The Abstraction Vs. Approximations Dilemma in Pointer Analysis

Uday Khedker

(www.cse.iitb.ac.in/˜uday)

Department of Computer Science and Engineering,

Indian Institute of Technology, Bombay

June 2019

Outline

- Disclaimer: This talk is
  - not about accomplishments but about opinions, and hopes
  - an idealistic view of pointer analysis
    (the destination we wish to reach)

Outline

- Disclaimer: This talk is
  - not about accomplishments but about opinions, and hopes
  - an idealistic view of pointer analysis
    (the destination we wish to reach)
- Outline:
  - Our Meanderings
  - Some short trips
  - Conclusions

Part 1

Our Meanderings

Pointer Analysis Musings

- A keynote address:

  > "The worst thing that has happened to Computer Science is C, because it brought pointers with it . . . "
  >
  > - Frances Allen, IITK Workshop (2007)

- A couple of influential papers
  - Which Pointer Analysis should I Use?

    Michael Hind and Anthony Pioli. ISTAA 2000
  - Pointer Analysis: Haven't we solved this problem yet ?

    Michael Hind PASTE 2001

Pointer Analysis Musings

- A keynote address:

  "The worst thing that has happened to Computer Science is C, because it brought pointers with it ..."

  - Frances Allen, IITK Workshop (2007)

- A couple of influential papers
  - Which Pointer Analysis should I Use?

    Michael Hind and Anthony Pioli. ISTAA 2000
  - Pointer Analysis: Haven't we solved this problem yet?

    Michael Hind PASTE 2001

Pointer Analysis Musings

- A keynote address:

  "The worst thing that has happened to Computer Science is C, because it brought pointers with it . . ."

  - Frances Allen, IITK Workshop (2007)

- A couple of influential papers
  - Which Pointer Analysis should I Use?

    Michael Hind and Anthony Pioli. ISTAA 2000
  - Pointer Analysis: Haven't we solved this problem yet ?

    Michael Hind PASTE 2001
  - 2017 . . . ☹

The Mathematics of Pointer Analysis

In the most general situation

- Alias analysis is undecidable.
  Landi-Ryder [POPL 1991], Landi [LOPLAS 1992], Ramalingam [TOPLAS 1994]
- Flow insensitive alias analysis is NP-hard
  Horwitz [TOPLAS 1997]
- Points-to analysis is undecidable
  Chakravarty [POPL 2003]

Adjust your expectations suitably to avoid disappointments!

So what should we expect?

To quote Hind [PASTE 2001]

So what should we expect?

To quote Hind [PASTE 2001]

- “Fortunately many approximations exist”

So what should we expect?

To quote Hind [PASTE 2001]

- "Fortunately many approximations exist"
- "Unfortunately too many approximations exist!"

So what should we expect?

To quote Hind [PASTE 2001]

- "Fortunately many approximations exist"
- "Unfortunately too many approximations exist!"

Engineering of pointer analysis is much more dominant than its science

Pointer Analysis: Engineering or Science?

- Engineering view
  - Build quick approximations
  - The tyranny of (exclusive) OR
    Precision OR Efficiency?
- Science view
  - Build clean abstractions
  - Can we harness the Genius of AND?
    Precision AND Efficiency?

Pointer Analysis: Engineering or Science?

- Engineering view
  - Build quick approximations
  - The tyranny of (exclusive) OR
    Precision OR Efficiency?
- Science view
  - Build clean abstractions
  - Can we harness the Genius of AND?
    Precision AND Efficiency?
- Most common trend as evidenced by publications
  - Build acceptable approximations guided by empirical observations
  - The notion of acceptability is often constrained by beliefs rather than possibilities

Abstraction Vs. Approximation in Static Analysis

- Static analysis needs to create abstract values that represent many concrete values
- Mapping concrete values to abstract values
  - *Abstraction*.
    Deciding which properties of the concrete values are essential *What*
    Ease of understanding, reasoning, modelling etc. *Why*
  - *Approximation*.
    Deciding which properties of the concrete values cannot be represented accurately and should be summarized *What*
    Decidability, tractability, or efficiency and scalability *Why*

Abstraction Vs. Approximation in Static Analysis

- Abstractions
  - focus on precision and conciseness of modelling
  - tell us what we can ignore without being imprecise
- Approximations
  - focus on efficiency and scalability
  - tell us the imprecision that we have to tolerate

Abstraction Vs. Approximation in Static Analysis

- Abstractions
  - focus on precision and conciseness of modelling
  - tell us what we can ignore without being imprecise
- Approximations
  - focus on efficiency and scalability
  - tell us the imprecision that we have to tolerate
- *Build clean abstractions before surrendering to the approximations*

The Basis of My Hope

- Common belief:
- The possibility that I dream of:
- The basis of my hope:

The Basis of My Hope

- Common belief:

  Pointer information is very large

- The possibility that I dream of:

- The basis of my hope:

The Basis of My Hope

- Common belief:

  Pointer information is very large

- The possibility that I dream of:

  Precision can reduce the size of pointer information to make it far more manageable

- The basis of my hope:

The Basis of My Hope

- Common belief:

  Pointer information is very large

- The possibility that I dream of:

  Precision can reduce the size of pointer information to make it far more manageable

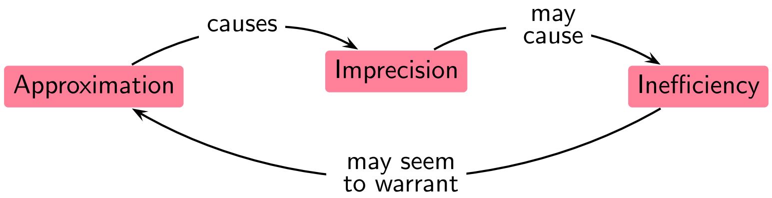
- The basis of my hope:

  At any program point, the usable pointer information is much smaller than the total pointer information

  Current methods perform many repeated and possibly avoidable computations

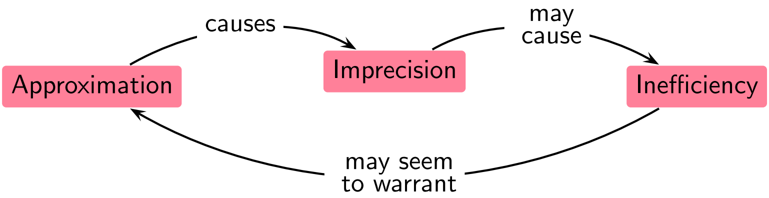
Why Avoid Approximations?

- Approximations may create a vicious cycle



Why Avoid Approximations?

- Approximations may create a vicious cycle



- Two examples of inefficiency cause by approximations
  - *k*-limited call strings may create "butterfly cycles" causing spurious fixed point computations [Hakjoo, 2010]
  - Imprecision in function pointer analysis overapproximates calls may create spurious recursion in call graphs

Which Approximations Would I like to Avoid?

Approximation	Admits
Flow insensitivity	
Context insensitivity (or partial context sensitivity)	
Imprecision in call graphs	
Allocation site based heap abstraction	

Which Approximations Would I like to Avoid?

Approximation	Admits
Flow insensitivity	Spurious intraprocedural paths
Context insensitivity (or partial context sensitivity)	Spurious interprocedural paths
Imprecision in call graphs	Spurious call sequences
Allocation site based heap abstraction	Spurious paths in memory graph

The Classical Precision-Efficiency Dilemma

Abstraction	Role in precision	Cause of inefficiency
	Distinguishes between	Needs to consider
Flow sensitivity		
Context sensitivity		
Precise heap abstraction		
Precise call structure		

The Classical Precision-Efficiency Dilemma

Abstraction	Role in precision	Cause of inefficiency
	Distinguishes between	Needs to consider
Flow sensitivity	Information at different program points	
Context sensitivity	Information in different contexts	
Precise heap abstraction	Different heap locations	
Precise call structure	Indirect calls made to different callees from the same program point	

The Classical Precision-Efficiency Dilemma

Abstraction	Role in precision	Cause of inefficiency
	Distinguishes between	Needs to consider
Flow sensitivity	Information at different program points	A large number of program points
Context sensitivity	Information in different contexts	Exponentially large number of contexts
Precise heap abstraction	Different heap locations	Unbounded number of heap locations
Precise call structure	Indirect calls made to different callees from the same program point	Precise points-to information

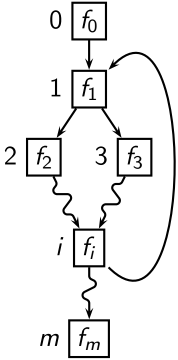
Flow Insensitivity in Data Flow Analysis

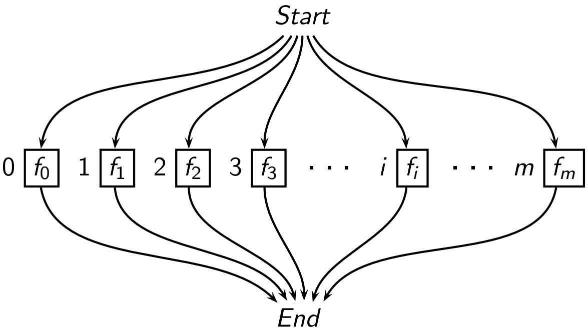
- Assumption: Statements can be executed in any order.
- Instead of computing point-specific data flow information, summary data flow information is computed.

  The summary information is required to be a safe approximation of point-specific information for each point.
- No data flow information is killed

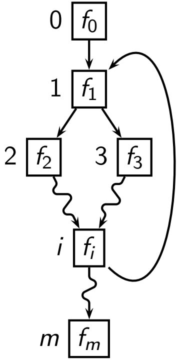
  If a statement kills data flow information, there is an alternate path that excludes the statement.

