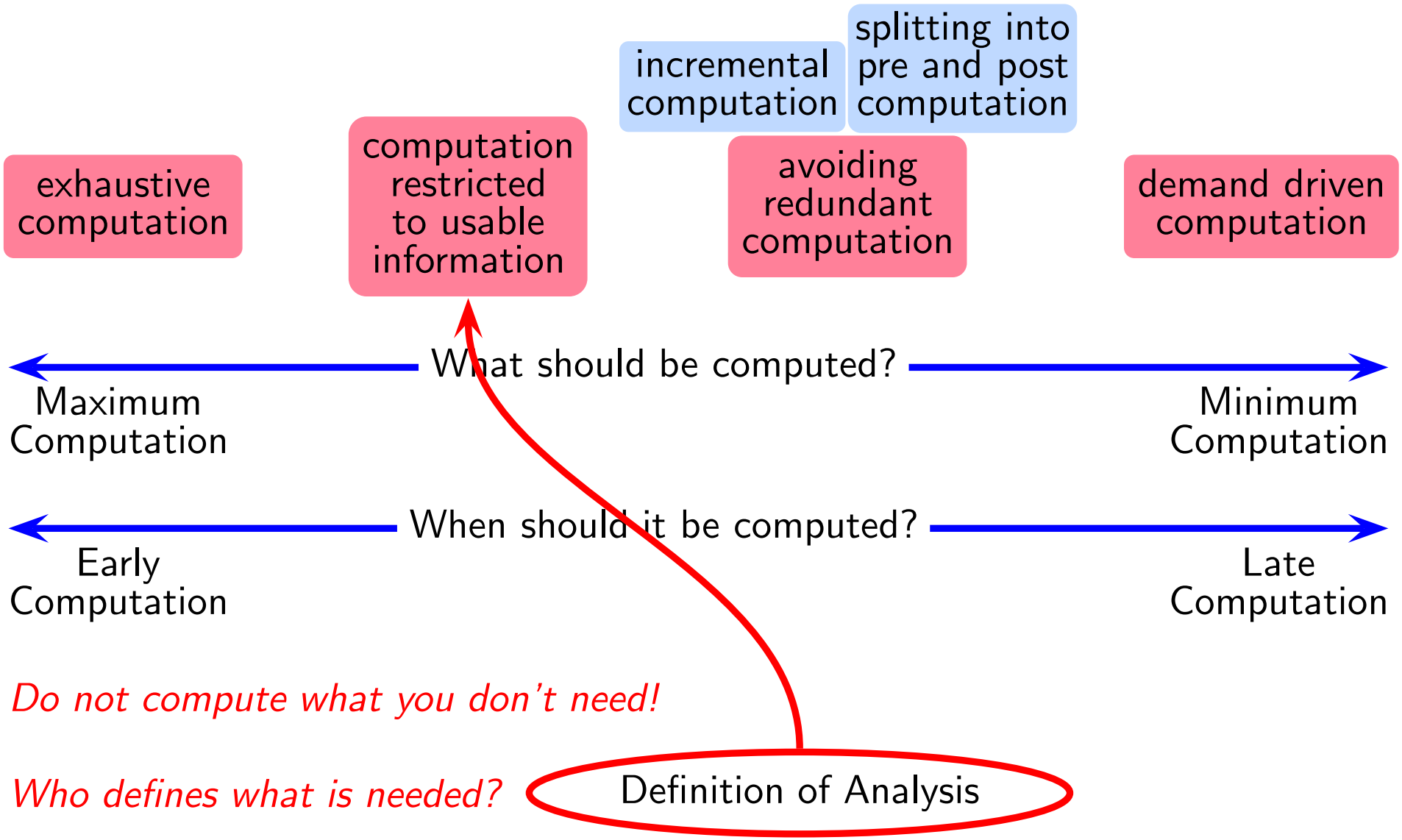
A Spectrum of Possible Ways of Performing Computation



