Some Examples of Program Analysis Research at IITB

Uday Khedker

(www.cse.iitb.ac.in/˜uday)

Department of Computer Science and Engineering,

Indian Institute of Technology, Bombay

Dec 2019

Part 1

About These Slides

Copyright

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

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

  (Indian edition published by Ane Books in 2013)

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

- A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley. 2006.
- M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland Inc. 1977.

These slides are being made available under GNU FDL v1.2 or later purely for academic or research use.

Outline

Examples of some research explorations in

- Intraprocedural Analysis
    - Liveness analysis of heap data
    - Livenes-based points-to analysis
    - Synergistic program analysis
    - Excluding known infeasible paths
- Interprocedural analysis
    - Broad categories of interprocedural analysis
    - Scaling top-down analysis using value contexts and bypassing
    - Improving bottom-up analysis by eliminating control flow

Part 2

Intraprocedural Analysis

Liveness Analysis of Heap Data

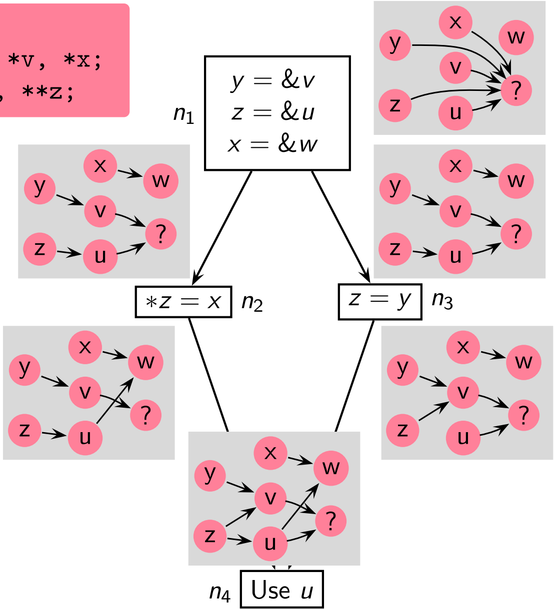
- Example already covered in the introductory lecture
- Unlike stack and static data,
    - Heap data accessible to any procedure is unbounded
    - The mapping between object names and their addresses needs to change at runtime
- We build bounded abstractions of heap data in terms of graphs (called *access graphs*) and perform analysis using these graphs as data flow values
- An access graph at a program point summarizes the paths in the heap memory that are traversed by the rest of the program
    - The paths in the memory constitute a regular language
    - Access graphs are NFAs for the regular language

An Outline of Research Explorations in Intraprocedural Analysis

- ○ Liveness analysis of heap data
- ● Livenes-based points-to analysis Next Topic
- ● Synergistic program analysis
- ● Excluding known infeasible paths

Our Motivating Example for FCPA

```
int w;
int *u, *v, *x;
int **y, **z;
```



n_1: $y = \&v$
$z = \&u$
$x = \&w$

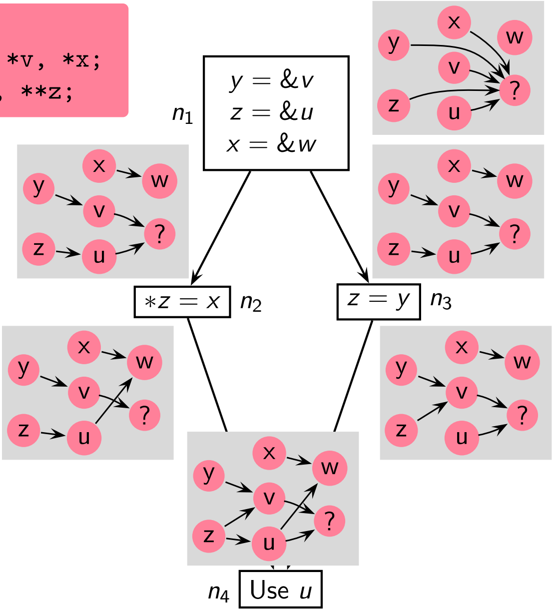
n_2: $*z = x$

n_3: $z = y$

n_4: Use u

Is All This Information Useful

```
int w;
int *u, *v, *x;
int **y, **z;
```



n_1

$y = \&v$
$z = \&u$
$x = \&w$

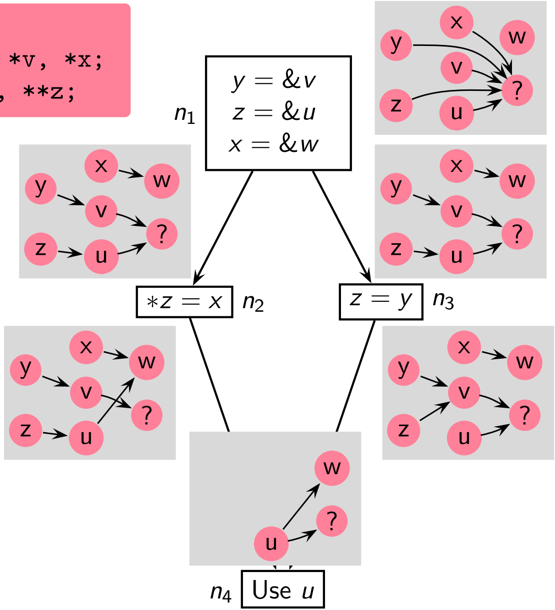
n_2 $*z = x$

n_3 $z = y$

n_4 Use u

Is All This Information Useful

```
int w;
int *u, *v, *x;
int **y, **z;
```



n_1
$y = \&v$
$z = \&u$
$x = \&w$

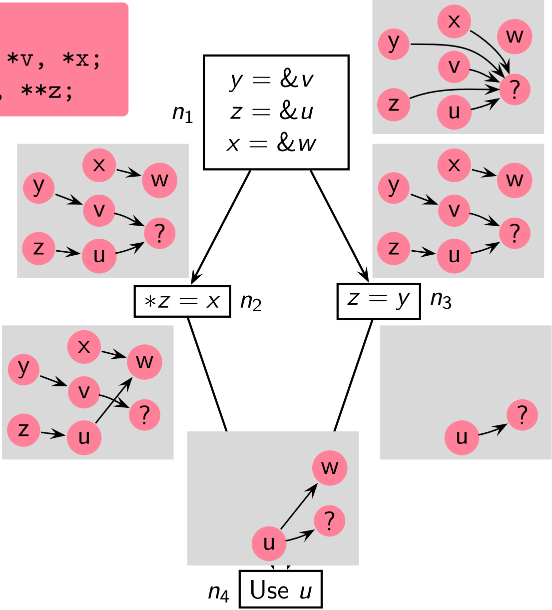
$*z = x$ n_2

$z = y$ n_3

n_4 Use u

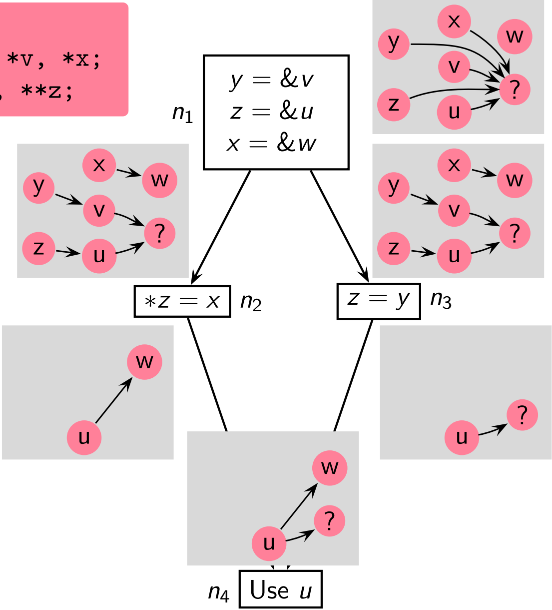
Is All This Information Useful

```
int w;
int *u, *v, *x;
int **y, **z;
```



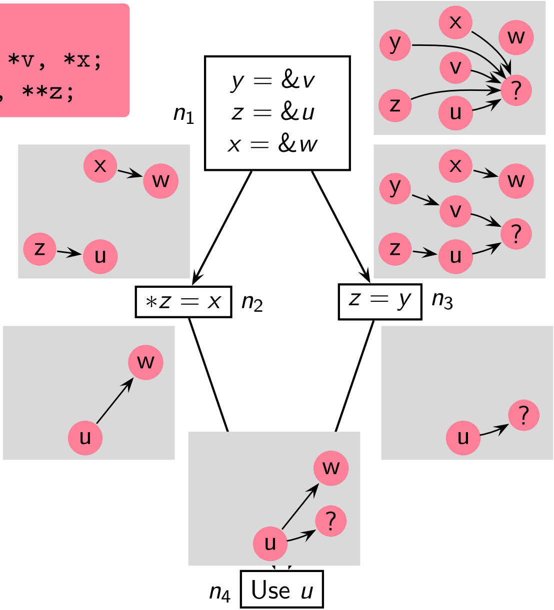
Is All This Information Useful

```
int w;
int *u, *v, *x;
int **y, **z;
```



Is All This Information Useful

```
int w;
int *u, *v, *x;
int **y, **z;
```



n_1

$y = \&v$

$z = \&u$

$x = \&w$

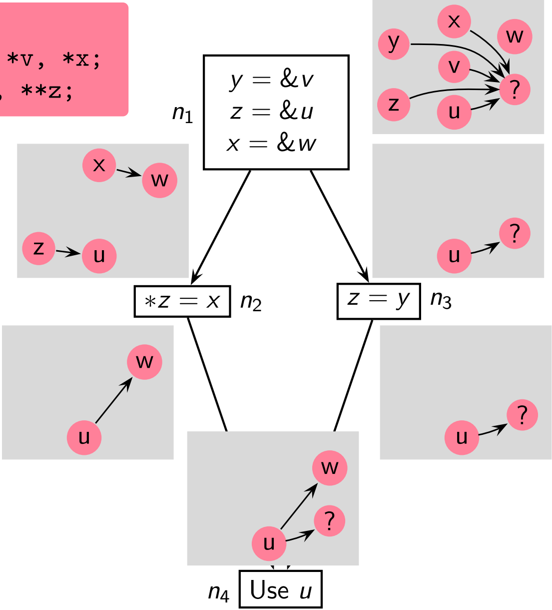
$*z = x$ n_2

$z = y$ n_3

n_4 Use u

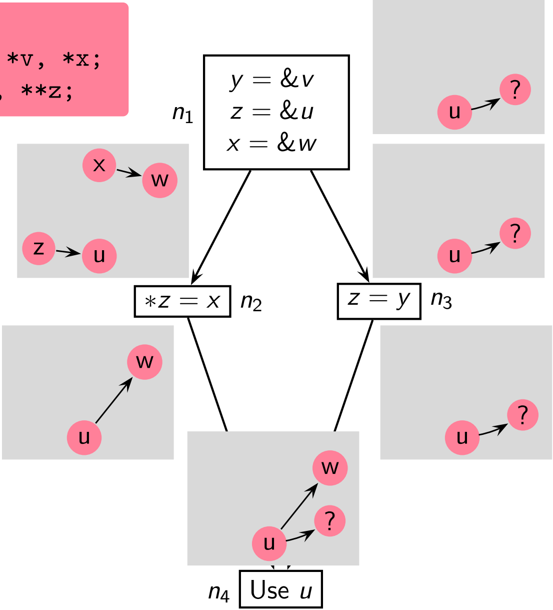
Is All This Information Useful

```
int w;
int *u, *v, *x;
int **y, **z;
```



Is All This Information Useful

```
int w;
int *u, *v, *x;
int **y, **z;
```



Liveness-Based Points-to Analysis (LFCPA)

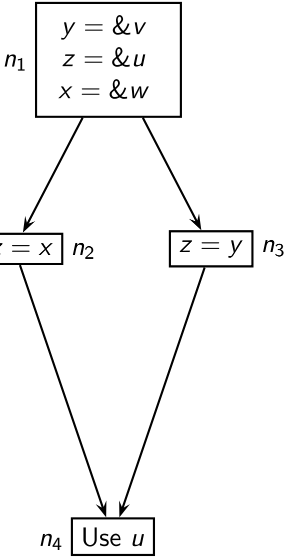
- Mutual dependence of liveness and points-to information
    - Define points-to information only for live pointers
    - For pointer indirections, define liveness information using points-to information
- Use strong liveness
    - Use of a pointer in a non-assignment statement
    - Indirect pointer assignment statement

Motivating Example Revisited

- For convenience, we show complete sweeps of liveness and points-to analysis repeatedly
- This is not required by the computation
- The data flow equations define a single fixed point computation

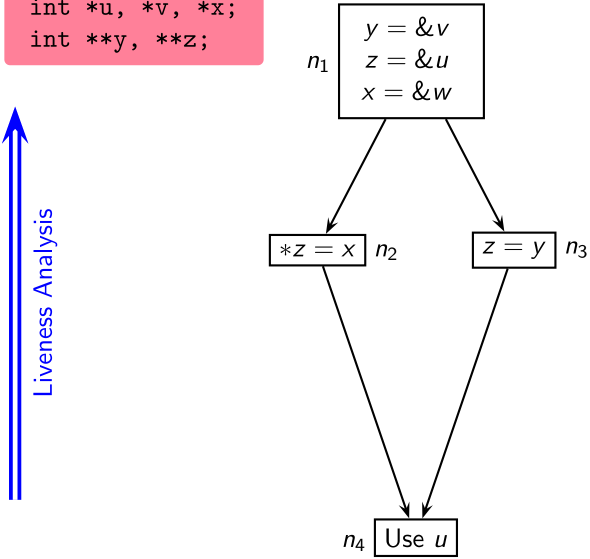
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