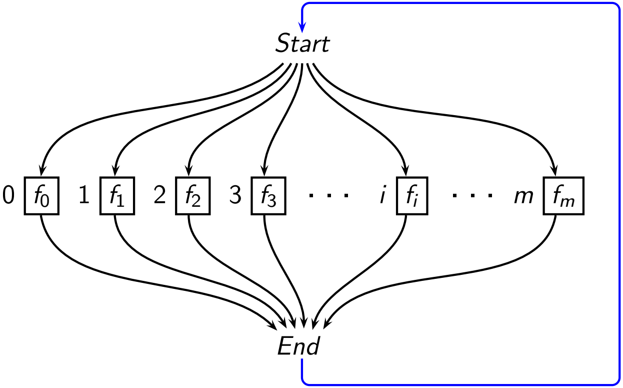
Flow Insensitivity in Data Flow Analysis



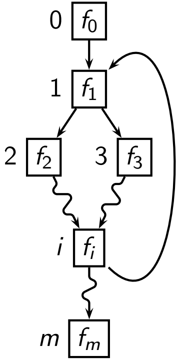


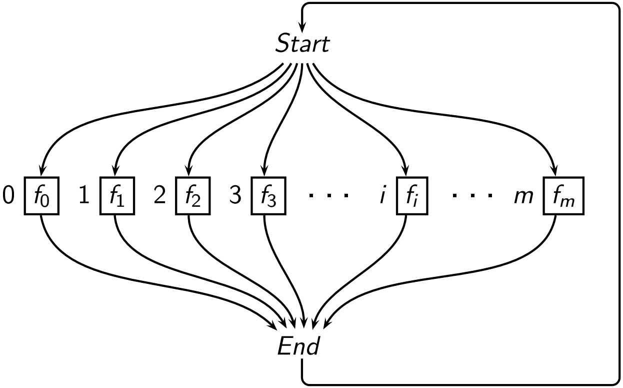
Flow Insensitivity in Data Flow Analysis





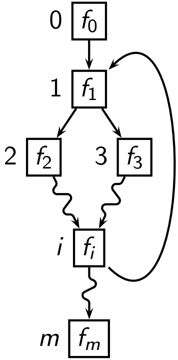
Flow Insensitivity in Data Flow Analysis

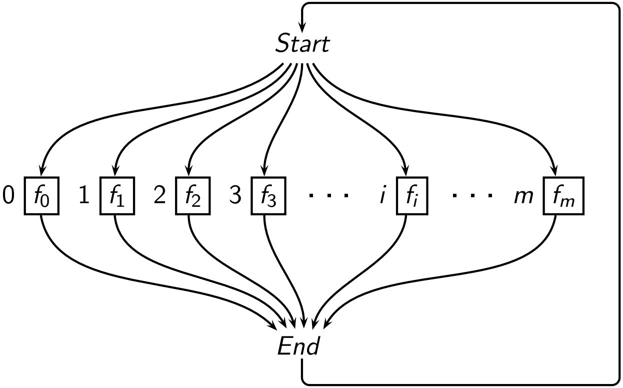




Allows arbitrary compositions of flow functions in any order
$\Rightarrow$ *Flow insensitivity*

Flow Insensitivity in Data Flow Analysis





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

If I am Allowed to Nitpick ...

- Context sensitivity should involve all of the following
  - [A] Full context sensitivity regardless of the call depth even in recursion
  - [B] Ability to store data flow information parameterized by contexts at each program point
  - [C] Flow sensitivity at the intraprocedural level (otherwise distinct calls to the same procedure within a procedure cannot be distinguished)

If I am Allowed to Nitpick . . .

- Context sensitivity should involve all of the following

  [A] Full context sensitivity regardless of the call depth even in recursion
  [B] Ability to store data flow information parameterized by contexts at each program point
  [C] Flow sensitivity at the intraprocedural level (otherwise distinct calls to the same procedure within a procedure cannot be distinguished)

- In particular
  - *k*-limiting violates [A]
  - Treating recursion as an SCC violates [A]
  - Functional approaches violate [B]
  - Object sensitivity violates [C]

If I am Allowed to Nitpick ...

- Context sensitivity should involve all of the following

  [A] Full context sensitivity regardless of the call depth even in recursion
  [B] Ability to store data flow information parameterized by contexts at each program point
  [C] Flow sensitivity at the intraprocedural level (otherwise distinct calls to the same procedure within a procedure cannot be distinguished)

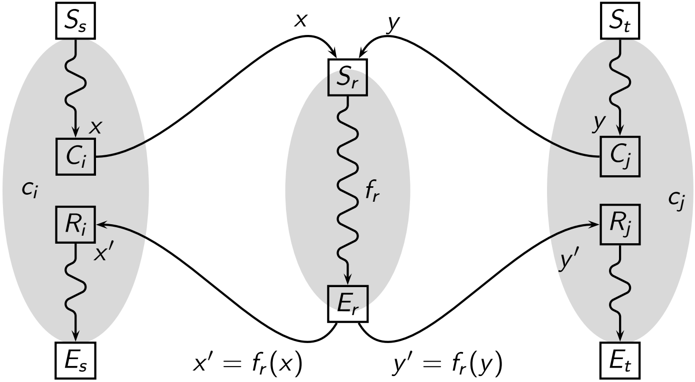
- In particular

  - k-limiting violates [A]
  - Treating recursion as an SCC violates [A]
  - Functional approaches violate [B]
  - Object sensitivity violates [C]

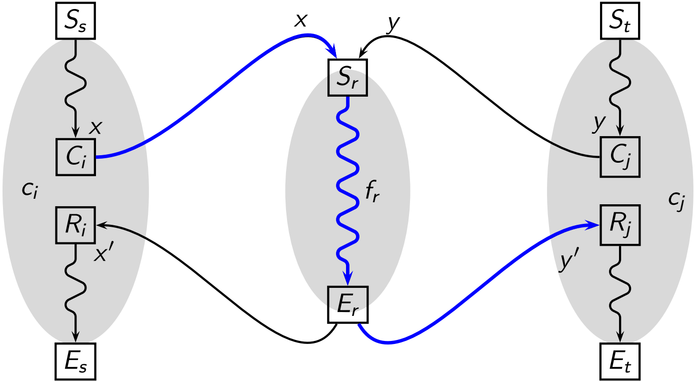
- Object sensitivity can be completely modelled by calling context sensitivity

  - by a flow sensitive propagation of values representing objects, and
  - identifying a procedure by an (object, procedure) pair, and
  - identifying a context by a call site and the pairs defined as above

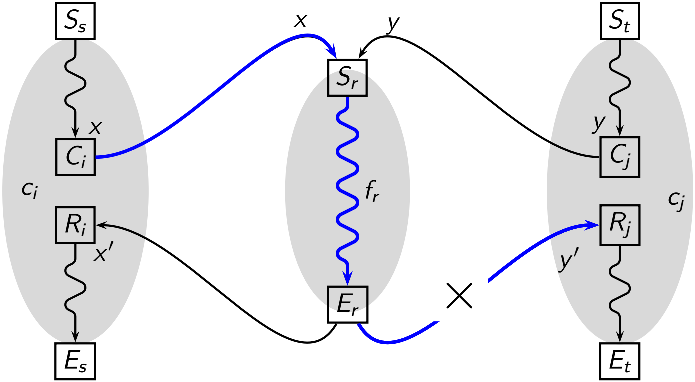
Context Sensitivity in Interprocedural Analysis



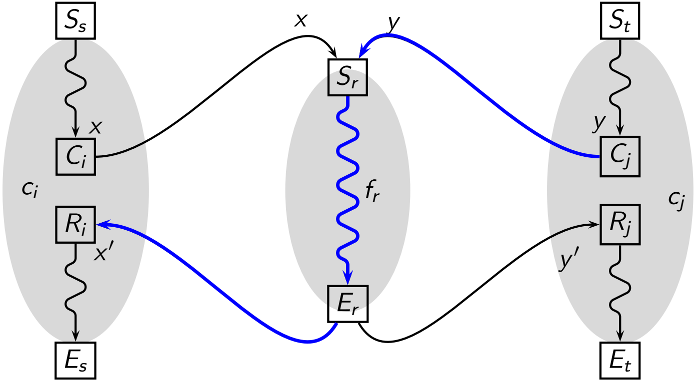
Context Sensitivity in Interprocedural Analysis



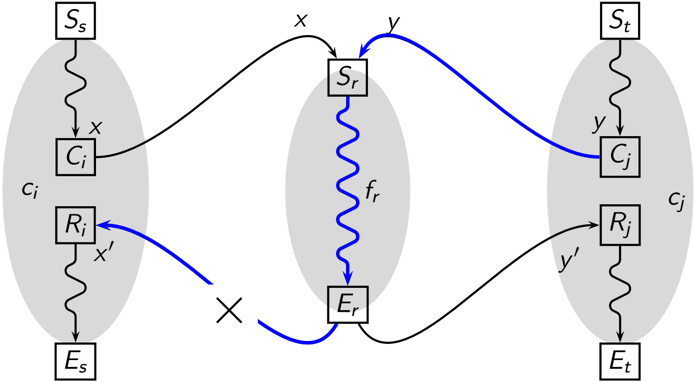
Context Sensitivity in Interprocedural Analysis



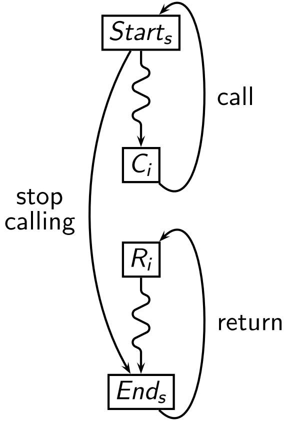
Context Sensitivity in Interprocedural Analysis



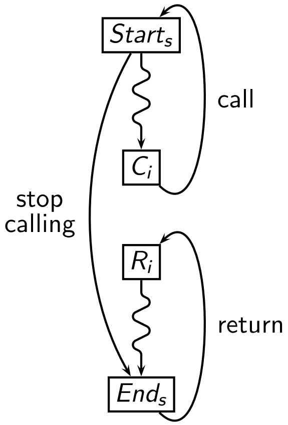
Context Sensitivity in Interprocedural Analysis



Context Sensitivity in the Presence of Recursion



Context Sensitivity in the Presence of Recursion



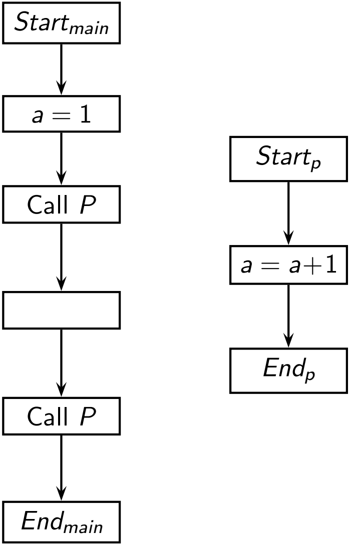
- Paths from $Start_s$ to End_s should constitute a context free language