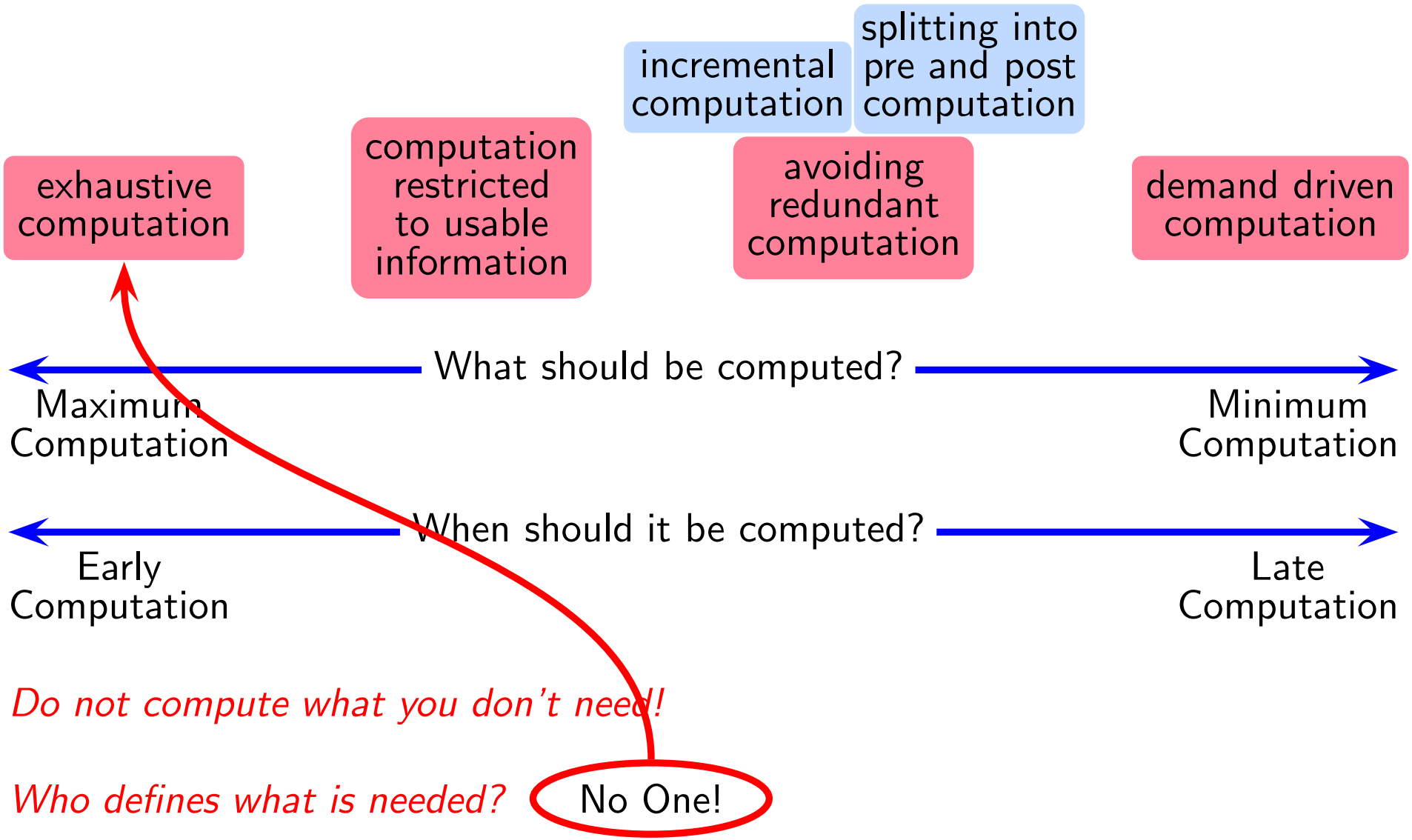
A Spectrum of Possible Ways of Performing Computation