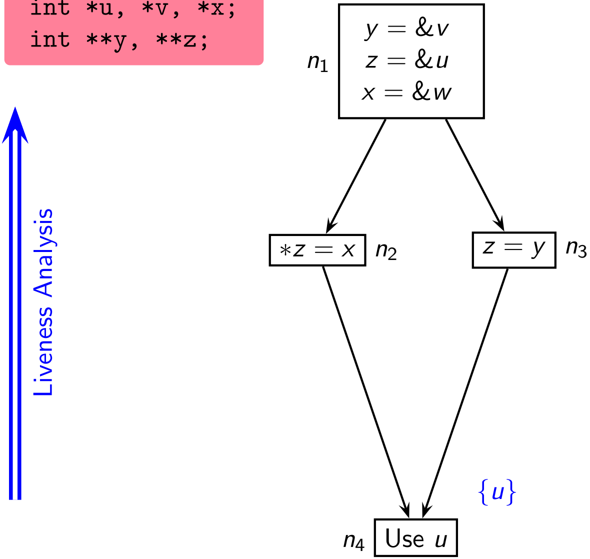
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



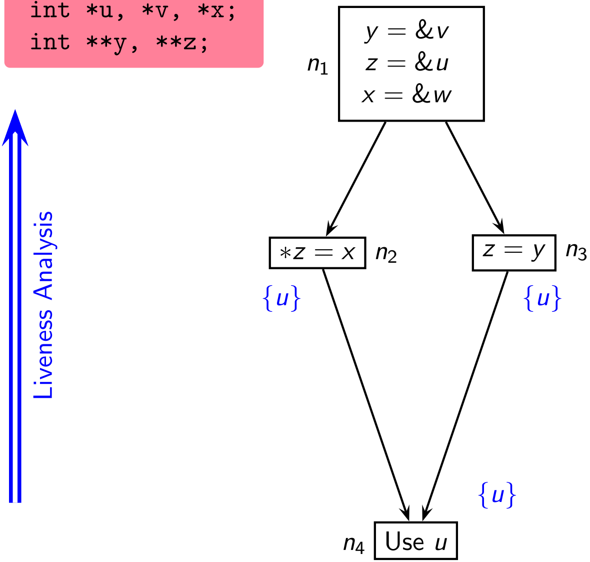
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



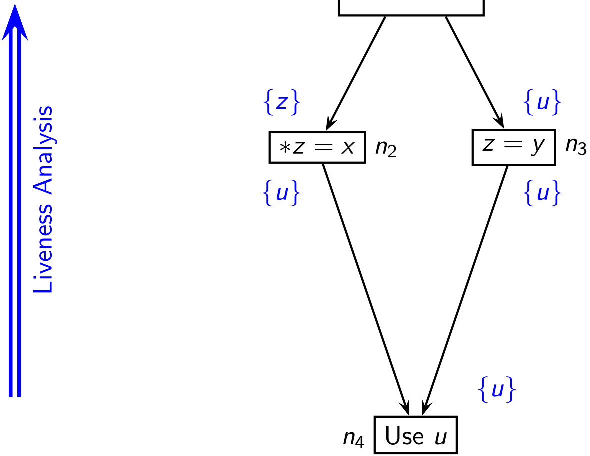
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



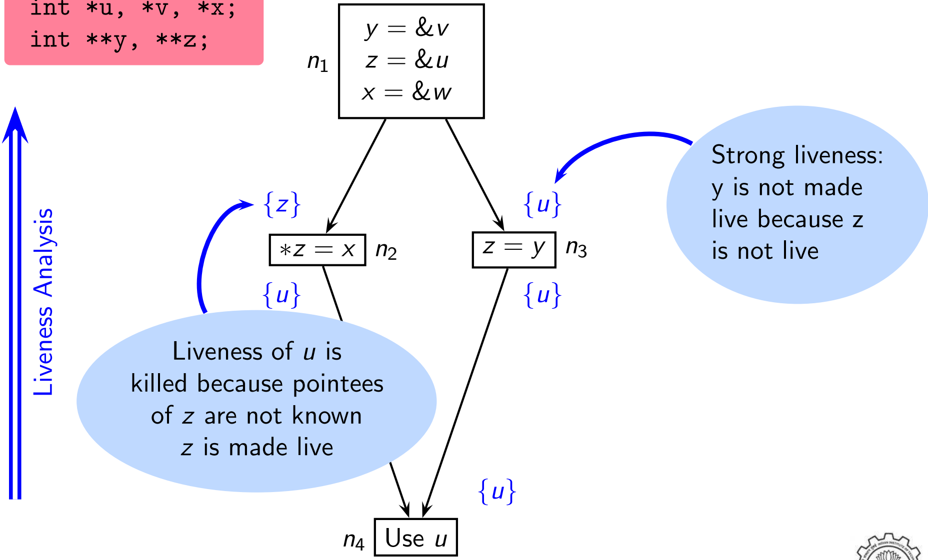
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```

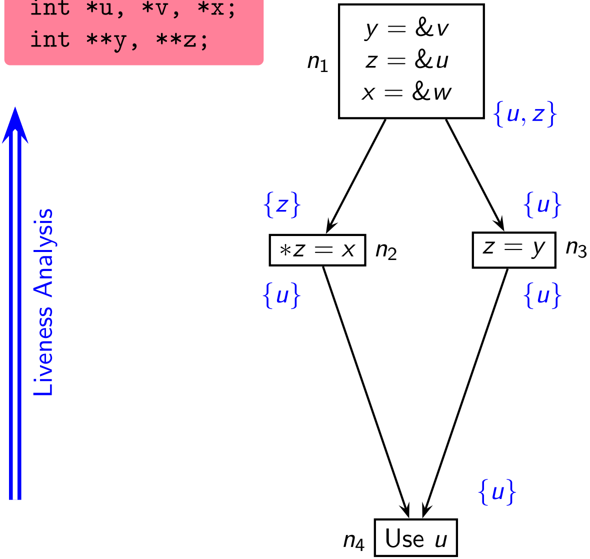


Strong liveness: y is not made live because z is not live

Liveness of u is killed because pointees of z are not known
z is made live

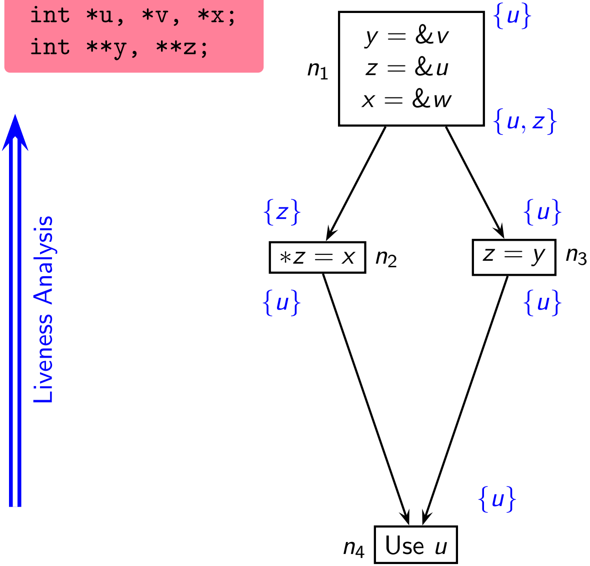
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



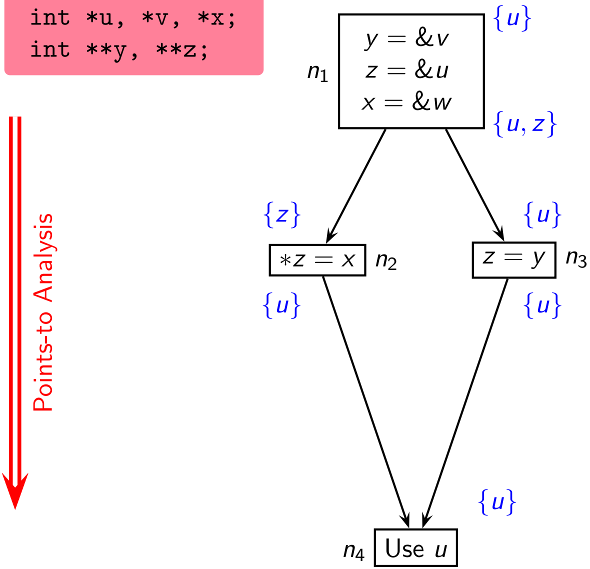
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



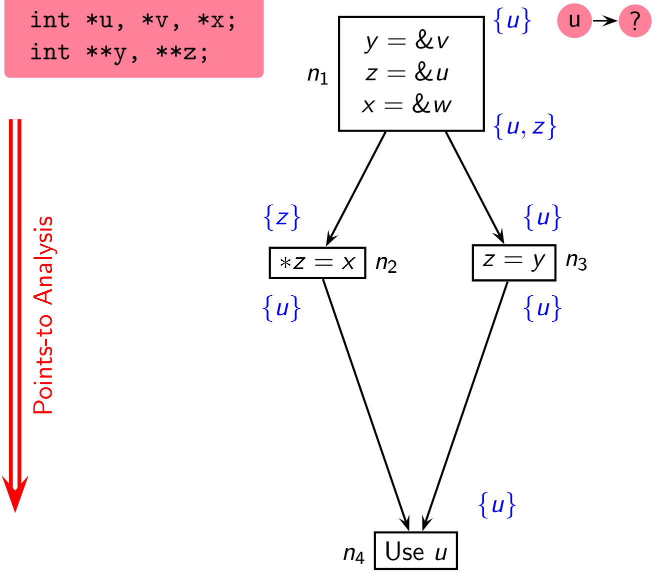
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



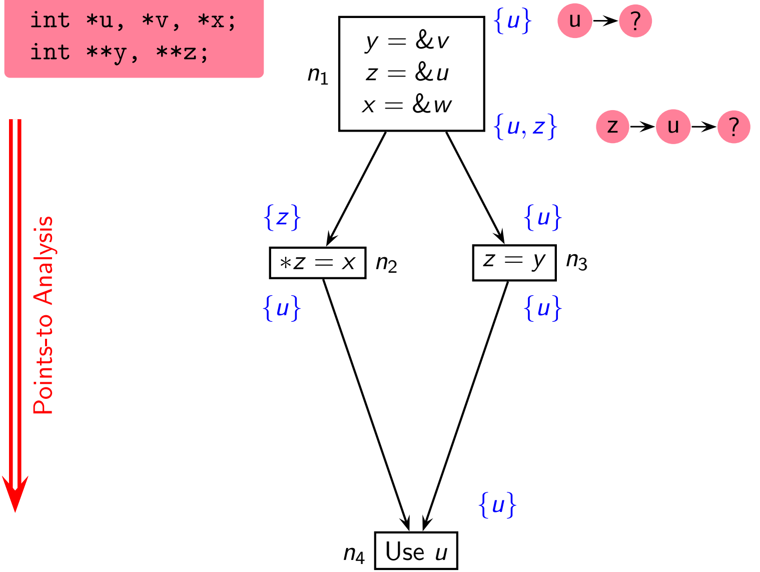
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



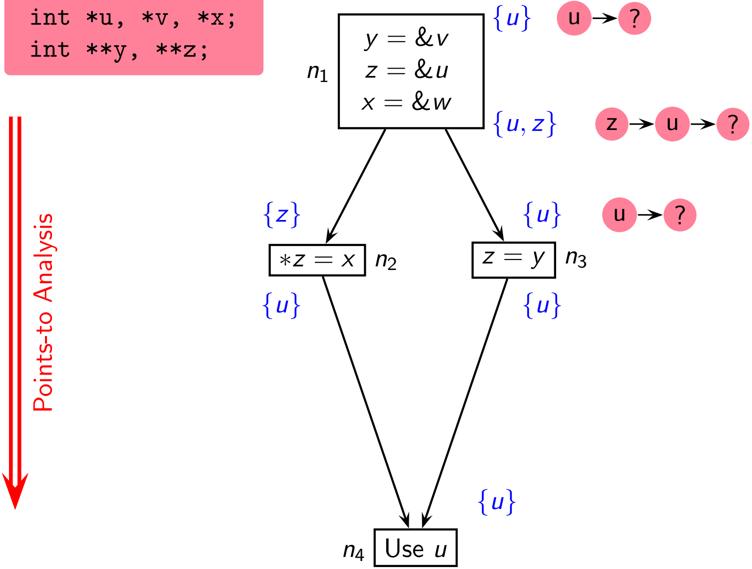
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



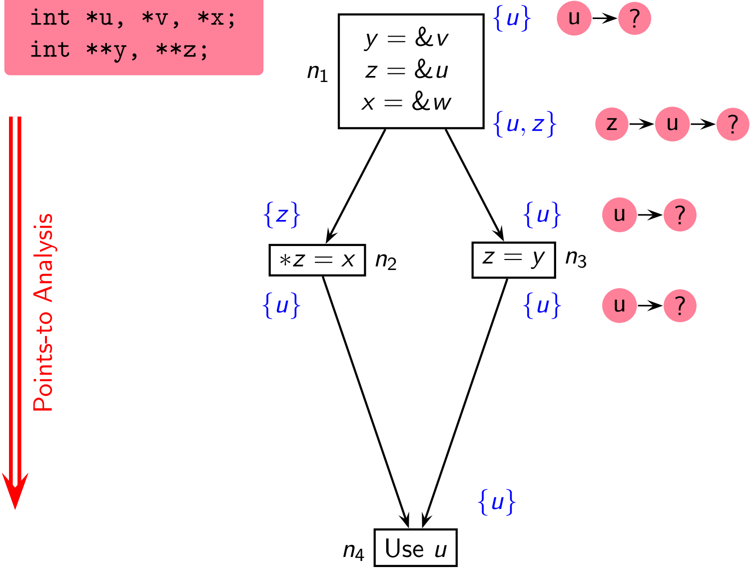
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



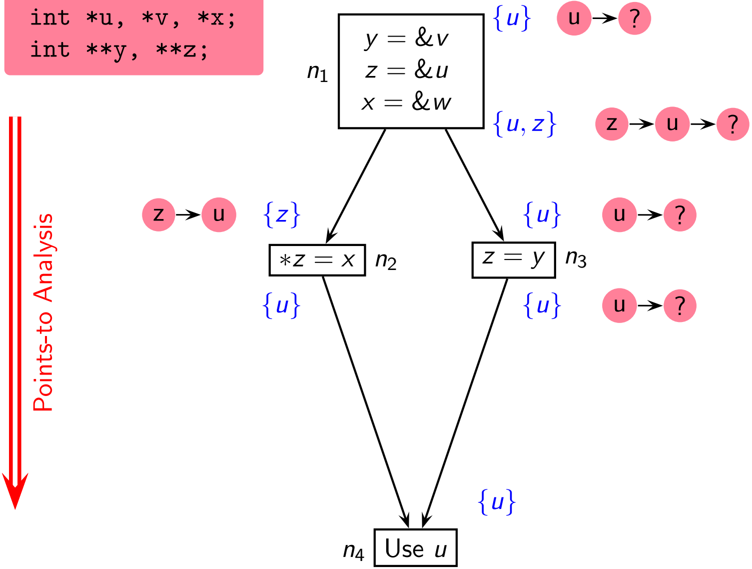
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