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

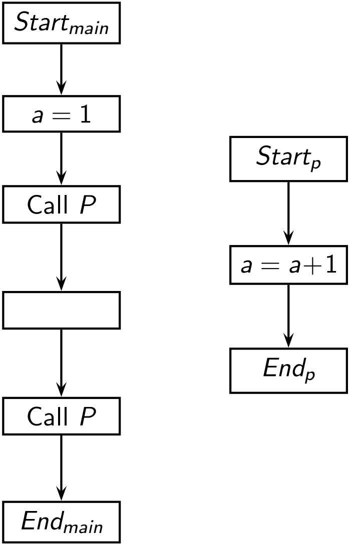
- If we treat cycle of recursion as an SCC
  - Calls and returns become jumps, and
  - paths are approximated by a regular language

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

Context Insensitivity = Imprecision + Potential Inefficiency

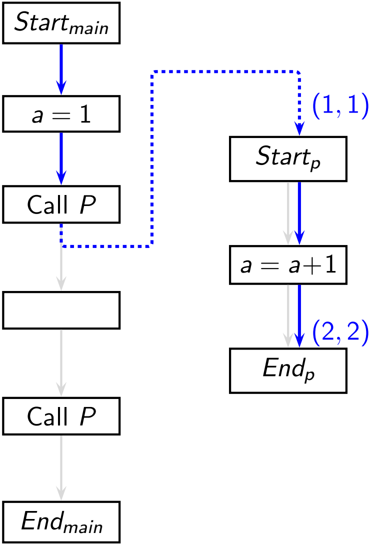


Context Insensitivity = Imprecision + Potential Inefficiency



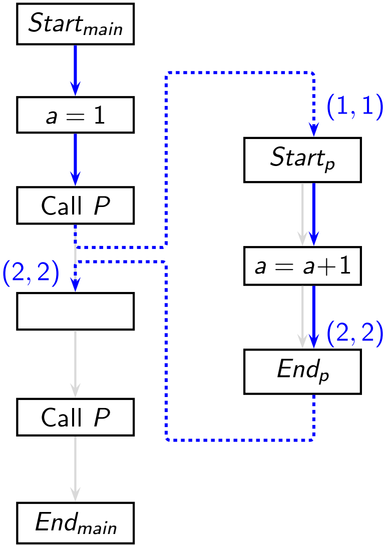
- What is the value range of a?

Context Insensitivity = Imprecision + Potential Inefficiency



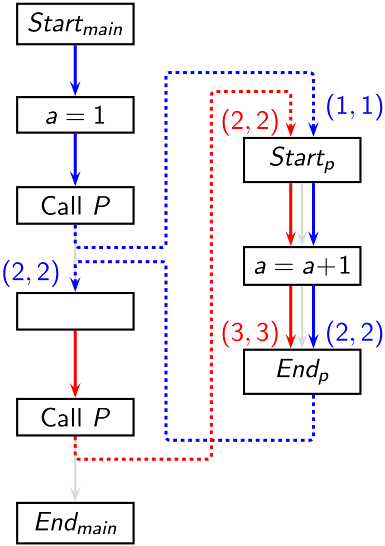
- What is the value range of a?

Context Insensitivity = Imprecision + Potential Inefficiency



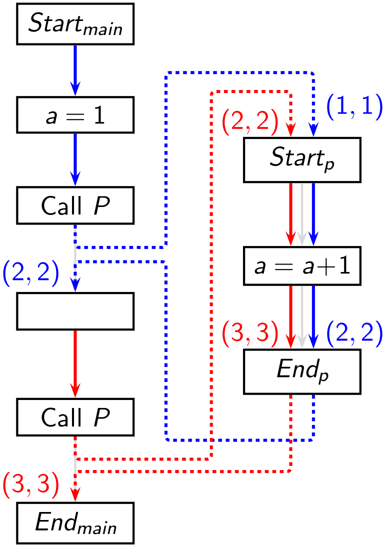
- What is the value range of a?
- Context sensitive analysis
    - Data flow value propagated back to the current caller of P

Context Insensitivity = Imprecision + Potential Inefficiency



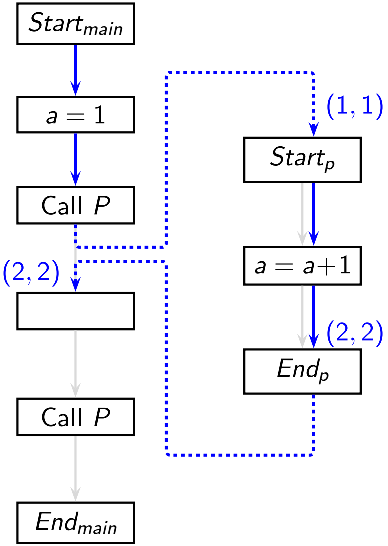
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P

Context Insensitivity = Imprecision + Potential Inefficiency



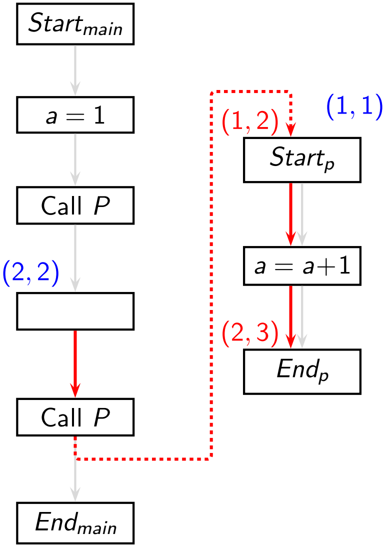
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3, 3)$

Context Insensitivity = Imprecision + Potential Inefficiency



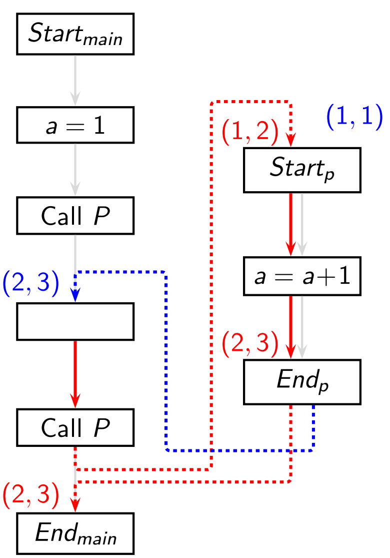
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3, 3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller

Context Insensitivity = Imprecision + Potential Inefficiency



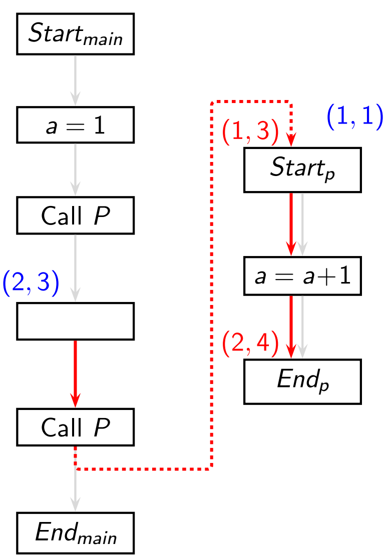
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3, 3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller

Context Insensitivity = Imprecision + Potential Inefficiency



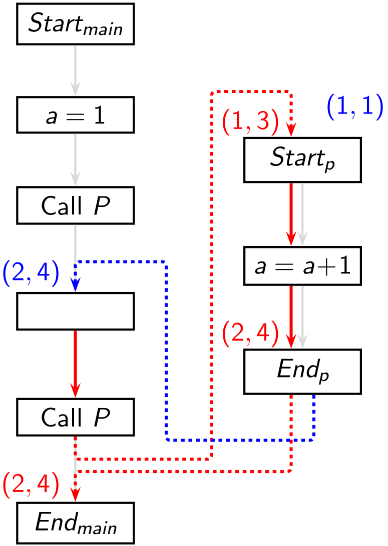
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3, 3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller

Context Insensitivity = Imprecision + Potential Inefficiency



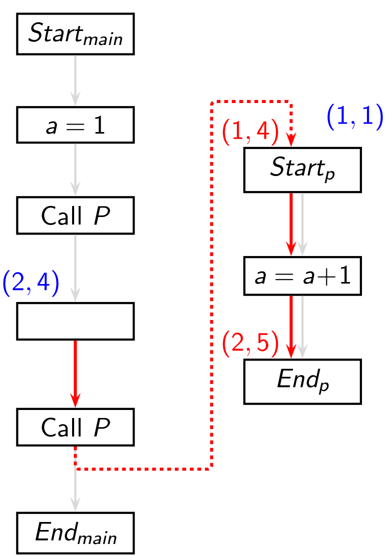
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3, 3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller

Context Insensitivity = Imprecision + Potential Inefficiency



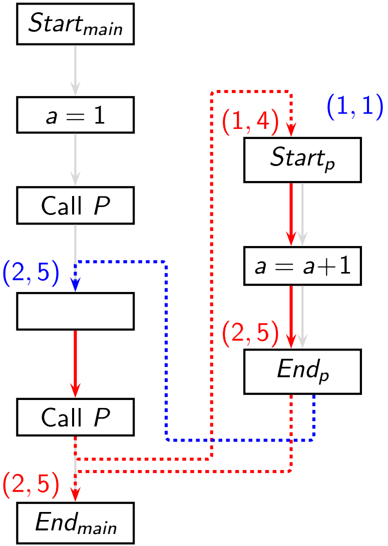
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3,3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller

Context Insensitivity = Imprecision + Potential Inefficiency



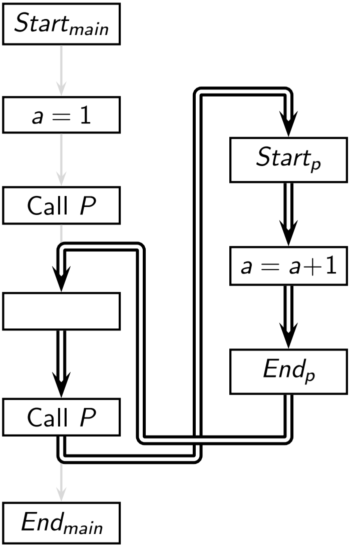
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3, 3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller