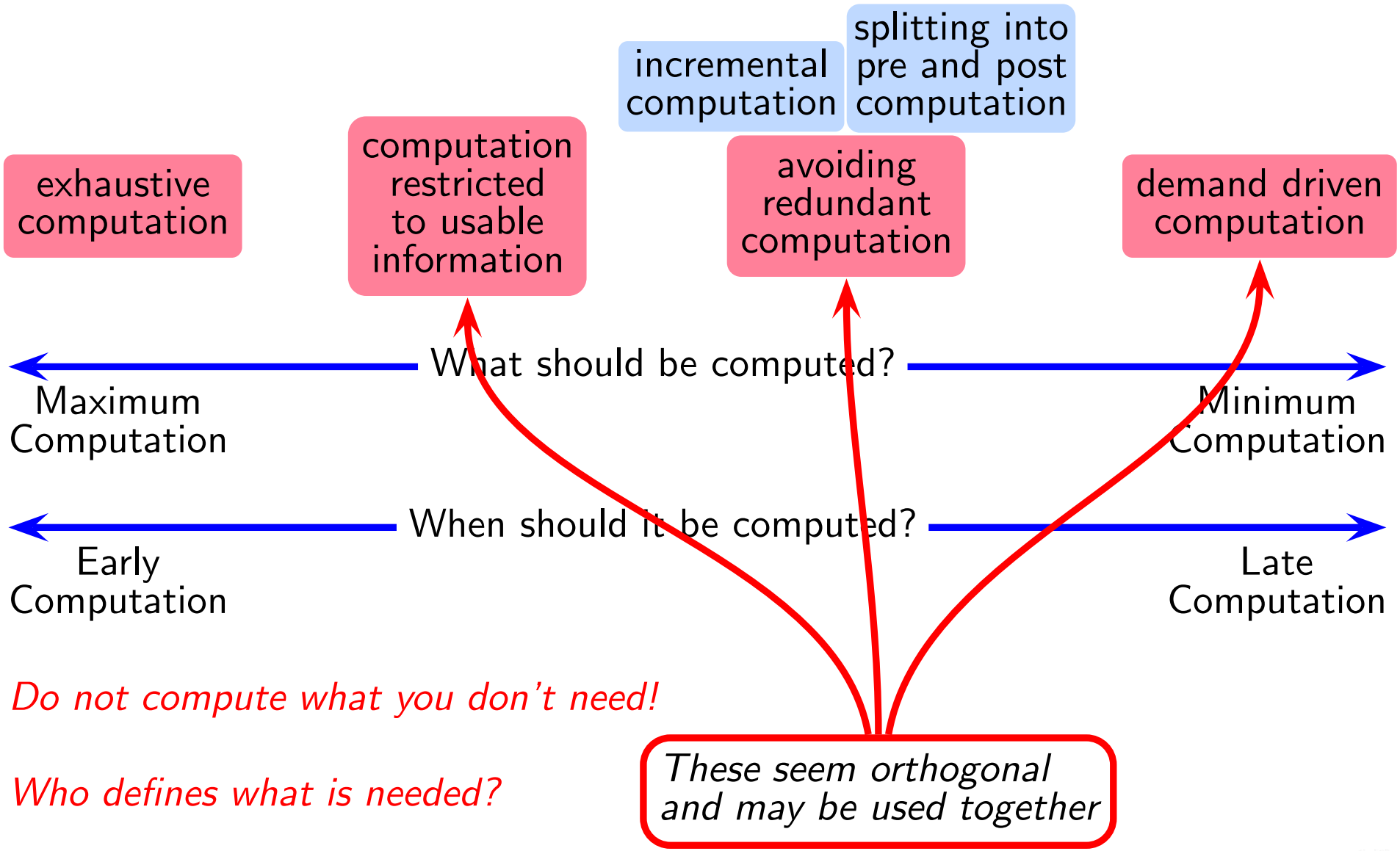
A Spectrum of Possible Ways of Performing Computation



A Spectrum of Possible Ways of Performing Computation



A Spectrum of Possible Ways of Performing Computation



The Road Ahead

- And yet, this is not sufficient to scale points-to analysis
- 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

- And yet, this is not sufficient to scale points-to analysis
- 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

- And yet, this is not sufficient to scale points-to analysis
- 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