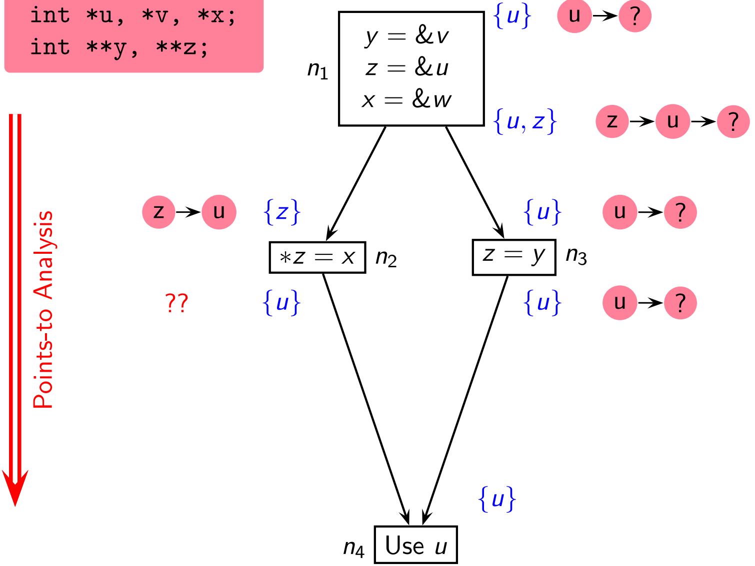
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



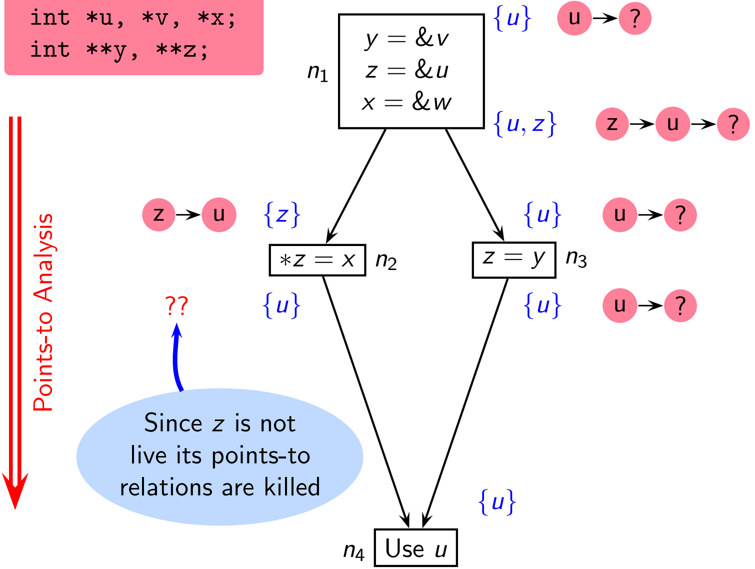
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



First Round of Liveness Analysis and Points-to Analysis

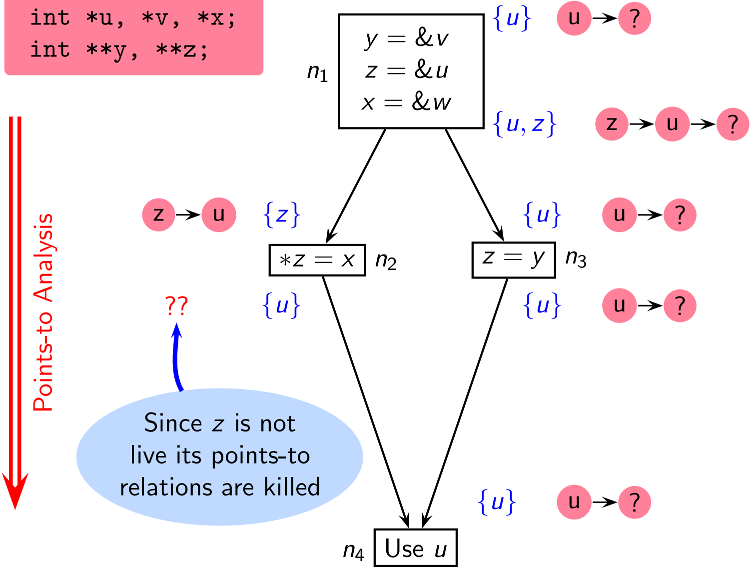
```
int w;
int *u, *v, *x;
int **y, **z;
```



Since z is not live its points-to relations are killed

First Round of Liveness Analysis and Points-to Analysis

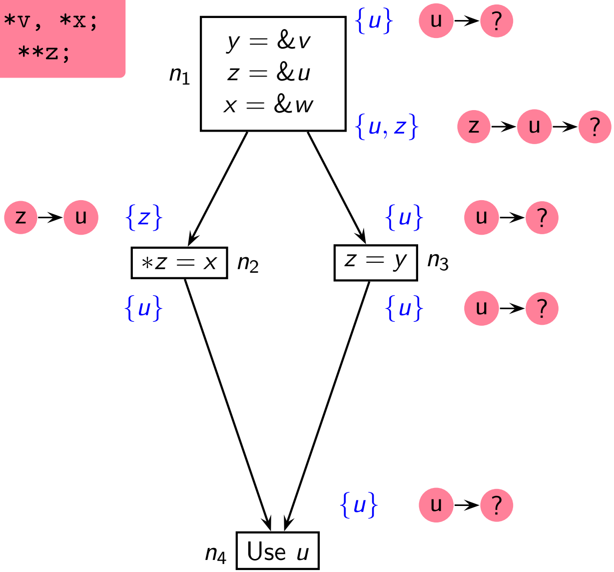
```
int w;
int *u, *v, *x;
int **y, **z;
```



Since z is not live its points-to relations are killed

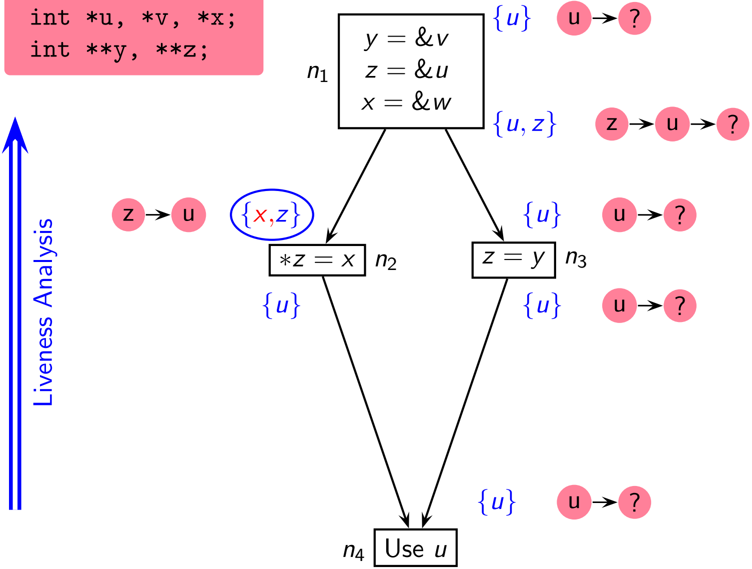
Second Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



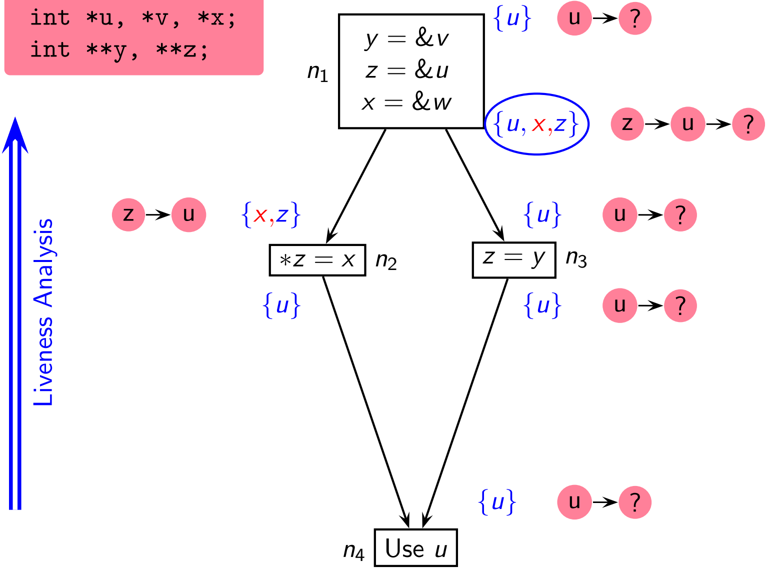
Second Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



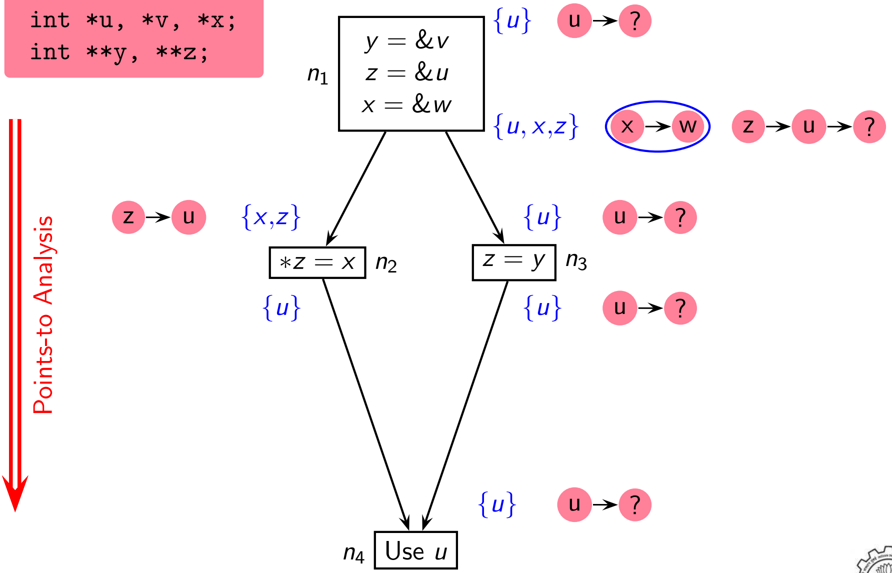
Second Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



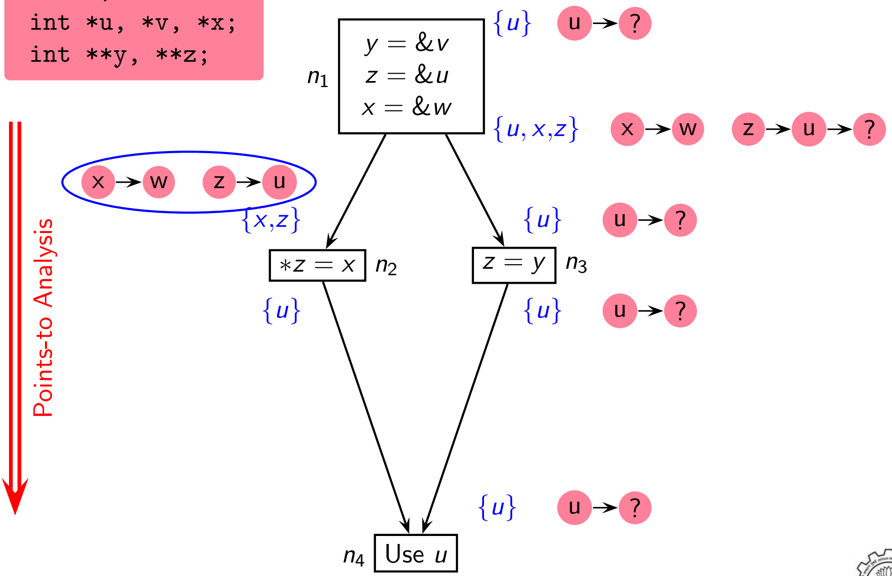
Second Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



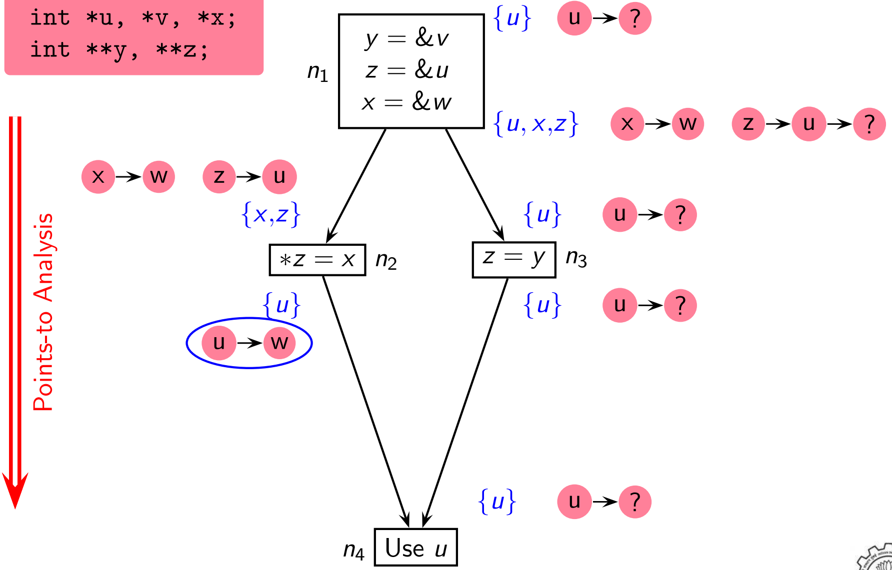
Second Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