Context Insensitivity = Imprecision + Potential Inefficiency



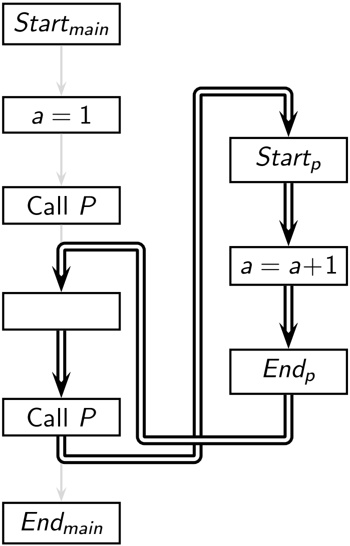
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3, 3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller
  - Range of a at End_{main} is $(2, \ldots)$

Context Insensitivity = Imprecision + Potential Inefficiency



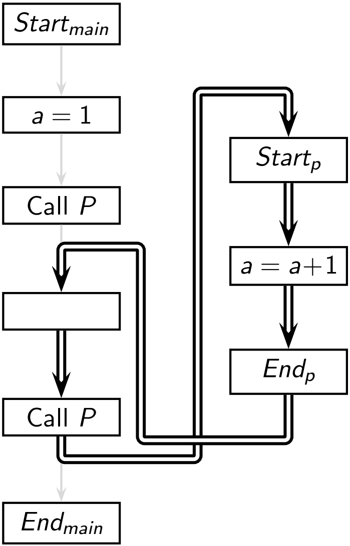
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3, 3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller
  - Range of a at End_{main} is $(2, \ldots)$

Context Insensitivity = Imprecision + Potential Inefficiency



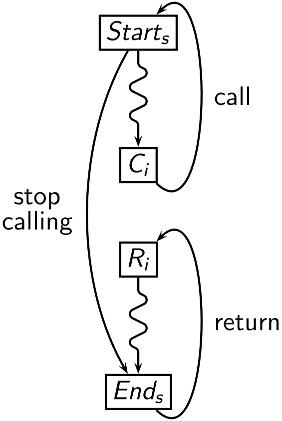
- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3, 3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller
  - Range of a at End_{main} is $(2, \ldots)$
- *Spurious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

Context Insensitivity = Imprecision + Potential Inefficiency

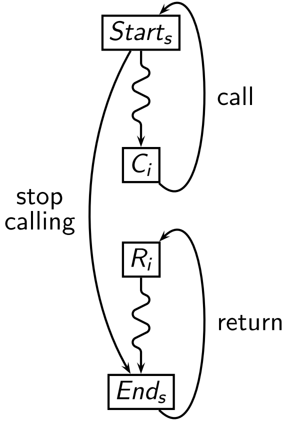


- What is the value range of a?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of P
  - Range of a at End_{main} is $(3, 3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller
  - Range of a at End_{main} is $(2, \dots)$
- *Spurious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

Context Sensitivity in the Presence of Recursion



Context Sensitivity in the Presence of Recursion



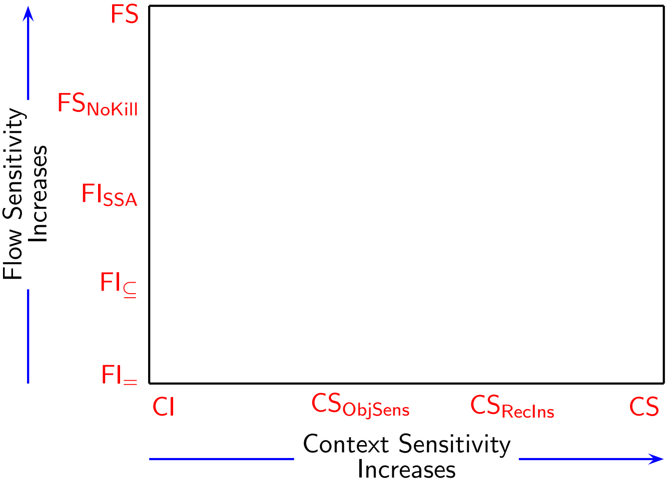
- Paths from $Start_s$ to End_s should constitute a context free language

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

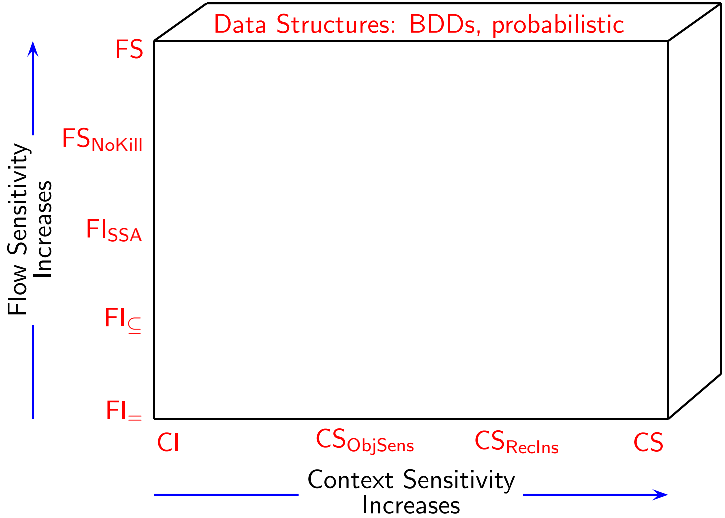
- If we treat cycle of recursion as an SCC
    - Calls and returns become jumps, and
    - paths are approximated by a regular language

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

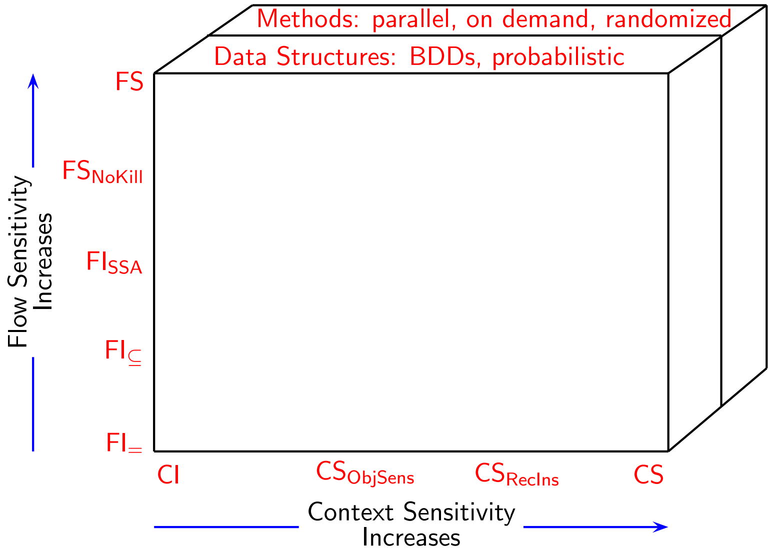
Pointer Analysis: An Engineer's Landscape



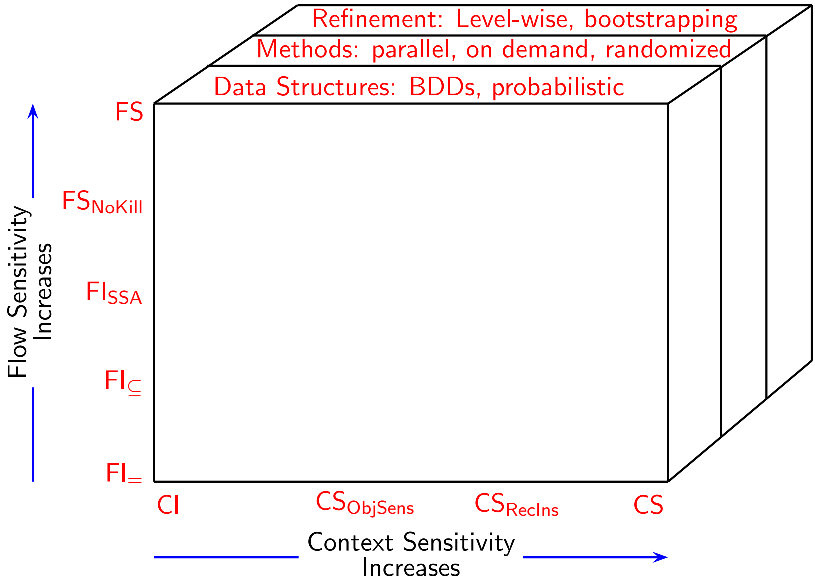
Pointer Analysis: An Engineer's Landscape



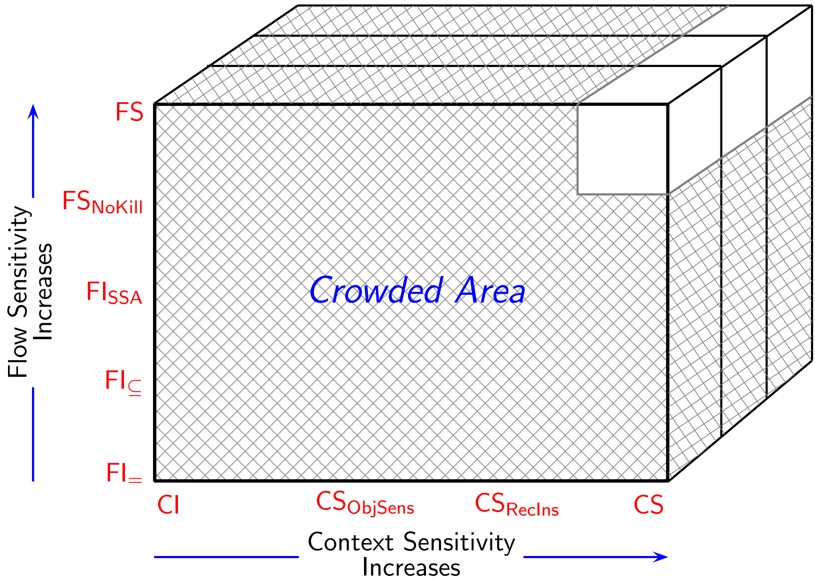
Pointer Analysis: An Engineer's Landscape