Missing control flow paths

Unsound

Original Control Flow

Sound and Precise

Over-approximated Control Flow

Flow insensitivity
Context insensitivity

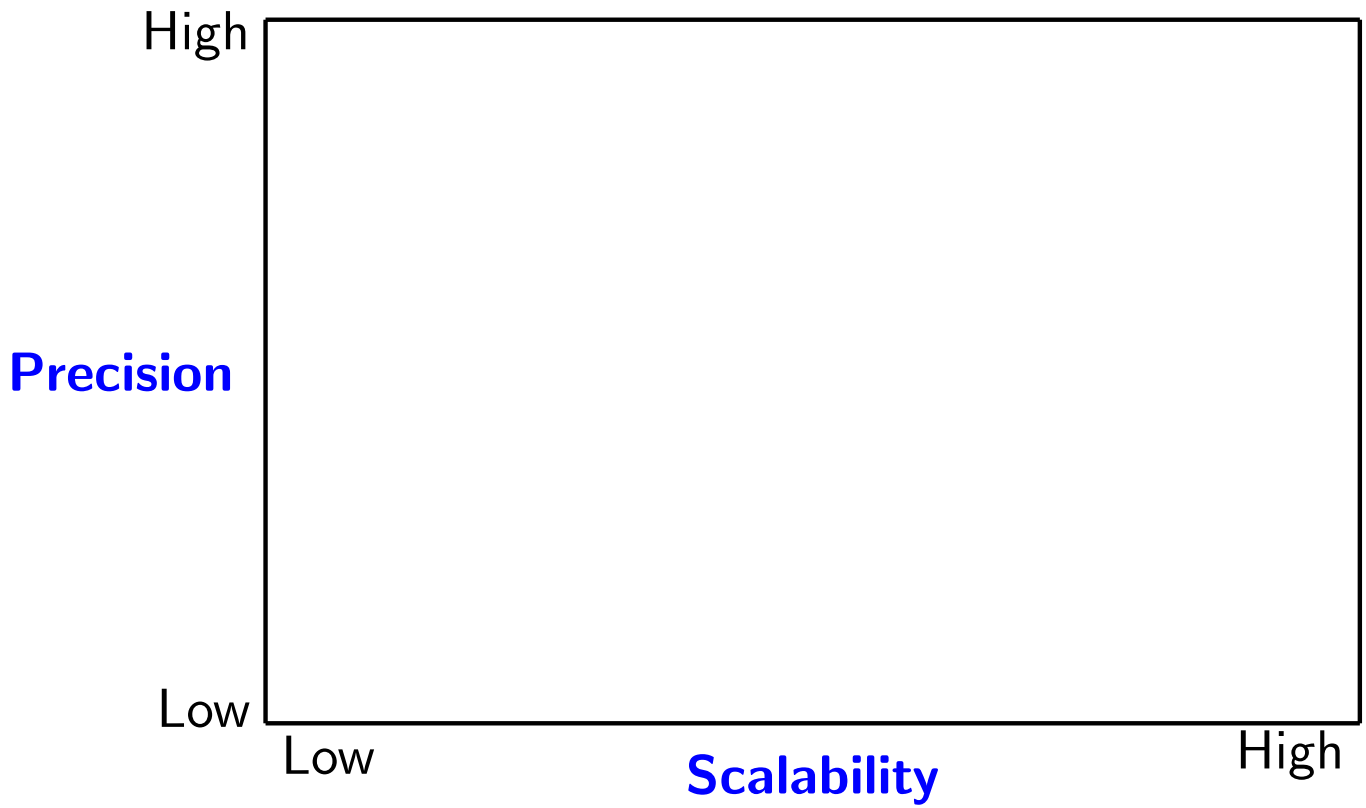
Sound and Imprecise

The Next Holy Grail in Search of Scalability?