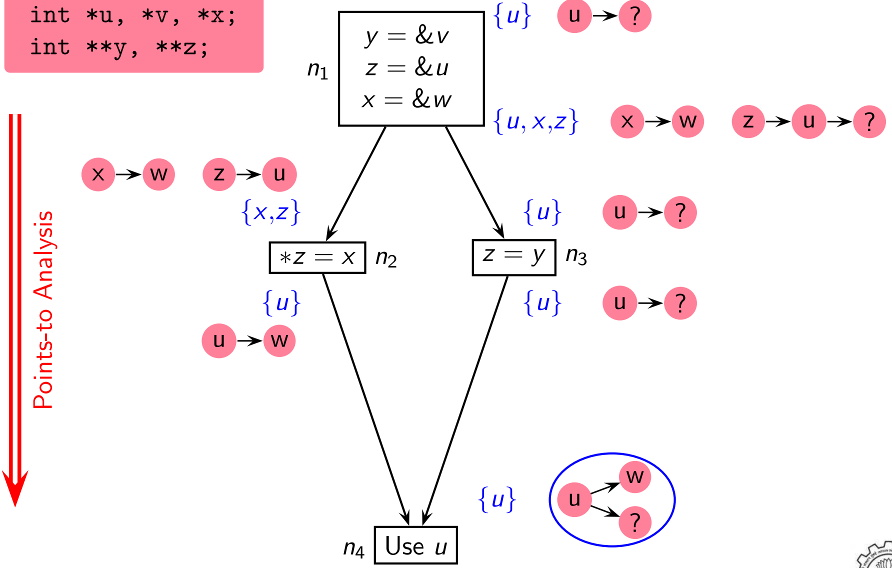
Second Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



Second Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```



LFCPA Observations

- Usable pointer information is very small and sparse
- Data flow propagation in real programs seems to involve only a small subset of all possible data flow values
- Earlier approaches reported inefficiency and non-scalability because they computed far more information than the actual usable information

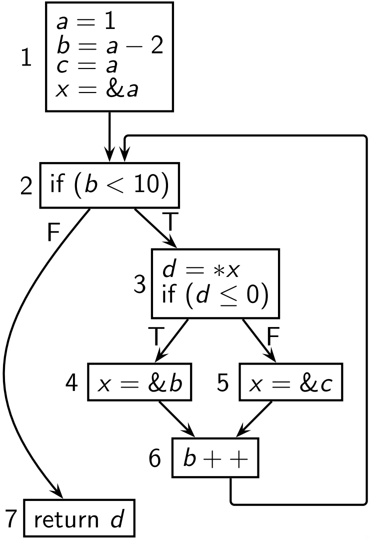
An Outline of Research Explorations in Intraprocedural Analysis

- Livenes-based points-to analysis
- Synergistic program analysis Next Topic
- Excluding known infeasible paths

Interaction Between Constant Propagation and Points-to Analysis

```
int main()
{  int a, b, c, d, *x;
   a=1; b=a-2; c=a;
   x=&a;
   while (b<10)
   {  d=*x;
      if (d<=0) x=&b;
      else x=&c;
      b++;
   }
   return d;
}
```

The value of d in the loop is 1, the condition fails, and x does not point to b at any time



Interaction Between Constant Propagation and Points-to Analysis

We have three options to enable interaction (illustrated next)

- Conventional Cascading. Perform analyses in a fixed sequence
    - CP → transform the program → PTA
    - PTA → transform the program → CP

  This method fails on our example

- Simultaneous Analyses (Lerner's method)

  Perform CP and PTA in locked steps and transform the program whenever possible, repeat the analyses as long as tranformations are possible

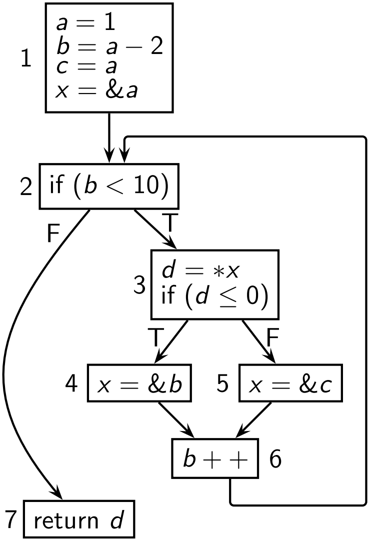
  This method also fails on our example

- Interleaved Synergistic Program Analysis (SPAN)

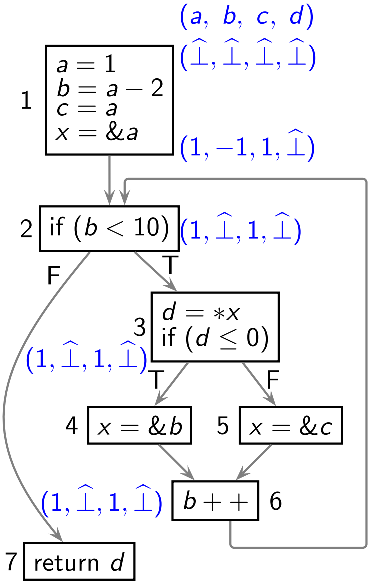
  Interleave the analyses on a need basis, use data flow values to achieve the effect of transforming the program (without actually transforming it)

  This method succeeds on our example

Interaction Between Constant Propagation and Points-to Analysis



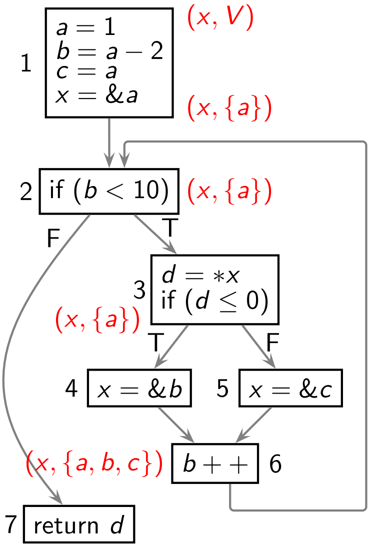
Interaction Between Constant Propagation and Points-to Analysis



If we perform constant propagation first,

- We do not know the pointees of x in node 3, hence we assume all variables as possible pointees
- Thus the value of d is $\bot$ and the branch outcome is uncertain and no path is ruled out

Interaction Between Constant Propagation and Points-to Analysis



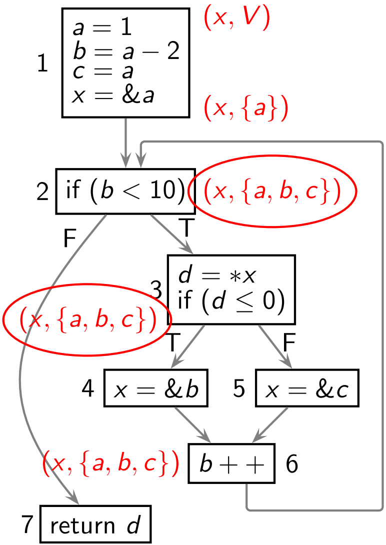
If we perform constant propagation first,

- We do not know the pointees of x in node 3, hence we assume all variables as possible pointees
- Thus the value of d is $\perp$ and the branch outcome is uncertain and no path is ruled out

Then, when we perform points-to analysis,

- The pointees of x are found to be a, b, and c
- $d = *x$ cannot be simplified
- A subsequent round of constant propagation will find d to be $\perp$

Interaction Between Constant Propagation and Points-to Analysis



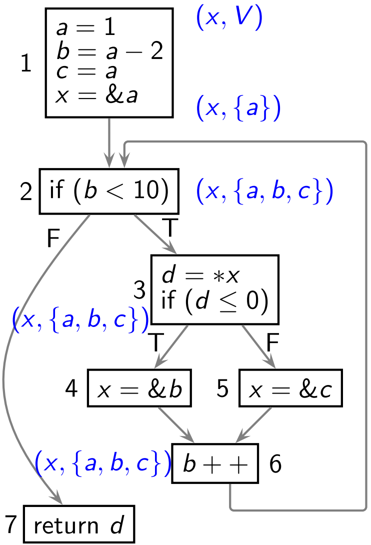
If we perform constant propagation first,

- We do not know the pointees of x in node 3, hence we assume all variables as possible pointees
- Thus the value of d is $\perp$ and the branch outcome is uncertain and no path is ruled out

Then, when we perform points-to analysis,

- The pointees of x are found to be a, b, and c
- $d = *x$ cannot be simplified
- A subsequent round of constant propagation will find d to be $\perp$

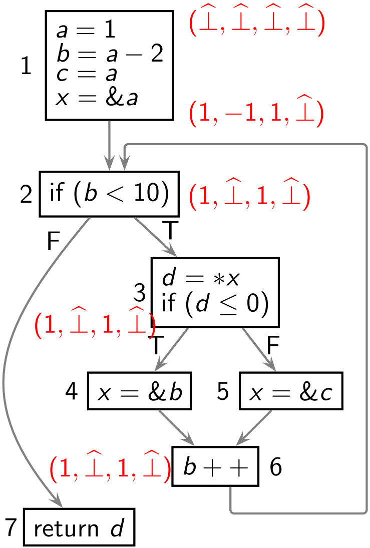
Interaction Between Constant Propagation and Points-to Analysis



If we perform points-to analysis first,

- The pointees of x are found to be a, b, and c because both branch outcomes are possible
- $d = *x$ cannot be simplified

Interaction Between Constant Propagation and Points-to Analysis



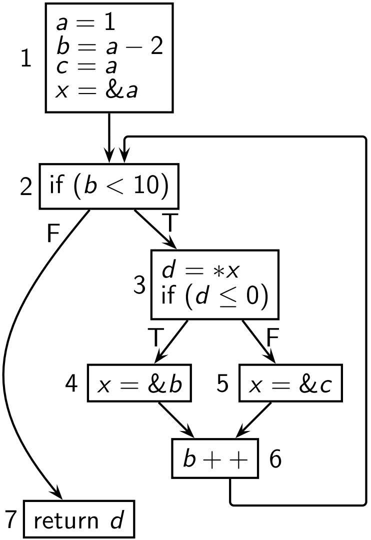
If we perform points-to analysis first,

- The pointees of x are found to be a, b, and c because both branch outcomes are possible
- $d = *x$ cannot be simplified

Then, when we perform constant propagation

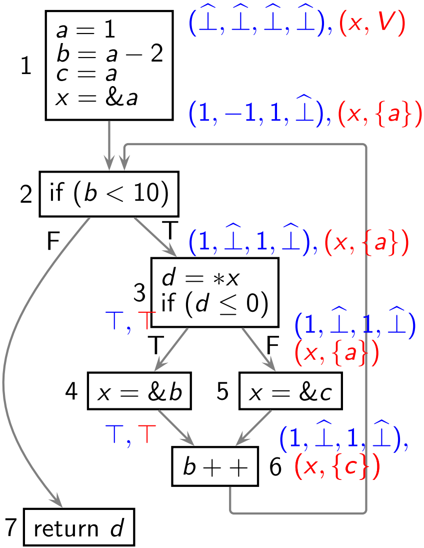
- The value of d is $\perp$ and the branch outcome is uncertain and no path is ruled out
- A subsequent round of points-to analysis will find the pointees of x as a, b and c

Interaction Between Constant Propagation and Points-to Analysis



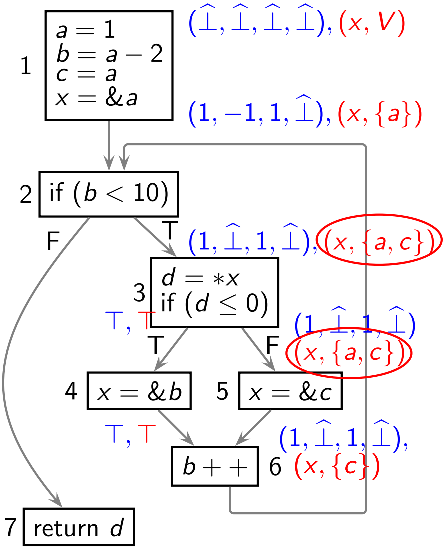
- The precision of the two analyses depends on each other's results
- If we perform them together, we can rule out the T branch out of node 3, x points to a and c, and both are 1

Interaction Between Constant Propagation and Points-to Analysis



- The precision of the two analyses depends on each other's results
- If we perform them together, we can rule out the T branch out of node 3, x points to a and c, and both are 1
- SPAN achieves this

Interaction Between Constant Propagation and Points-to Analysis



- The precision of the two analyses depends on each other's results
- If we perform them together, we can rule out the T branch out of node 3, x points to a and c, and both are 1
- SPAN achieves this

Interaction Between Constant Propagation and Points-to Analysis

SPAN is more general than Lerner's method because

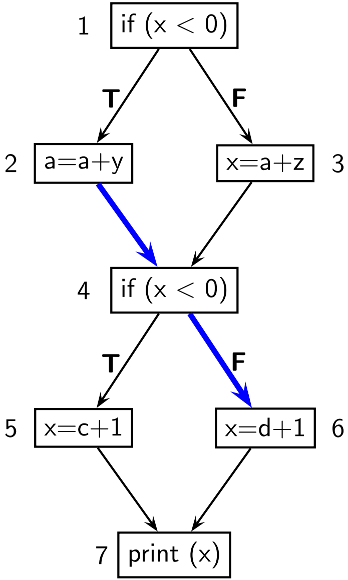
- SPAN does not tranform the program but uses data flow values (Lerner's method tries to transform $d = *x$ and fails)
- The analyses need not be performed in locked steps and hence forward and backward analyses can be combined
- The need of interaction is inferred automatically and the user does not need to specify it
- Arbitrary data flow analyses can be added to the system at will

  Each analysis must specify the statements that it can (conceptually) simplify and the statements that it cannot simplify

An Outline of Research Explorations in Intraprocedural Analysis

- Livenes-based points-to analysis
- Synergistic program analysis
- Excluding known infeasible paths Next Topic