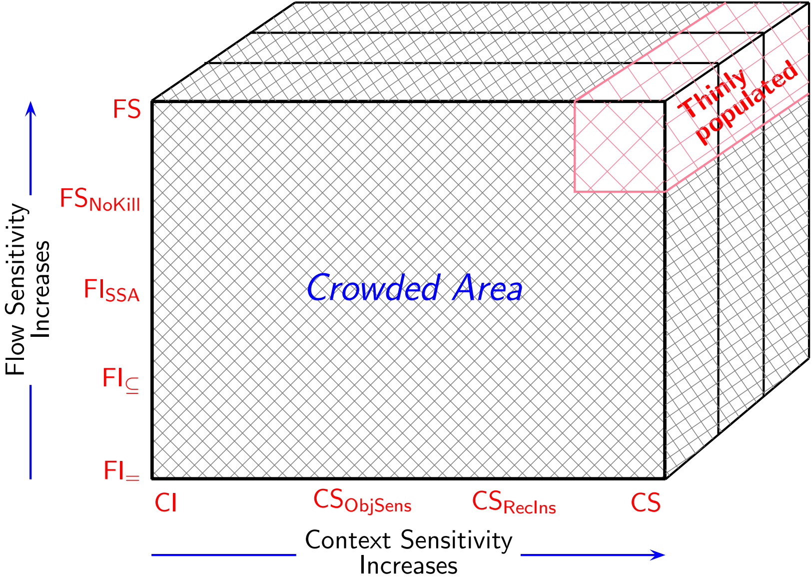
Pointer Analysis: An Engineer's Landscape



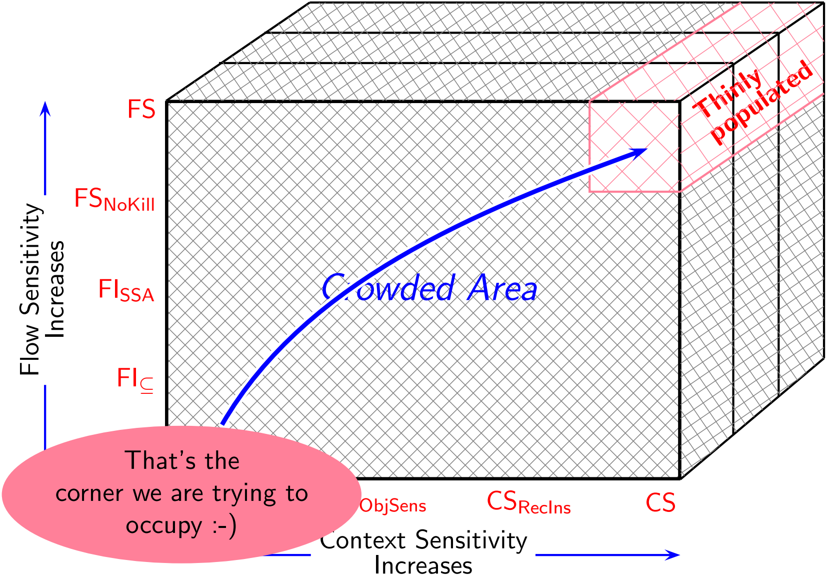
Pointer Analysis: An Engineer's Landscape



Pointer Analysis: An Engineer's Landscape



Pointer Analysis: An Engineer's Landscape



In Search of Abstractions for Precision Without Inefficiency

Desired Abstraction	Enabling Abstraction	Status of our work
Flow sensitivity		
Context sensitivity (Caller sensitivity)		
Precise heap abstraction		
Precise call structure		

In Search of Abstractions for Precision Without Inefficiency

Desired Abstraction	Enabling Abstraction	Status of our work
Flow sensitivity	Joint liveness and points-to analysis	Partial accomplishment (SAS12)
Context sensitivity (Caller sensitivity)		
Precise heap abstraction		
Precise call structure		

Restrict the computation only to the usable data.
Weave liveness discovery into the analysis

In Search of Abstractions for Precision Without Inefficiency

Desired Abstraction	Enabling Abstraction	Status of our work
Flow sensitivity	Joint liveness and points-to analysis	Partial accomplishment (SAS12)
	High level abstraction of memory	Partial accomplishment (SAS16)
Context sensitivity (Caller sensitivity)		
Precise heap abstraction		
Precise call structure		

Postpone low level connections explicated by the classical points-to facts

In Search of Abstractions for Precision Without Inefficiency

Desired Abstraction	Enabling Abstraction	Status of our work
Flow sensitivity	Joint liveness and points-to analysis	Partial accomplishment (SAS12)
	High level abstraction of memory	Partial accomplishment (SAS16)
Context sensitivity (Caller sensitivity)	Value contexts	Mature accomplishment (CC08, SAS12, SOAP13)
Precise heap abstraction		
Precise call structure		

Distinguish between contexts by their data flow values and not their call chains

In Search of Abstractions for Precision Without Inefficiency

Desired Abstraction	Enabling Abstraction	Status of our work
Flow sensitivity	Joint liveness and points-to analysis	Partial accomplishment
	High level abstrac of memory	ent
Context sensitivity (Caller sensitivity)	Value contexts	ent P13)
	GPG based bottom-up summary flow functions	Mature accomplishment (SAS16)
Precise heap abstraction		
Precise call structure		

Avoid recomputations for each context.
Use a higher level abstraction of memory.

In Search of Abstractions for Precision Without Inefficiency

Desired Abstraction	Enabling Abstraction	Status of our work
Flow sensitivity	Joint liveness and points-to analysis	Partial accomplishment (SAS12)
	High level abstraction of memory	Partial accomplishment (SAS16)
Context sensitivity (Caller sensitivity)	Value contexts	ent P13)
	GPG based bo summary flow	ent
Precise heap abstraction	Liveness access graphs	Partial accomplishment (TOPLAS07)
Precise call structure		

Identify the part of heap actually accessed in terms of patterns of accesses

In Search of Abstractions for Precision Without Inefficiency

Desired Abstraction	Enabling Abstraction	Status of our work
Flow sensitivity	Joint liveness and points-to analysis	Partial accomplishment (SAS12)
	High level abstraction of memory	Partial accomplishment (SAS16)
Context sensitivity (Caller sensitivity)	Value contexts	Mature accomplishment P13)
	GPG based bo summary flow	ent
Precise heap abstraction	Liveness acces graphs	ent
	Access based abstraction	Mature accomplishment (ISMM17)
Precise call structure		

Distinguish between heap locations based on how they are accessed and not how they are allocated

In Search of Abstractions for Precision Without Inefficiency

Desired Abstraction	Enabling Abstraction	Status of our work
Flow sensitivity	Joint liveness and points-to analysis	Partial accomplishment (SAS12)
	High level abstraction of memory	Partial accomplishment (SAS16)
Context sensitivity (Caller sensitivity)	Value contexts	Mature accomplishment (CC08, SAS12, SOAP13)
	GPG based bottom-up summary flow functions	Mature accomplishment (SAS16)
Precise heap abstraction	Liveness access graphs	ent
	Access based abstraction	ent
Precise call structure	Callee sensitivity	Work in progress

Call strings record call *history*. We need to record call *future* also.

In Search of Abstractions for Precision Without Inefficiency

Desired Abstraction	Enabling Abstraction	Status of our work
Flow sensitivity	Joint liveness and points-to analysis	Partial accomplishment (SAS12)
	High level abstraction of memory	Partial accomplishment (SAS16)
Context sensitivity (Caller sensitivity)	Value contexts	Mature accomplishment (CC08, SAS12, SOAP13)
	GPG based bottom-up summary flow functions	Mature accomplishment (SAS16)
Precise heap abstraction	Liveness acces graphs	ent
	Access based abstraction	ent
Precise call structure	Callee sensitivity	Work in progress
	Virtual call resolution	Work in progress

Make the call graph more precise by computing a more precise set of callees

In Search of Abstractions for Precision Without Inefficiency

Desired Abstraction	Enabling Abstraction	Status of our work
Flow sensitivity	Joint liveness and points-to analysis	Partial accomplishment (SAS12)
	High level abstraction of	Partial accomplishment (SAS16)
Context sensitivi (Caller sensitivi		ture accomplishment , SAS12, SOAP13)
		accomplishment)
Precise heap abstraction		l accomplishment OPLAS07)
	Acce abstraction	Mature accomplishment (ISMM17)
Precise call structure	Callee sensitivity	Work in progress
	Virtual call resolution	Work in progress

We are destined to a long haul with no guarantees :-)

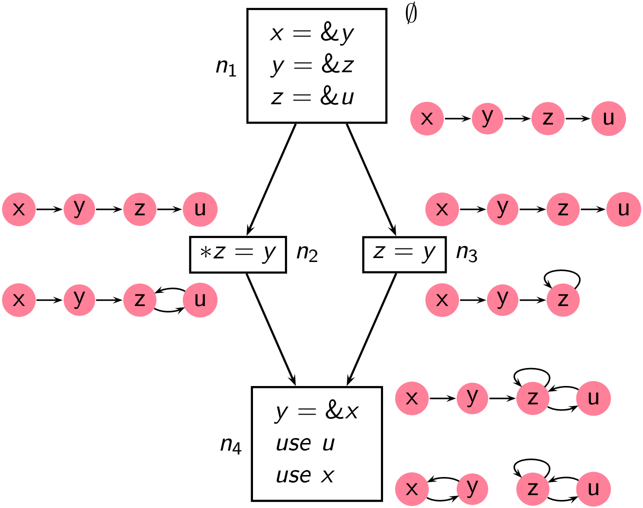
Part 2

Some Short Trips

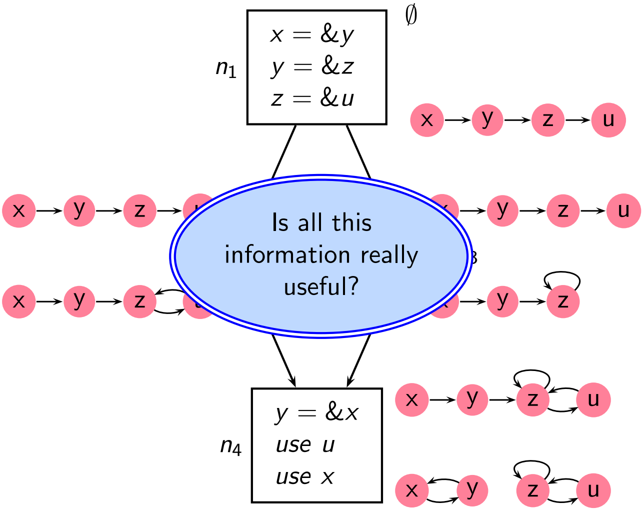
In Search of Abstractions for Precision Without Inefficiency