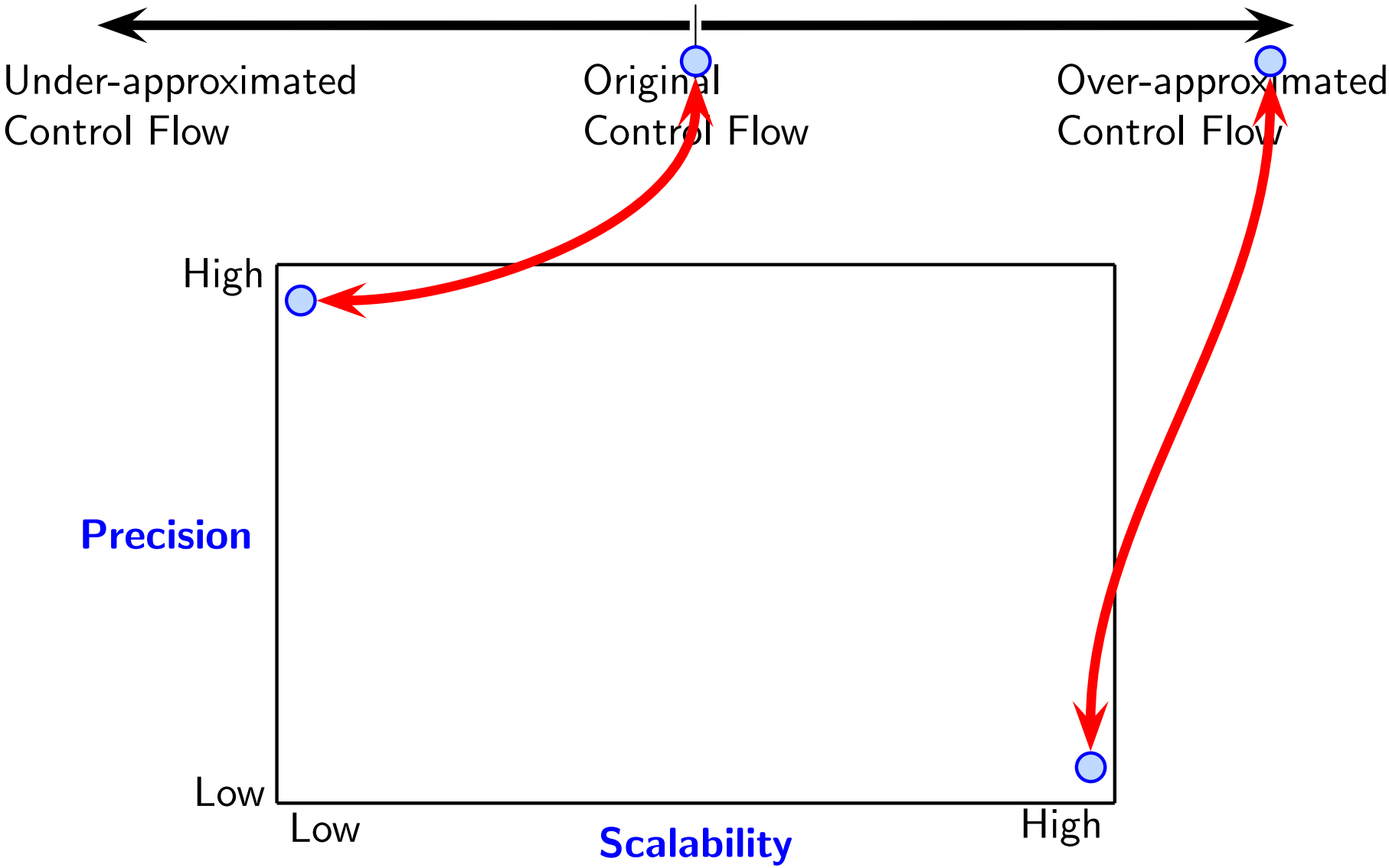
Under-approximated Control Flow

Original Control Flow

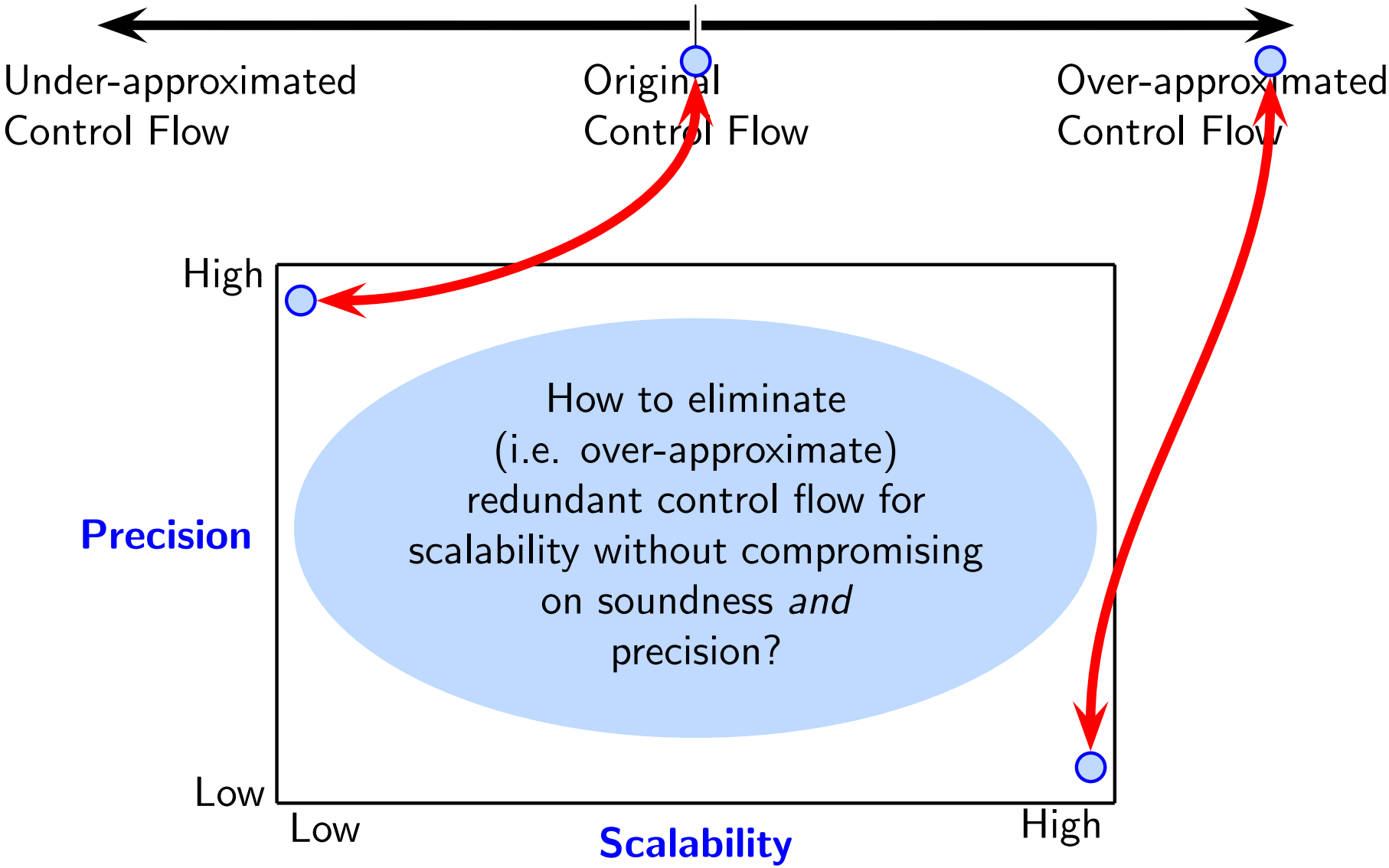