Excluding the Known Infeasible Paths

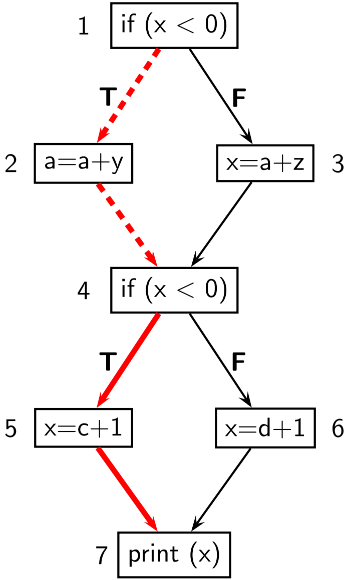


- Every path containing $\rho\colon(2,4,6)$ is infeasible

  It could lead to imprecision (e.g. *d* is spuriously marked live at the exit of node 2)
- We cannot delete any edge to exclude this path

  Such deletion could lead to unsoundness

Excluding the Known Infeasible Paths



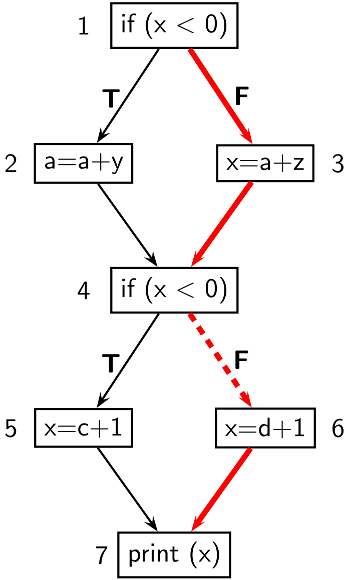
- Every path containing $\rho : (2, 4, 6)$ is infeasible

  It could lead to imprecision (e.g. *d* is spuriously marked live at the exit of node 2)

- We cannot delete any edge to exclude this path
  - If we delete edge $(2, 4)$, it excludes a feasible path $(1, 2, 4, 5, 7)$

Such deletion could lead to unsoundness

Excluding the Known Infeasible Paths



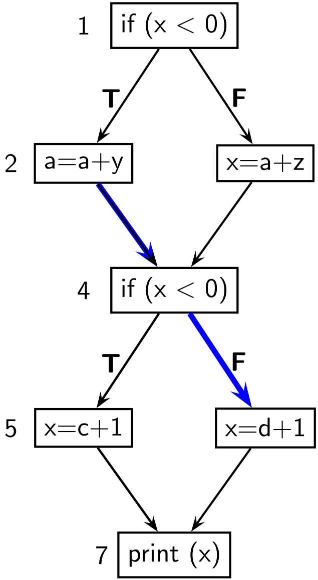
- Every path containing $\rho : (2, 4, 6)$ is infeasible

  It could lead to imprecision (e.g. *d* is spuriously marked live at the exit of node 2)

- We cannot delete any edge to exclude this path
  - If we delete edge $(2, 4)$, it excludes a feasible path $(1, 2, 4, 5, 7)$
  - If we delete edge $(4, 6)$, it excludes a feasible path $(1, 3, 4, 6, 7)$

  Such deletion could lead to unsoundness

Excluding the Known Infeasible Paths



- Every path containing $\rho : (2, 4, 6)$ is infeasible

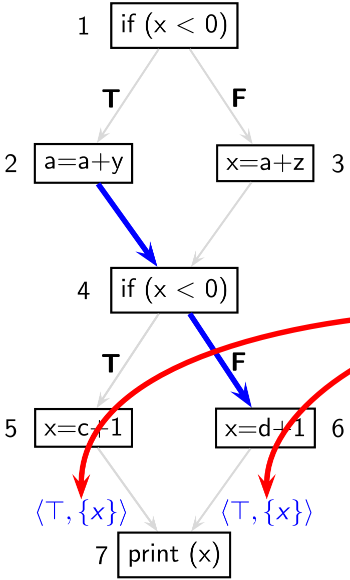
  It could lead to imprecision (e.g. *d* is spuriously marked live at the exit of node 2)

- We cannot delete any edge to exclude this path
  - If we delete edge $(2, 4)$, it excludes a feasible path $(1, 2, 4, 5, 7)$
  - If we delete edge $(4, 6)$, it excludes a feasible path $(1, 3, 4, 6, 7)$

  Such deletion could lead to unsoundness

- Our solution: At each edge, distinguish the data flow value of ρ from other values so that it is not allowed to go out of ρ on an infeasible path

Excluding the Known Infeasible Paths

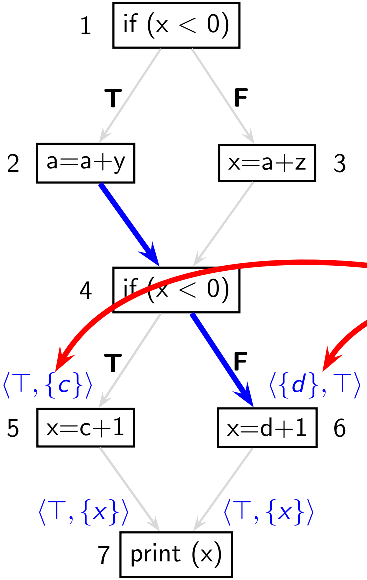


Our Notation: $\langle$dfv ofρ, other dfv$\rangle$

Edges 5 → 7 and 6 → 7 are not a part of ρ

Hence the data flow value in the first component is $\top$

Excluding the Known Infeasible Paths

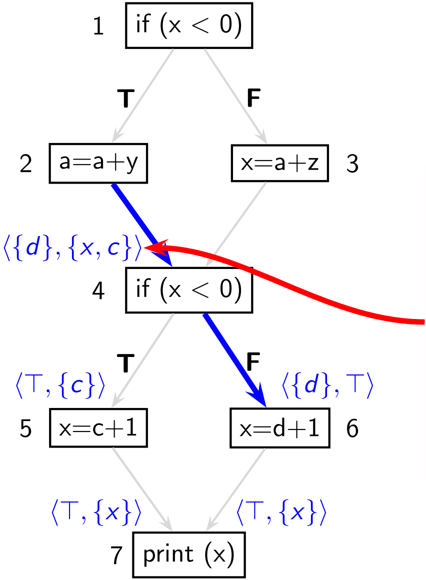


Our Notation: $\langle$dfv ofρ, other dfv$\rangle$

Edge 4 → 5 is not a part of ρ and the first component is $\top$

Edge 4 → 6 is a part of ρ but edge 6 → 7 is not a part of ρ (i.e. the effect of ρ begins here) so the data flow value shifts from the second component to the first component

Excluding the Known Infeasible Paths

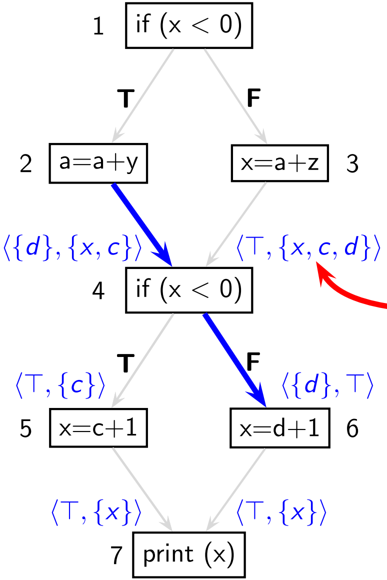


Our Notation: $\langle$dfv ofρ, other dfv$\rangle$

Edge 2 → 4 is a part of ρ hence it will continue to hold the data flow value of ρ coming from edge 4 → 6 which is also a part of ρ

The data flow value generated in node 4 or the data flow value coming from edge 4 → 5 go to the second component because a path that does not include ρ completely, is not infeasible

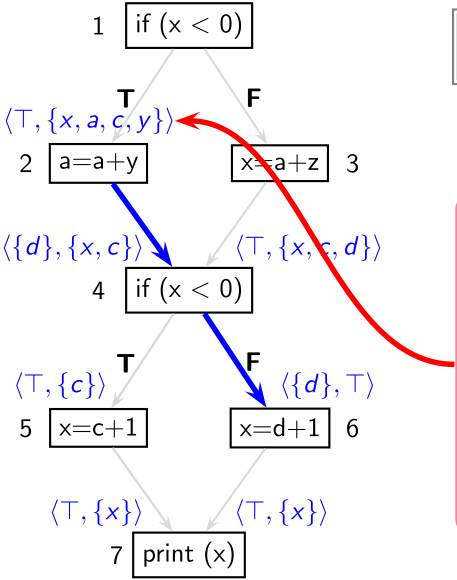
Excluding the Known Infeasible Paths



Our Notation: $\langle$dfv ofρ, other dfv$\rangle$

Edge 3 → 4 is not a part of ρ hence the first component is $\top$ and all data flow values move to the second component

Excluding the Known Infeasible Paths



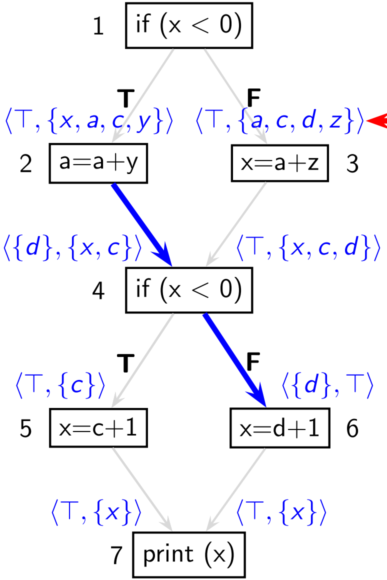
Our Notation: $\langle \text{dfv of} \rho, \text{other dfv} \rangle$

Edge 1 → 2 is not a part of ρ hence the first component is $\top$ and all data flow values move to the second component

Since d belongs to ρ, it is blocked and is not propagated further because the path $(1, 2, 4, 6, 7)$ is infeasible

This separation and blocking of values gives a more precise solution than the usual MFP solution

Excluding the Known Infeasible Paths



Our Notation: $\langle$dfv ofρ, other dfv$\rangle$

Edge 1 → 3 is not a part of ρ hence the first component is $\top$ and all data flow values move to the second component

Excluding the Known Infeasible Paths

- Infeasibility is a property of the control flow graph and not that of an analysis
- Our method takes as input the information about the minimal infeasible path segments in program
- Our method is very general
    - It handles multiple minimal infeasible path segments that may overlap with each other
    - It lifts any data flow analysis to an analysis that excludes the effect of known infeasible paths
- Existing approaches to remove the effect of infeasible paths are either analysis specific or involve CFG restructuring

  Our approach avoids CFG restructuring and still achieves a generic solution

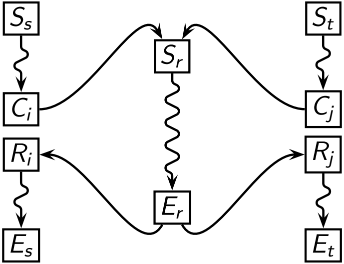
Part 3

Interprocedural Analysis

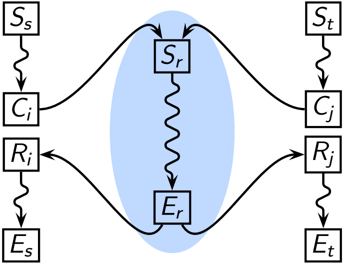
An Outline of Research Explorations in Interprocedural Analysis

- Broad categories of interprocedural analysis Next Topic
- Scaling top-down analysis using value contexts and bypassing
- Improving bottom-up analysis by eliminating control flow

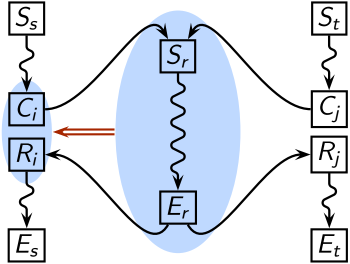
Understanding Context Sensitivity



Understanding Context Sensitivity

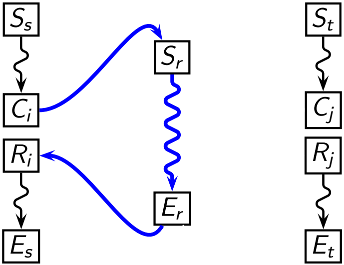


Understanding Context Sensitivity



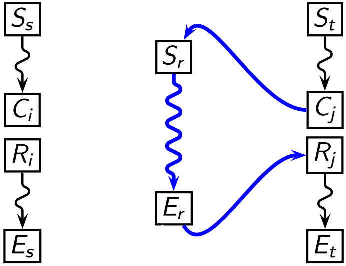
Precise interprocedural analysis aims to achieve the effect of inlining

Understanding Context Sensitivity



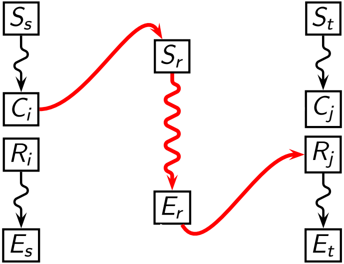
Interprocedurally valid path

Understanding Context Sensitivity



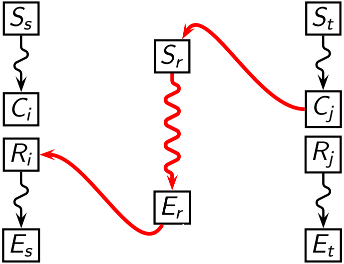
Interprocedurally valid path

Understanding Context Sensitivity



Interprocedurally invalid path

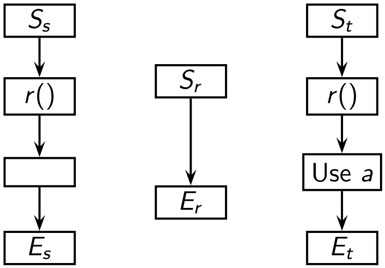
Understanding Context Sensitivity



Interprocedurally invalid path

Context Sensitivity Vs. Context Insensitivity

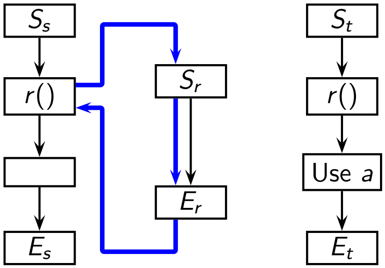
Context-sensitive Analysis	Context-insensitive Analysis