Desired Abstraction	Enabling Abstraction	Status of our work
Flow sensitivity	Joint liveness and points-to analysis	Partial accomplishment (SAS12)
	High level abstraction of memory	Partial accomplishment (SAS16)
Context sensitivity (Caller sensitivity)	Value contexts	Mature accomplishment (CC08, SAS12, SOAP13)
	GPG based bottom-up summary flow functions	Mature accomplishment (SAS16)
Precise heap abstraction	Liveness access graphs	Partial accomplishment (TOPLAS07)
	Access based abstraction	Mature accomplishment (ISMM17)
Precise call structure	Callee sensitivity	Work in progress
	Virtual call resolution	Work in progress

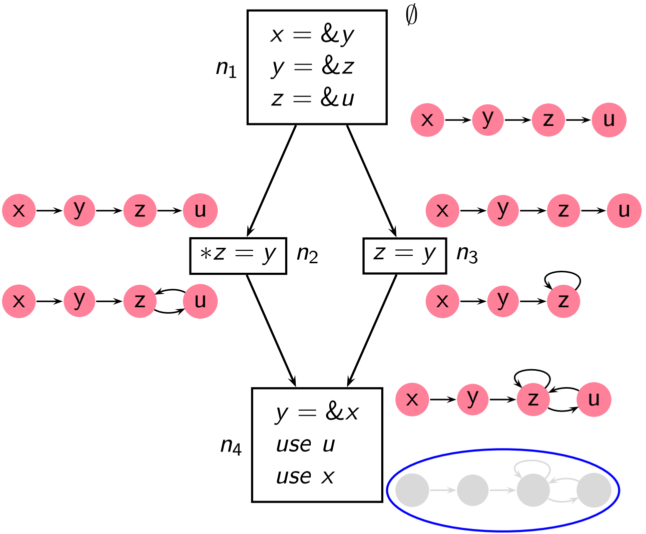
Liveness Based Pointer Analysis: Motivation



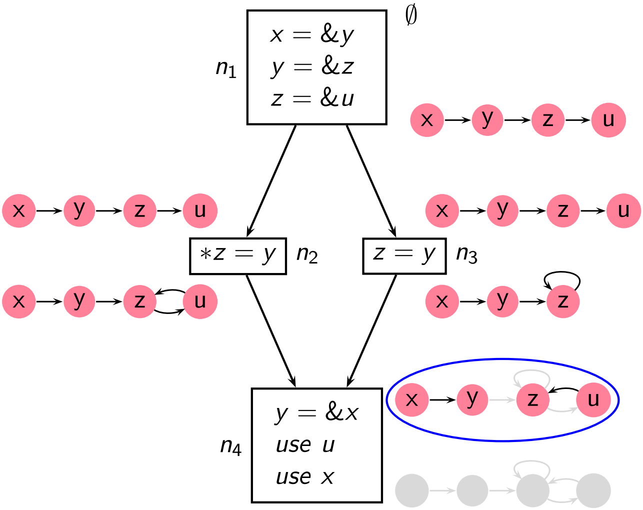
Liveness Based Pointer Analysis: Motivation



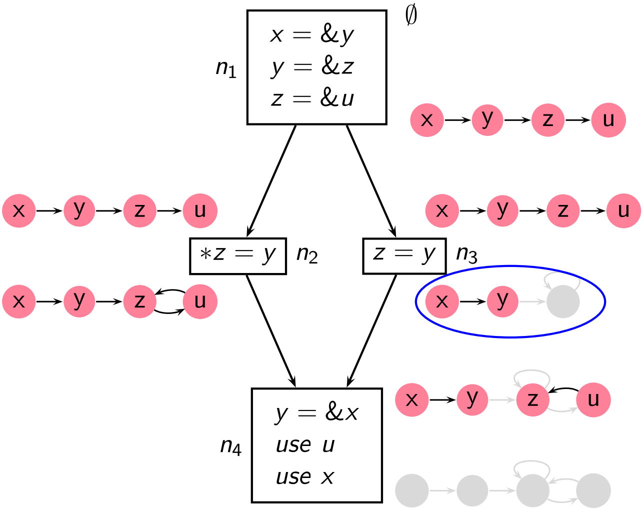
Liveness Based Pointer Analysis: Motivation



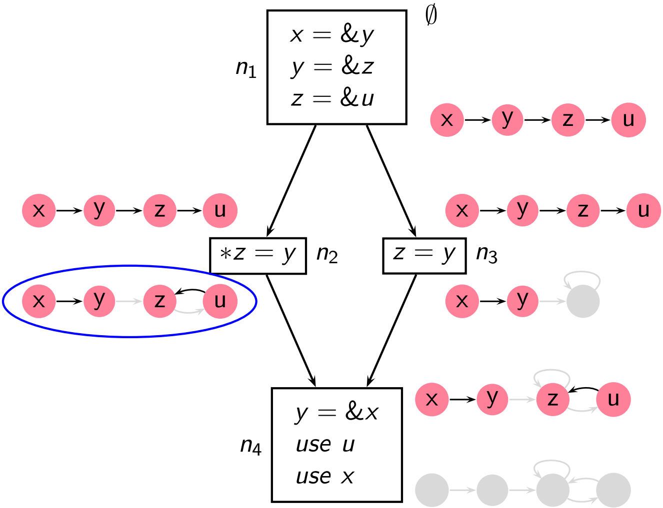
Liveness Based Pointer Analysis: Motivation



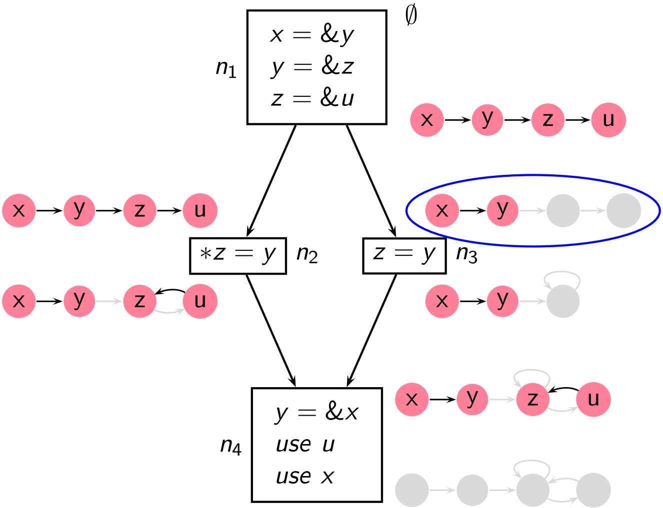
Liveness Based Pointer Analysis: Motivation



Liveness Based Pointer Analysis: Motivation



Liveness Based Pointer Analysis: Motivation



Liveness Based Points-to Analysis (SAS-2012)

- 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 call strings method for full flow and context sensitivity
  - Value based termination of call strings construction (CC-2008)
- Use strong liveness

Liveness Based Interprocedural Points-to Analysis: Empirical Measurements

- Observations on SPEC CPU 2006 benchmarks in GCC 4.6.0
  (Prashant Singh Rawat, IITB 2012)

  Usable pointer information is small and sparse

No of Points-to pairs	Percentable of basic blocks
0	64-96%
1-4	9-25%
5-8	0-10%
8+	0-4%

Liveness Based Interprocedural Points-to Analysis: Empirical Measurements

- Observations on SPEC CPU 2006 benchmarks in GCC 4.6.0 (Prashant Singh Rawat, IITB 2012)

Usable pointer information is small and sparse

No of Points-to pairs	Percentable of basic blocks
0	64-96%
1-4	9-25%
5-8	0-10%
8+	0-4%

Liveness Based Interprocedural Points-to Analysis: Empirical Measurements

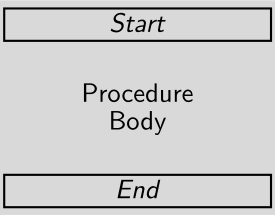
- Observations on SPEC CPU 2006 benchmarks in GCC 4.6.0 (Prashant Singh Rawat, IITB 2012)

  Usable pointer information is small and sparse

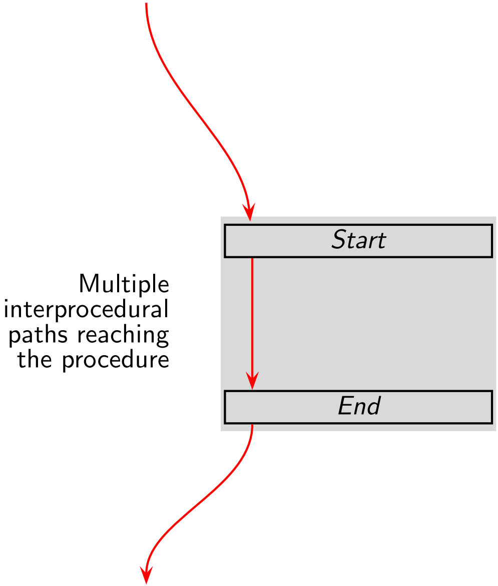
No of Points-to pairs	Percentable of basic blocks
0	64-96%
1-4	9-25%
5-8	0-10%
8+	0-4%

- Independently implemented and verified in
  - LLVM (Dylan McDermott, Cambridge, 2016) and
  - GCC 4.7.2 (Priyanka Sawant, IITB, 2016)

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

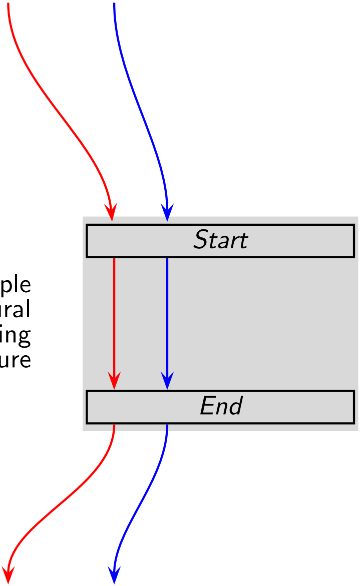


Value Contexts (CC-2008, SAS-2012, SOAP-2013)

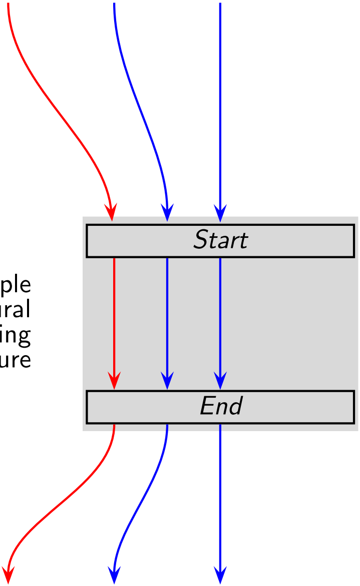


Value Contexts (CC-2008, SAS-2012, SOAP-2013)