Over-approximated Control Flow



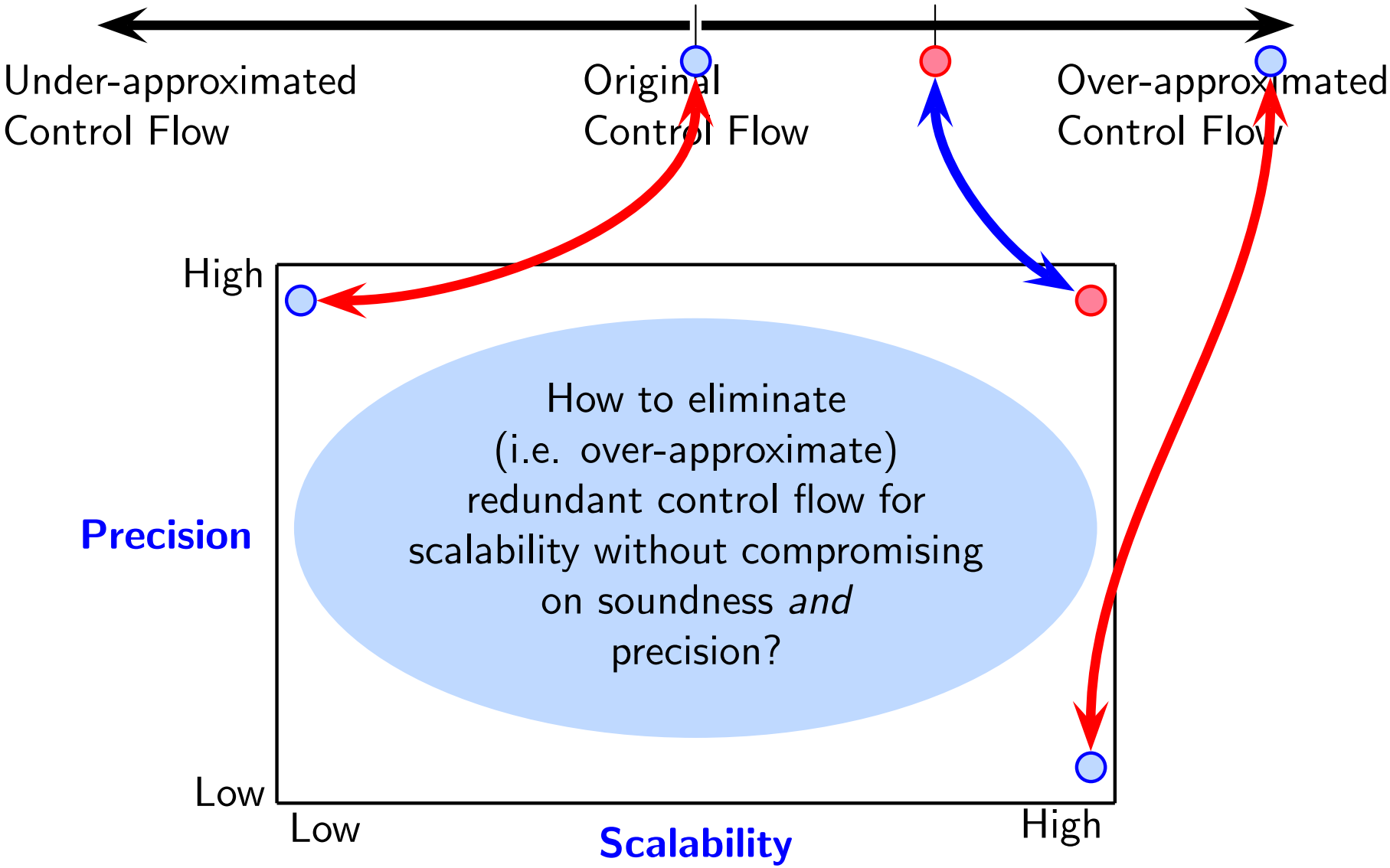
The Next Holy Grail in Search of Scalability?