Data flow values of distinct contexts are kept as separate values

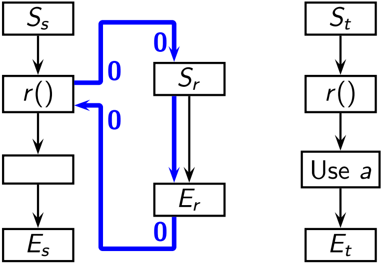
Data flow values of all contexts are merged into a single value

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

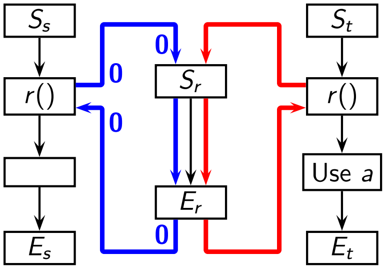
Context Sensitivity Vs. Context Insensitivity

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

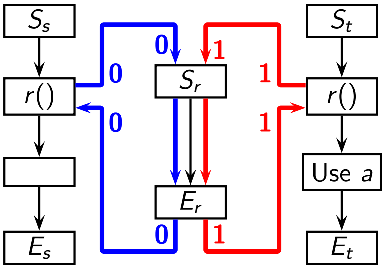
Context-sensitive Analysis	Context-insensitive Analysis



Data flow values of distinct contexts are kept as separate values

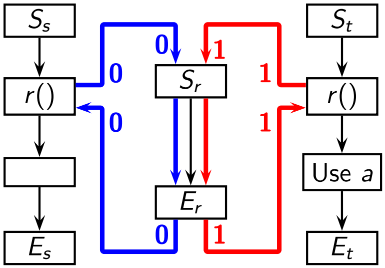
Data flow values of all contexts are merged into a single value

Context Sensitivity Vs. Context Insensitivity

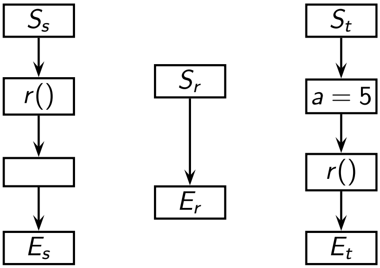
Context-sensitive Analysis	Context-insensitive Analysis
	
Data flow values of distinct contexts are kept as separate values	Data flow values of all contexts are merged into a single value

Context Sensitivity Vs. Context Insensitivity

Context-sensitive Analysis	Context-insensitive Analysis



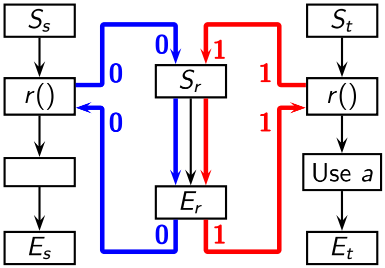
Data flow values of distinct contexts are kept as separate values



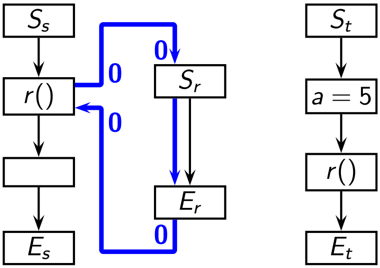
Data flow values of all contexts are merged into a single value

Context Sensitivity Vs. Context Insensitivity

Context-sensitive Analysis	Context-insensitive Analysis



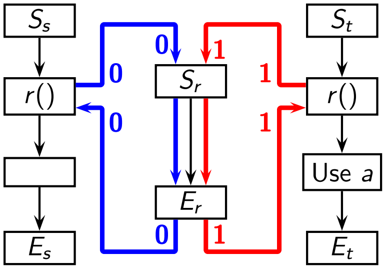
Data flow values of distinct contexts are kept as separate values



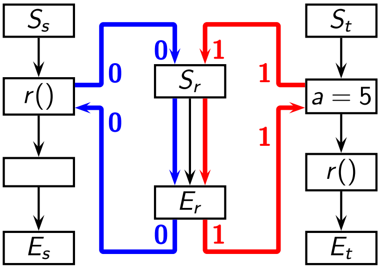
Data flow values of all contexts are merged into a single value

Context Sensitivity Vs. Context Insensitivity

Context-sensitive Analysis	Context-insensitive Analysis



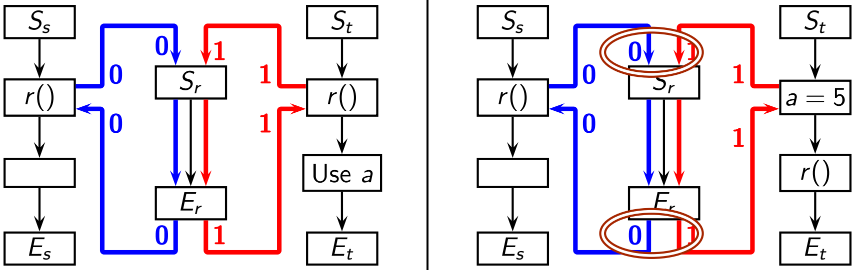
Data flow values of distinct contexts are kept as separate values



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

Context Sensitivity Vs. Context Insensitivity

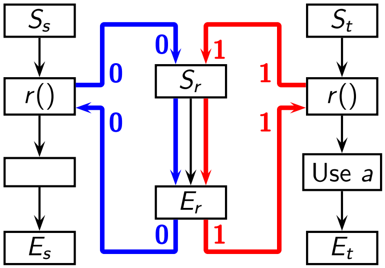
Context-sensitive Analysis	Context-insensitive Analysis



Data flow values of distinct contexts are kept as separate values

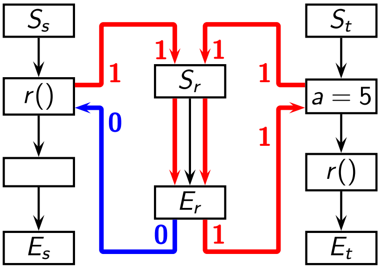
Data flow values of all contexts are merged into a single value

Context Sensitivity Vs. Context Insensitivity



Context-sensitive Analysis

Data flow values of distinct contexts are kept as separate values



Context-insensitive Analysis

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

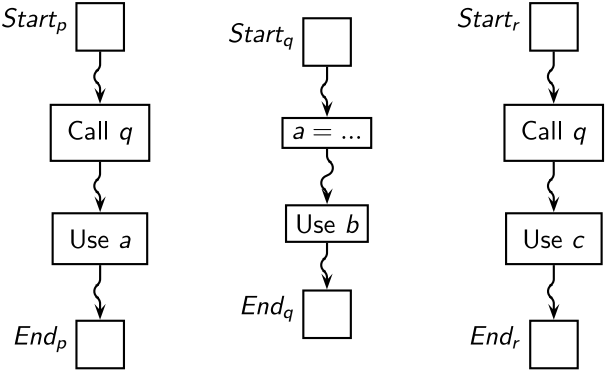
Understanding Context Sensitivity

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
- analyzing a procedure for a particular data flow value and using the analysed result at the call point (graph reachability, value context method)

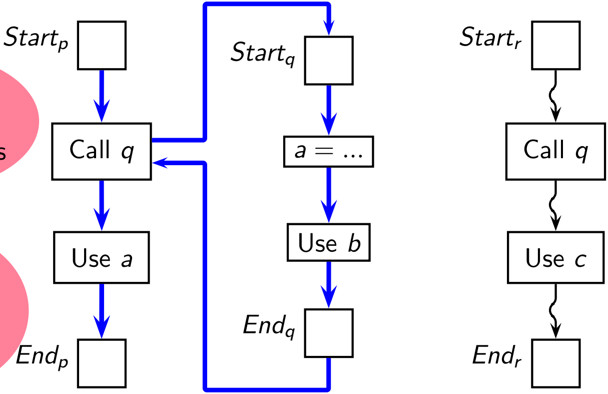
Top-down Vs. Bottom-up (Context-Sensitive) Procedure Summaries

Top-down Live Variables Analysis



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

Top-down Live Variables Analysis



Procedure q is analysed multiple times

Contexts are created explicitly

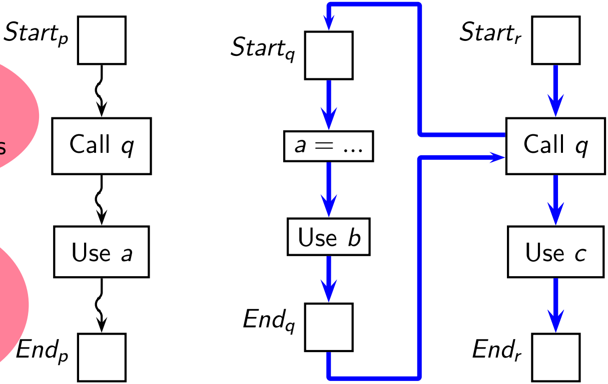
Context σ_1

Variable b is live at S_p

Variables a and c are not live at S_p

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

Top-down Live Variables Analysis



Procedure q is analysed multiple times

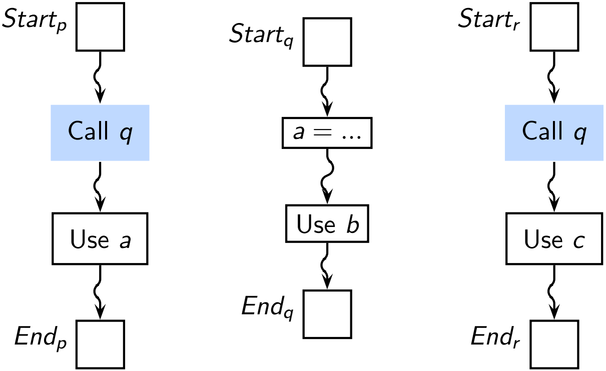
Contexts are created explicitly

Context σ_2 Variables b and c are live at S_r

Variable a is not live at S_r

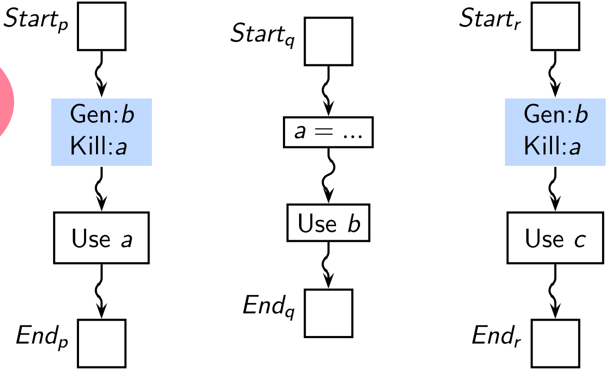
Top-down Vs. Bottom-up (Context-Sensitive) Procedure Summaries

Bottom-Up Live Variables Analysis



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

Bottom-Up Live Variables Analysis



Procedure q is analysed once

Contexts are left implicit

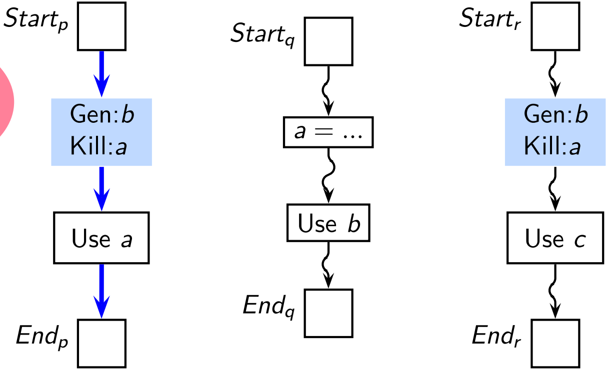
Using procedure summary of q at call sites

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

Bottom-Up Live Variables Analysis

Procedure q is analysed once

Contexts are left implicit



Variable b is live at S_p

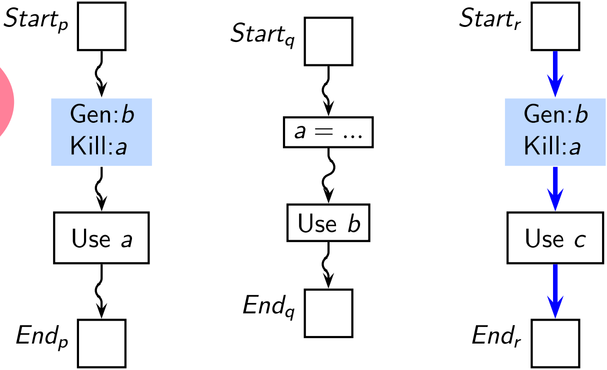
Variables a and c are not live at S_p

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

Bottom-Up Live Variables Analysis

Procedure q is analysed once

Contexts are left implicit



Variables b and c are live at S_r

Variable a is not live at S_r

An Outline of Research Explorations in Interprocedural Analysis

- Broad categories of interprocedural analysis
- Scaling top-down analysis using value contexts and bypassing

  Next Topic
- Improving bottom-up analysis by eliminating control flow

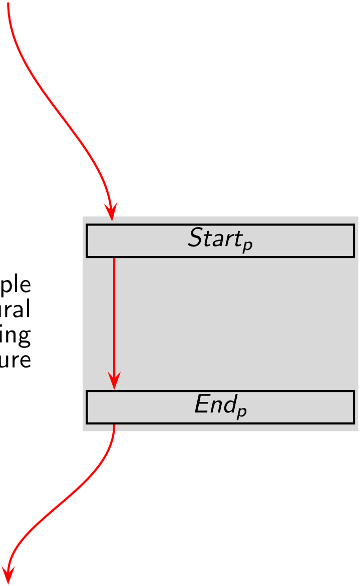
Value Contexts

$Start_p$

Procedure
Body

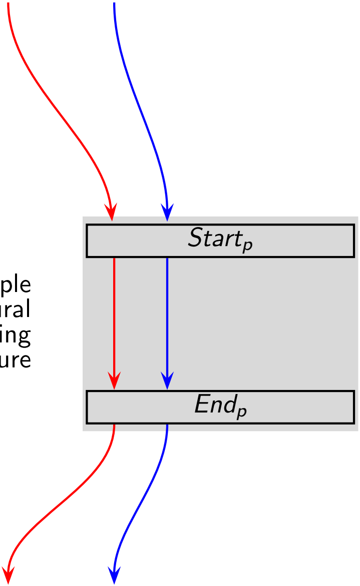
End_p

Value Contexts



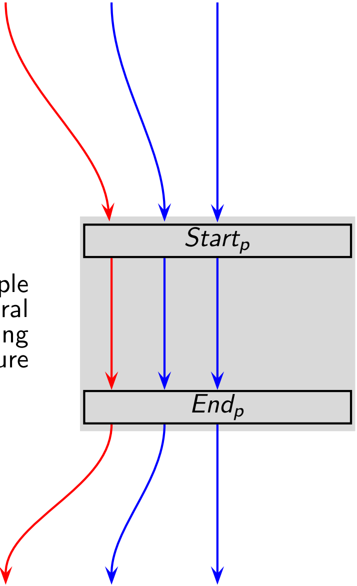
Multiple interprocedural paths reaching the procedure