Multiple interprocedural paths reaching the procedure



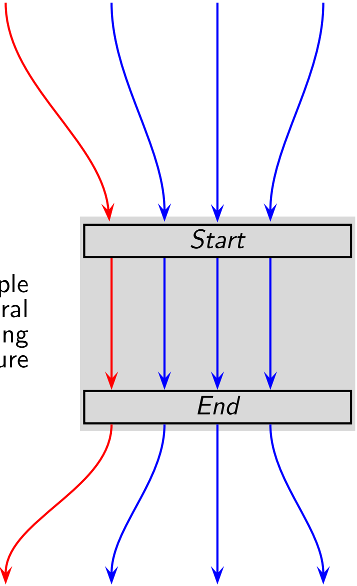
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



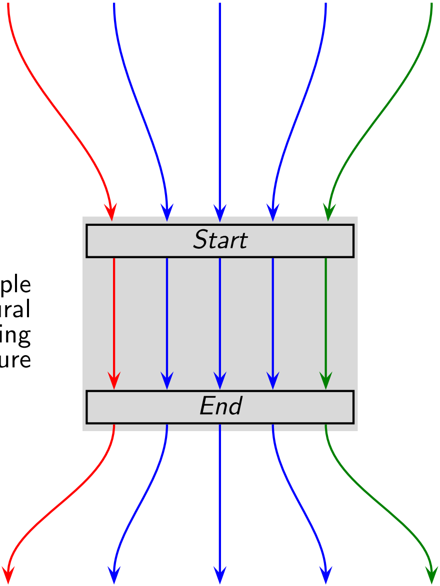
Multiple interprocedural paths reaching the procedure

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

Multiple interprocedural paths reaching the procedure

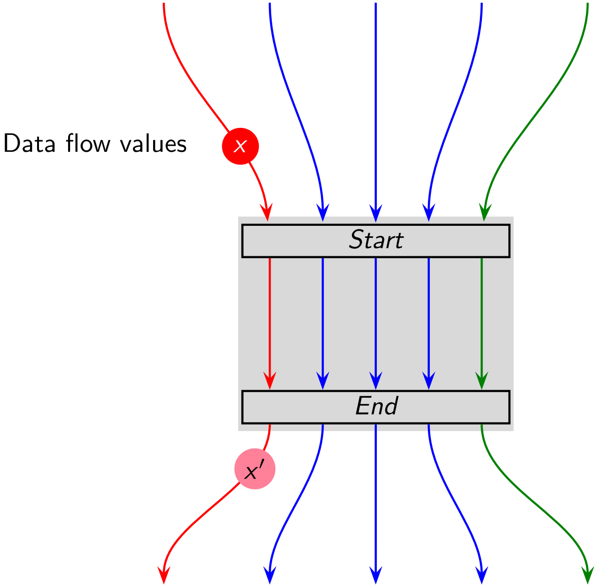


Value Contexts (CC-2008, SAS-2012, SOAP-2013)

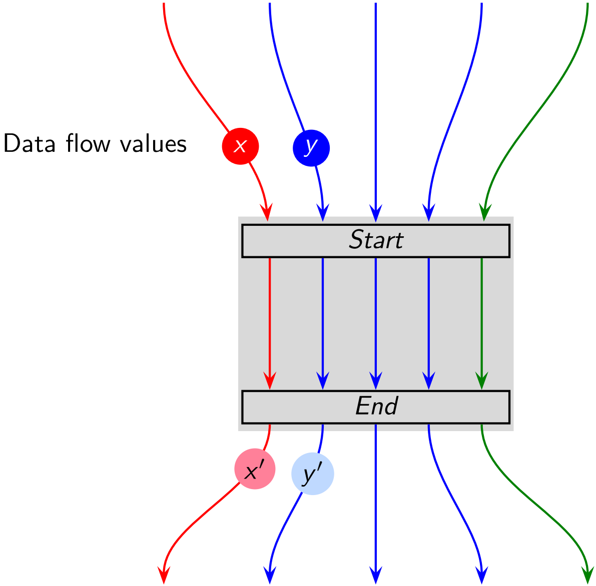


Multiple interprocedural paths reaching the procedure

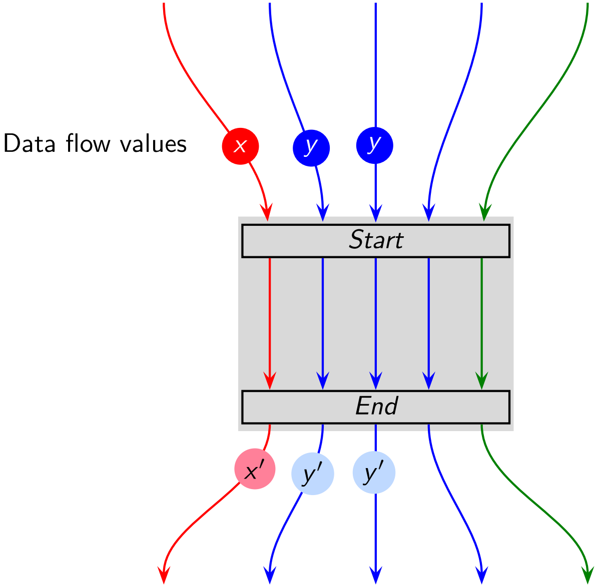
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



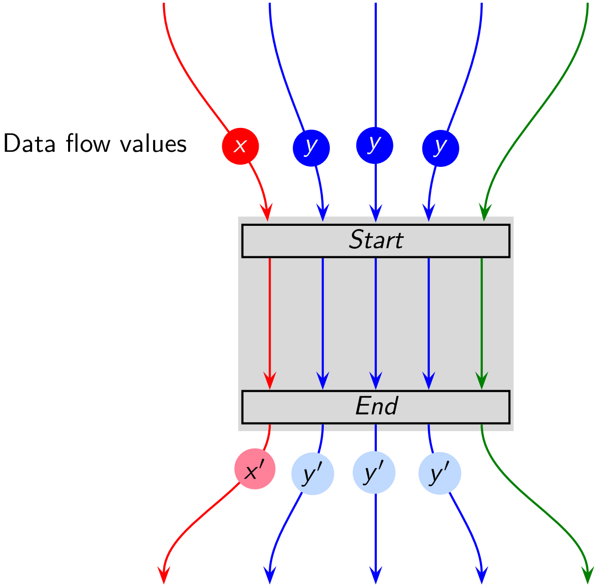
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



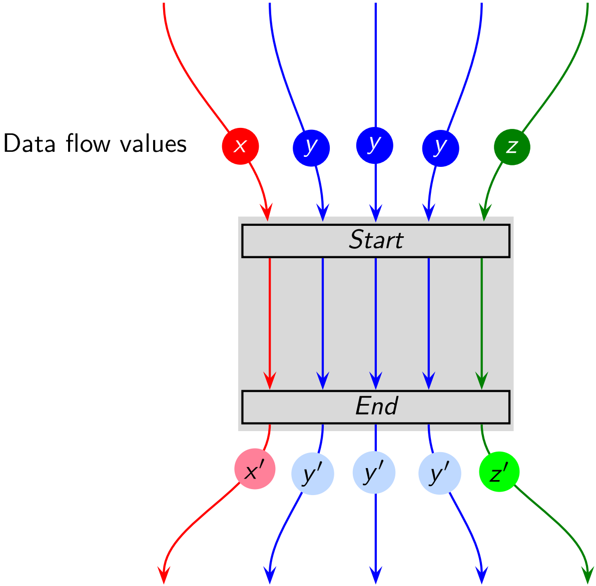
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



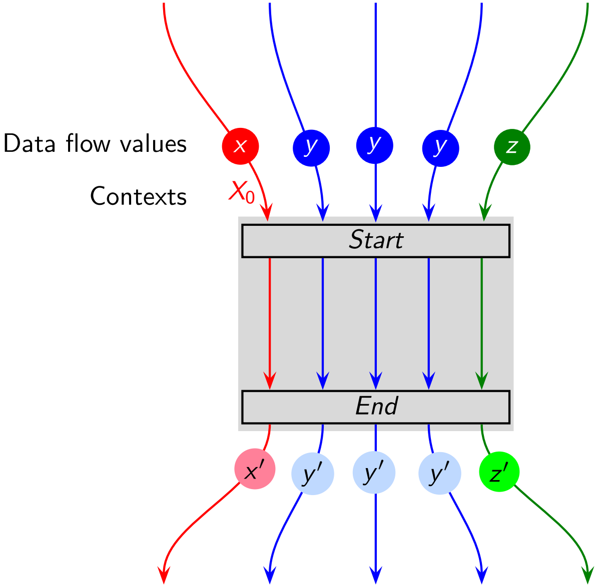
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



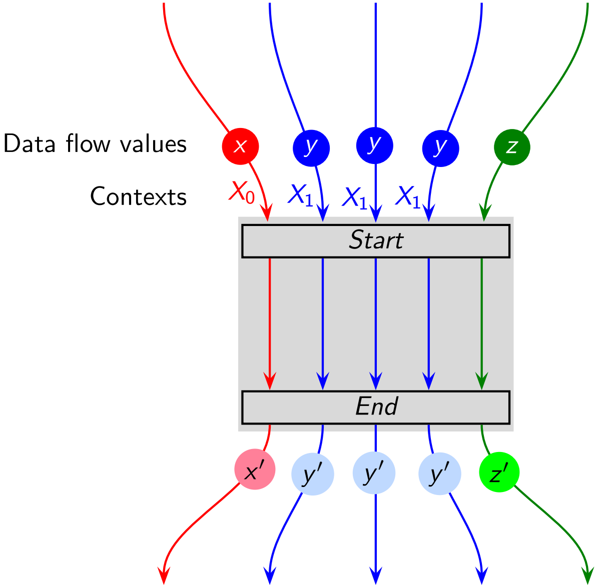
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