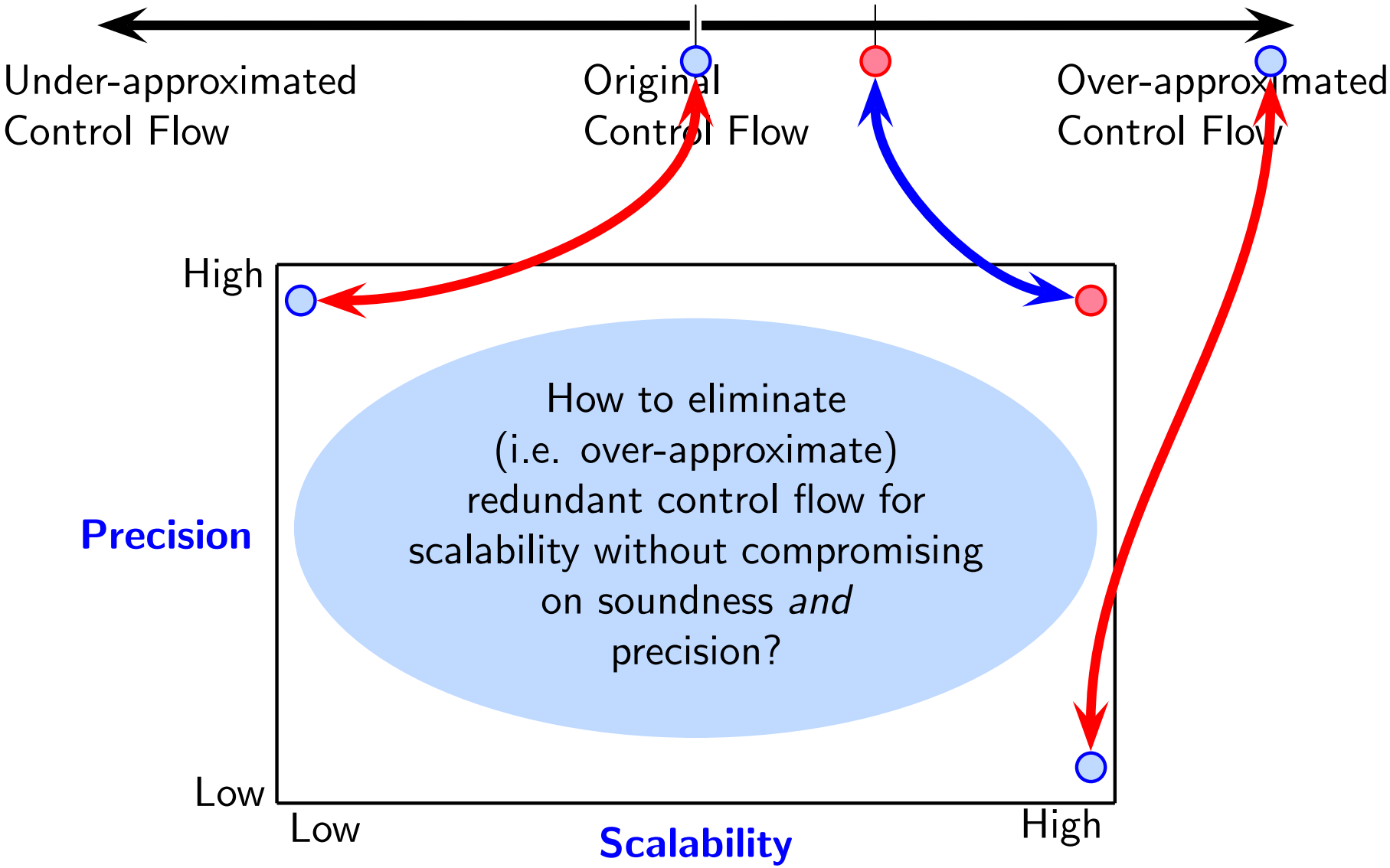
The Next Holy Grail in Search of Scalability?