Value Contexts



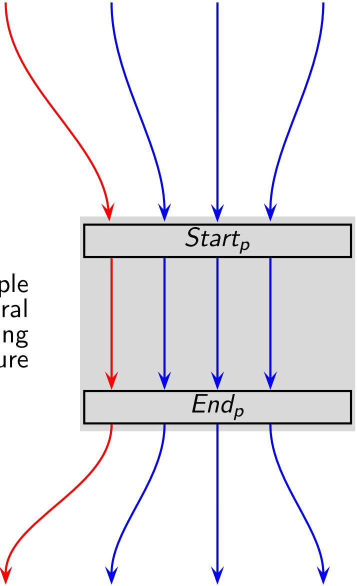
Multiple interprocedural paths reaching the procedure

Value Contexts



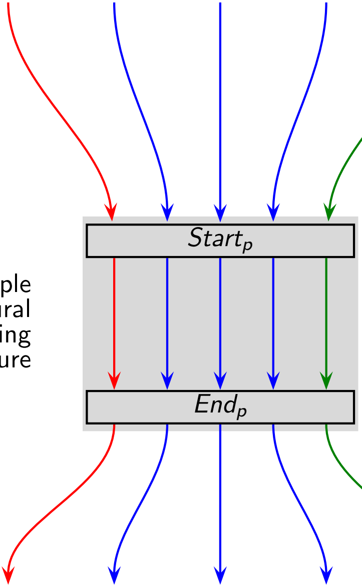
Multiple interprocedural paths reaching the procedure

Value Contexts



Multiple interprocedural paths reaching the procedure

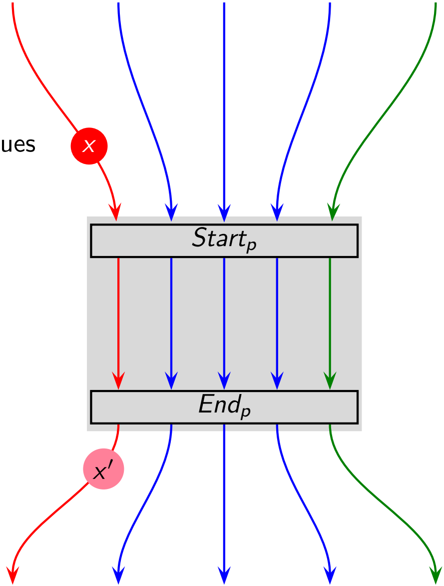
Value Contexts



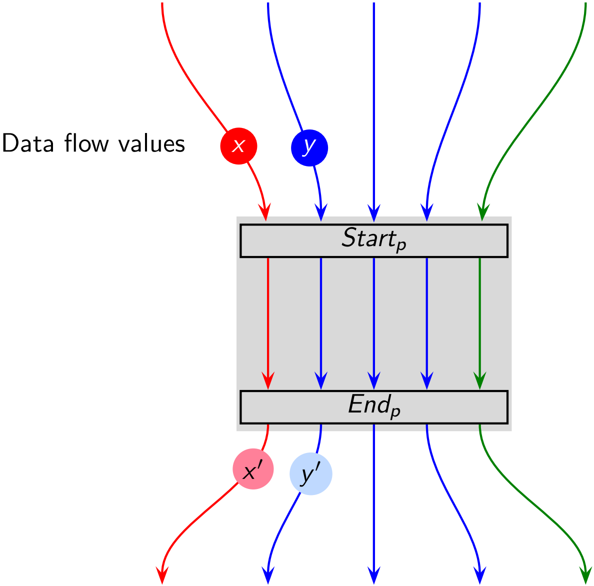
Multiple
interprocedural
paths reaching
the procedure