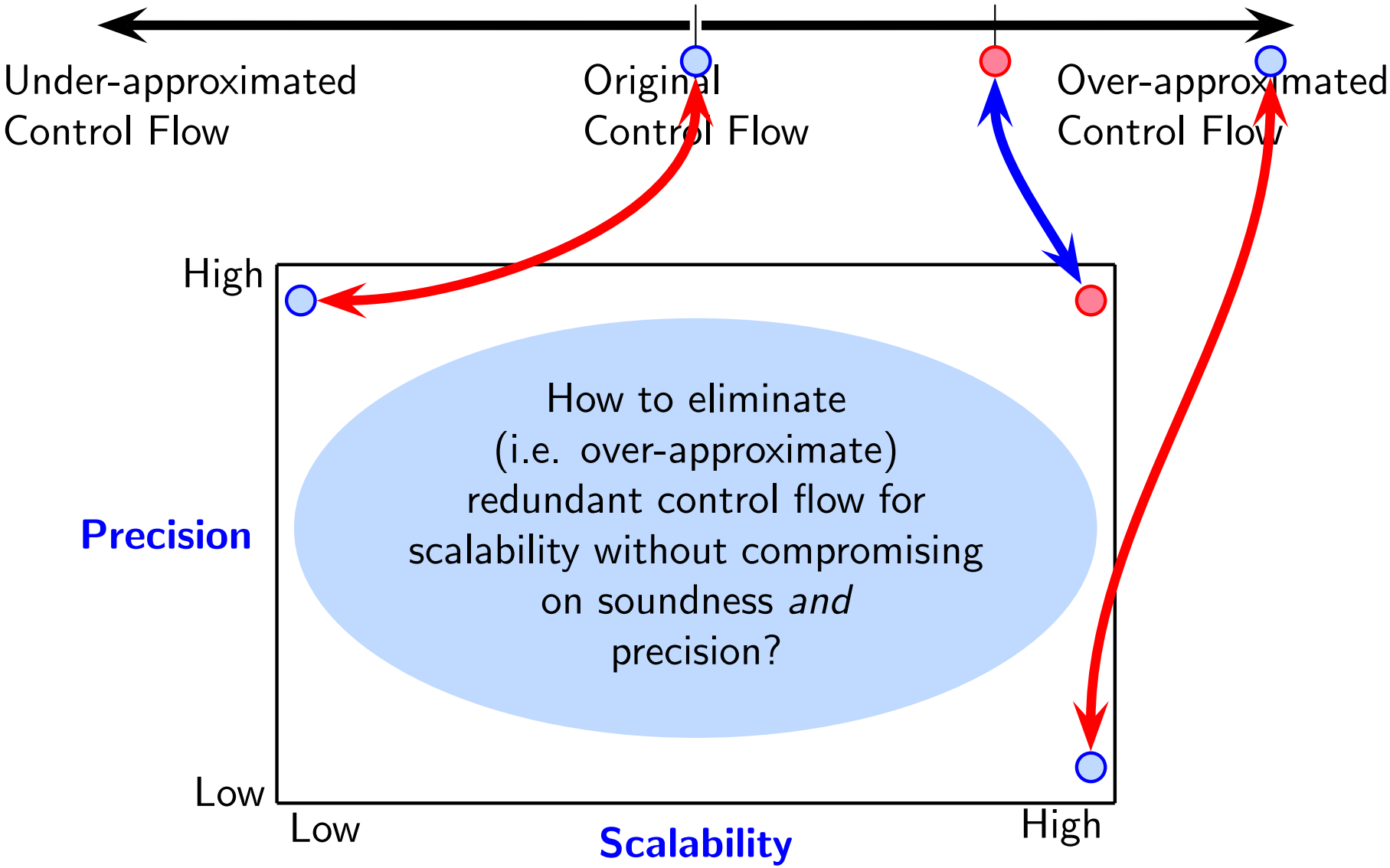
The Next Holy Grail in Search of Scalability?



The Next Holy Grail in Search of Scalability?



The Next Holy Grail in Search of Scalability?



Acknowledgements

- Many people have worked on the topics presented here

Alan Mycroft,
Alefiya Lightwala
Amey Karkare,
Amitabha Sanyal,
Avantika Gupta
Bageshri Sathe,
Binapani Beria,
Prachee Yogi,
Prashant Singh Rawat,
Pritam Gharat,
Priyanka Sawant,
Rohan Padhye,
Shubhangi Agrawal,
Sudakshina Das,
Sushmita Nikose,
Swati Rathi,
Vini Kanvar,
Vinit Deodhar

. . . and many more

- Many others have contributed to the other topics that I work on . . .