Value Contexts

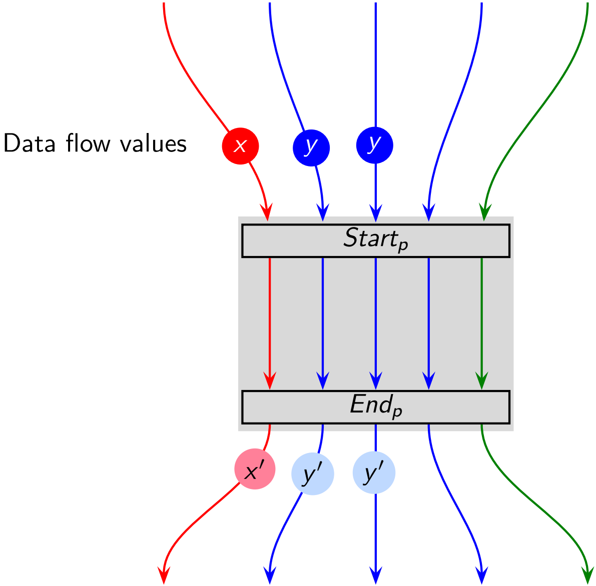
Data flow values



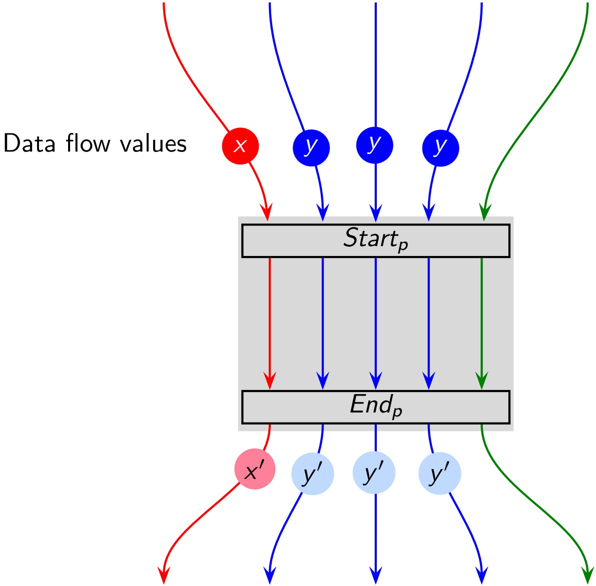
Value Contexts



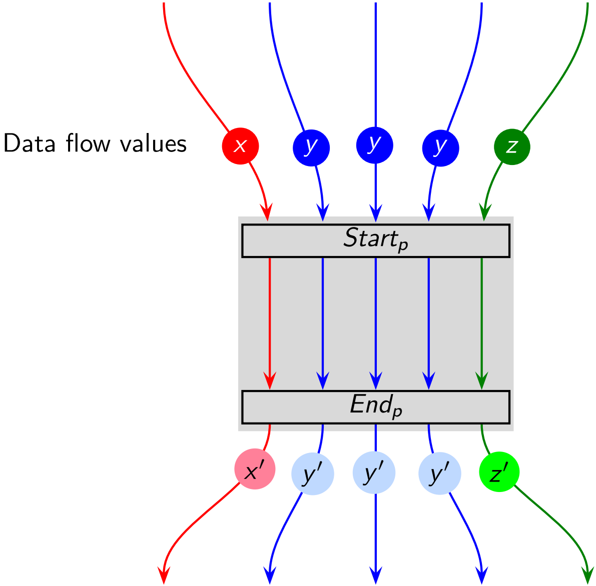
Value Contexts



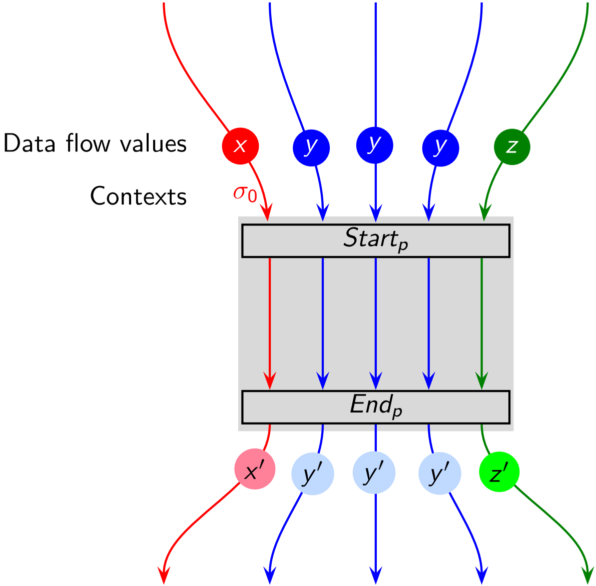
Value Contexts



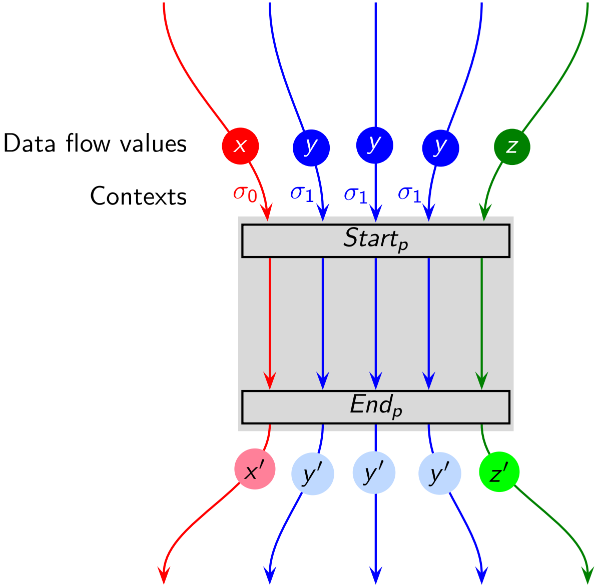
Value Contexts



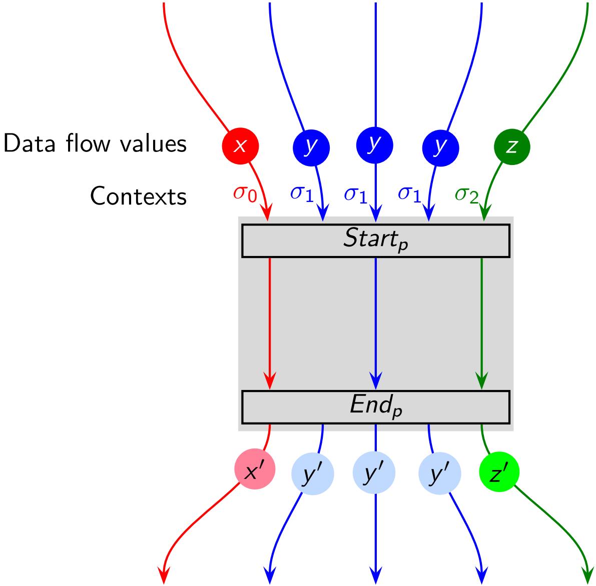
Value Contexts



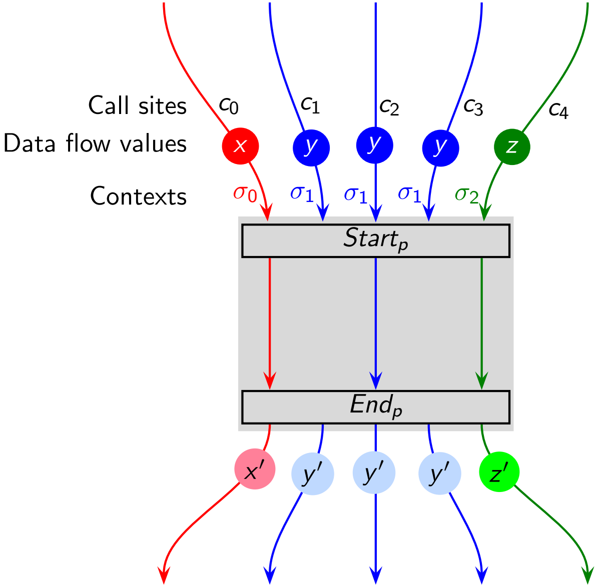
Value Contexts



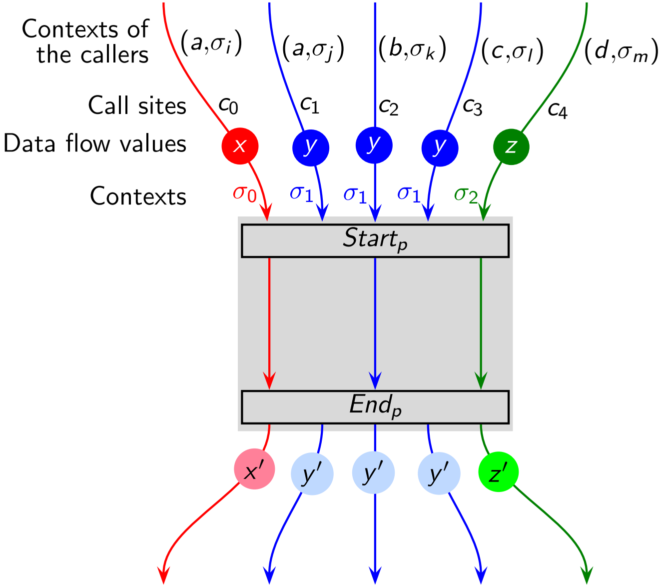
Value Contexts



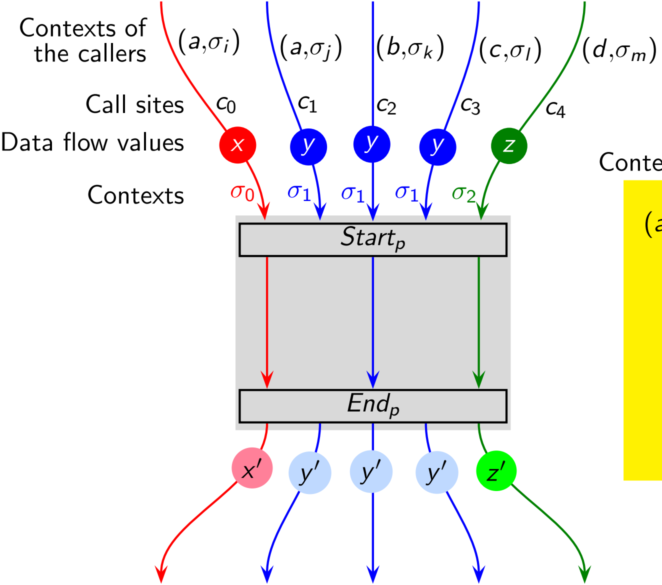
Value Contexts



Value Contexts



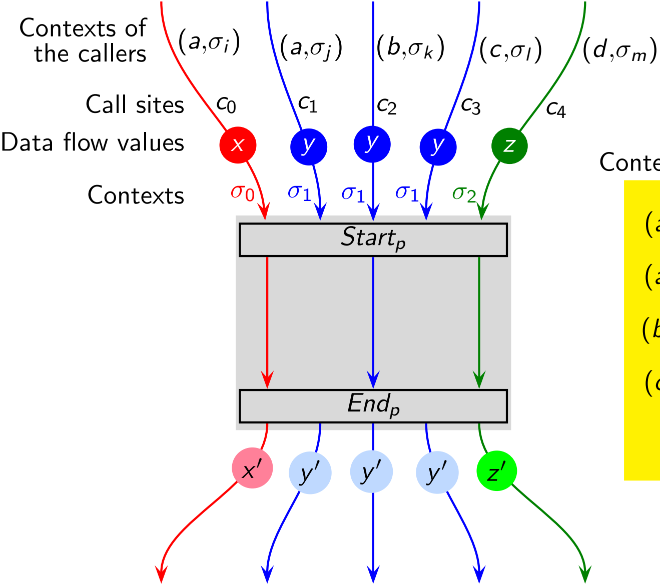
Value Contexts



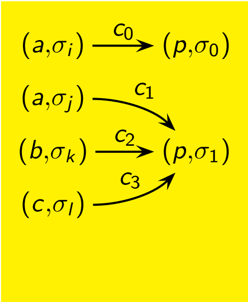
Context-sensitive call graph

$$(a,\sigma_i) \xrightarrow{c_0} (p,\sigma_0)$$

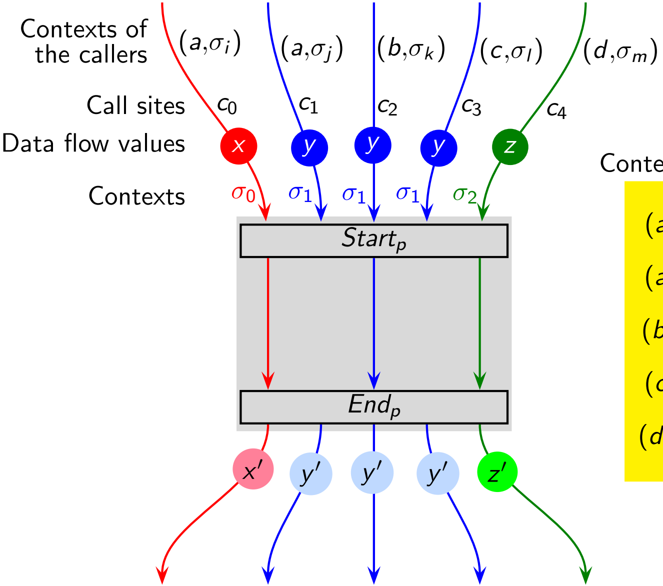
Value Contexts



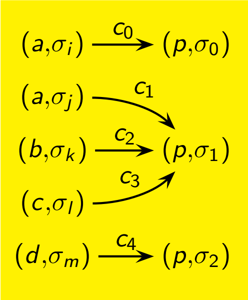
Context-sensitive call graph



Value Contexts



Context-sensitive call graph



Value Contexts

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]

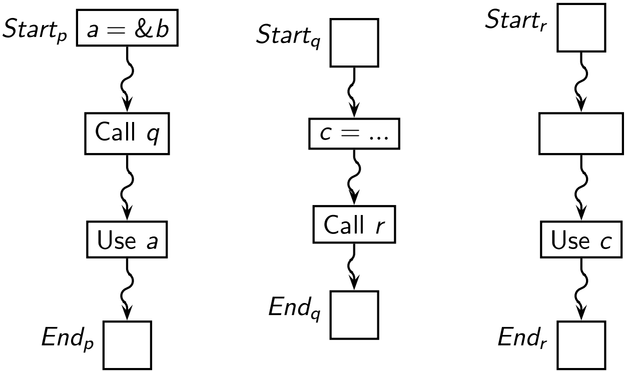
  However,

  - 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

Observations About Value Contexts

- The number of contexts reduce significantly
- Much fewer call chains need to be considered
- And yet, it is insufficient for scaling flow- and context-sensitive points-to analysis to more than 35 kLoC

Top-down Analysis With Bypassing



- Procedures q and r do not access a
- Can we avoid progapating the points-to pair (a, b) *through* procedure q (and hence through r)?
- How do we know which pairs should *bypass* a call?

  Compute the bypassing set for each procedure during the analysis

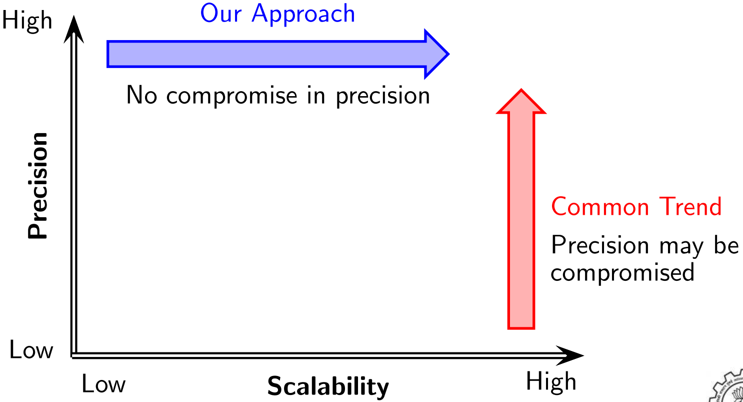
An Outline of Research Explorations in Interprocedural Analysis

- Broad categories of interprocedural analysis
- Scaling top-down analysis using value contexts and bypassing
- Improving bottom-up analysis by eliminating control flow Next Topic

Our Approach

Improve the scalability of *exhaustive* pointer analysis without losing precision

- Construct sound and precise but compact statement level summaries
- Combine them naively and optimize for scalability without compromising soundness or precision



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

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

$\Downarrow$

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

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. $x = \&b;$

$\Downarrow$

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

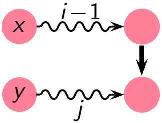
Data dependence is unknown ⇒
  More information is required (available in callers)
  Control flow between the updates is required

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

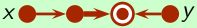
Generalized Points-to Updates (GPUs)

General Case

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



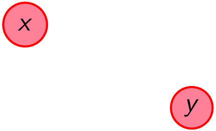
Specific Examples

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 $j = 0$

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

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

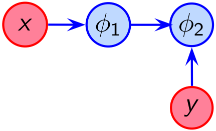


All variables are global

Red nodes are known named locations

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

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



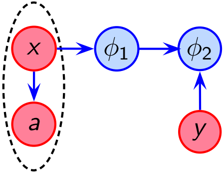
All variables are global

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
}
```



Information from callers

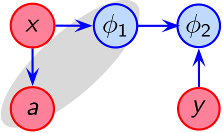
All variables are global

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
}
```



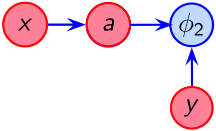
All variables are global

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
}
```



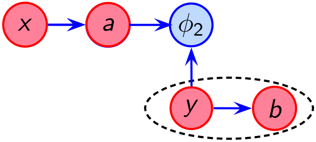
All variables are global

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
}
```



Information from callers

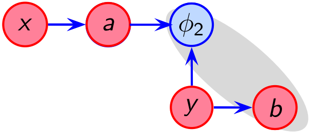
All variables are global

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
}
```



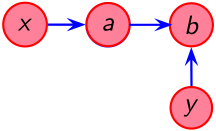
All variables are global

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
}
```



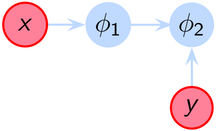
All variables are global

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

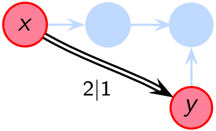
```
f()
{
  *x = y
}
```



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
}
```

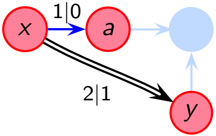


Blue arrows are low level view of memory in terms of classical points-to facts

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
}
```

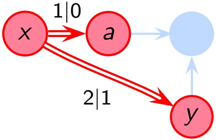


Blue arrows are low level view of memory in terms of classical points-to facts

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
}
```

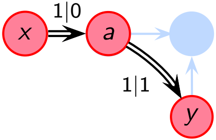


Blue arrows are low level view of memory in terms of classical points-to facts

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
}
```

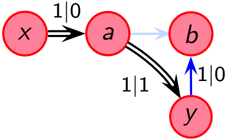


Blue arrows are low level view of memory in terms of classical points-to facts

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
}
```

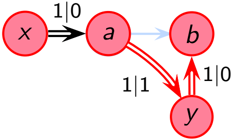


Blue arrows are low level view of memory in terms of classical points-to facts

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
}
```

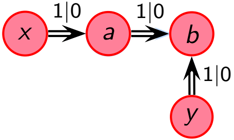


Blue arrows are low level view of memory in terms of classical points-to facts

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
}
```

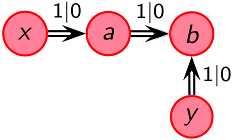


Blue arrows are low level view of memory in terms of classical points-to facts

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
}
```

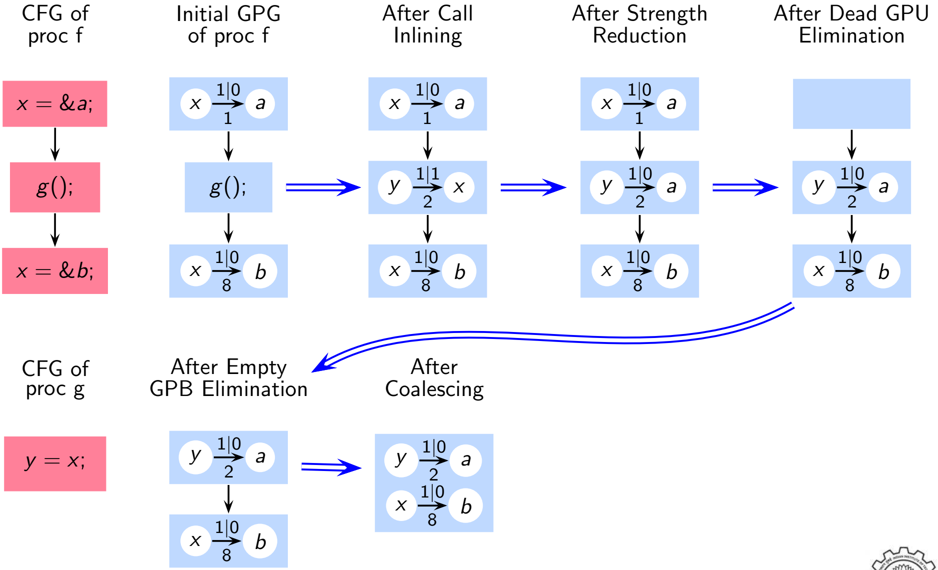


Blue arrows are low level view of memory in terms of classical points-to facts

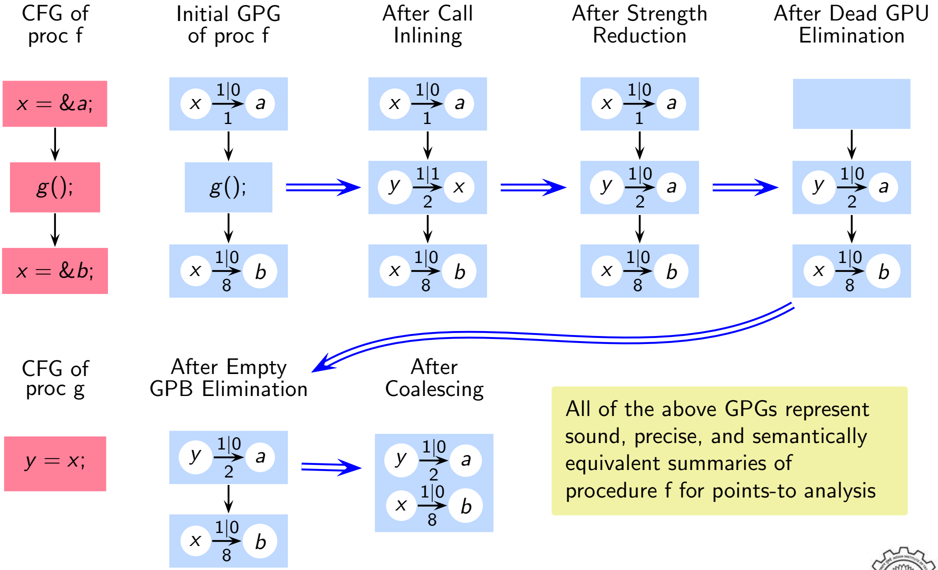
Black arrows are high level view of memory in terms of generalized points-to facts

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

GPGs Across Optimizations



GPGs Across Optimizations



All of the above GPGs represent sound, precise, and semantically equivalent summaries of procedure f for points-to analysis

Part 4

Conclusion

Observations

- Relevant pointer information in a program is very small and sparse
- 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!
- Precision of analysis can be improved by
  - Excluding infeasible control flow paths
  - Interleaving program analyses

Observations

- 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

Observations

- 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
- We achieve this without compromising soundness or precision