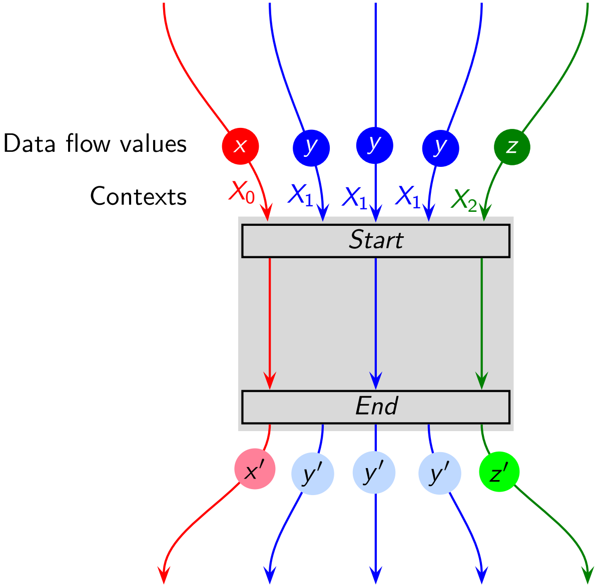
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



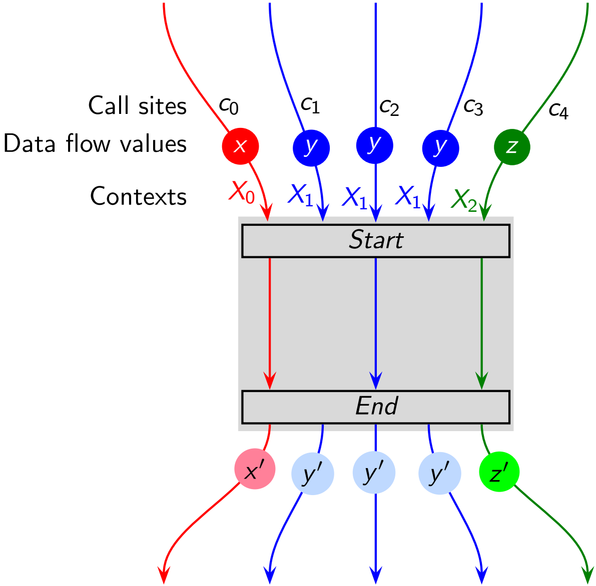
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



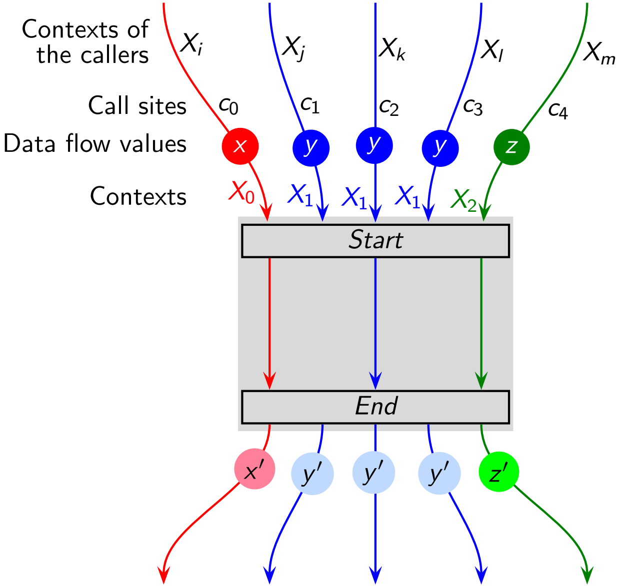
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



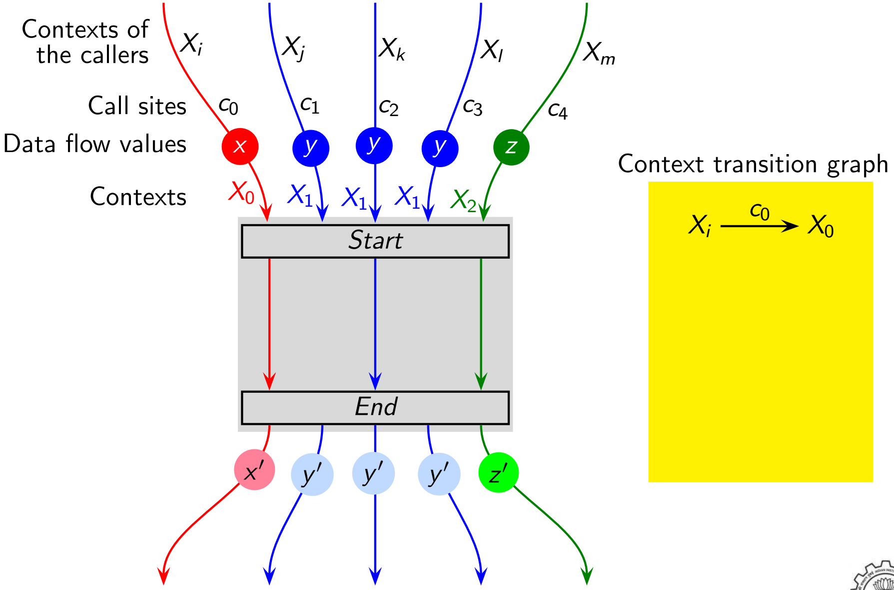
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



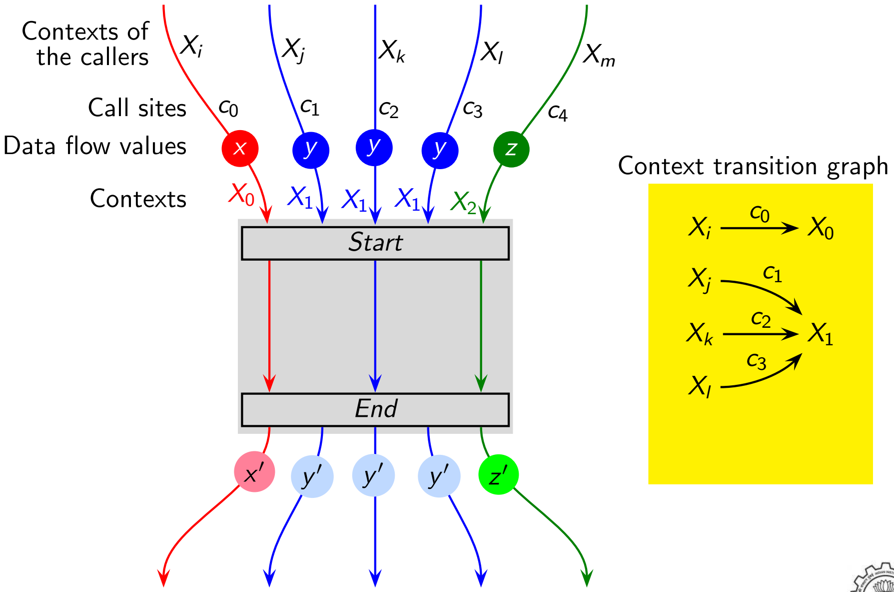
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



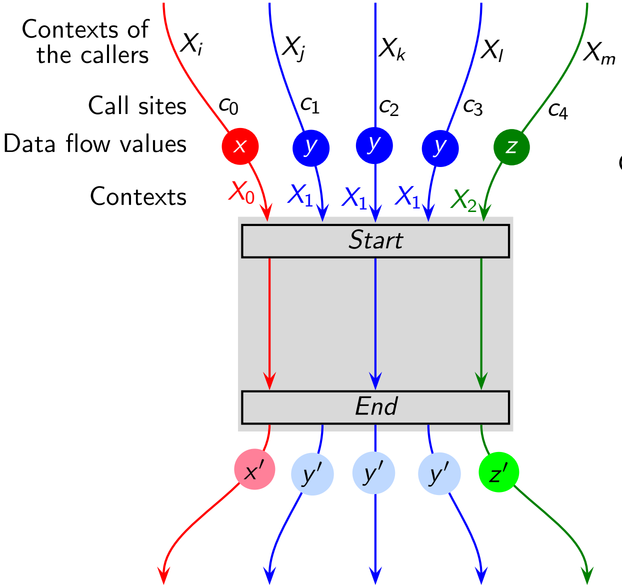
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



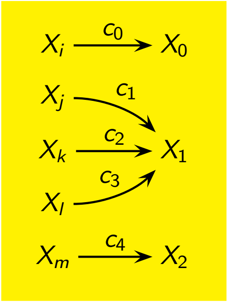
Value Contexts (CC-2008, SAS-2012, SOAP-2013)



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



Context transition graph



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

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

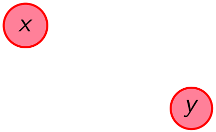
  - Time brought down from 3.973×10^6 ms to 2.37 ms
  - No of call strings brought down from 10^6+ to 8

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 legnth 7, less than 50%
    For length 10, less than 5%

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

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

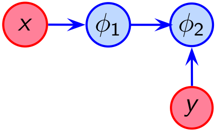


All variables are global

Red nodes are known named locations

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

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



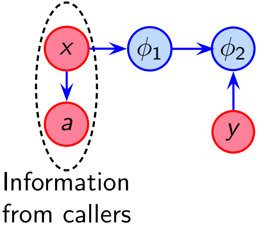
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

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

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



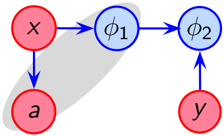
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

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

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



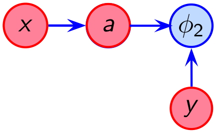
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

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

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



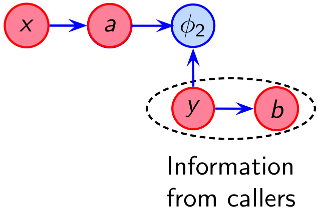
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

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

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



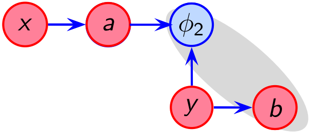
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

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

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



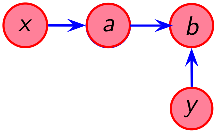
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

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

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



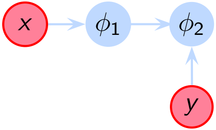
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

Generalized Points-to Facts: A High Level Abstraction of Memory for Points-to Analysis (SAS-2016)

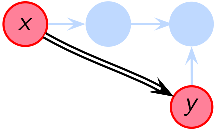
```
f()
{
  *x = y
}
```



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

Generalized Points-to Facts: A High Level Abstraction of Memory for Points-to Analysis (SAS-2016)

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

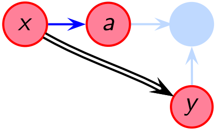


Blue arrows are low level view of memory in terms of classical points-to facts

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

Generalized Points-to Facts: A High Level Abstraction of Memory for Points-to Analysis (SAS-2016)

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

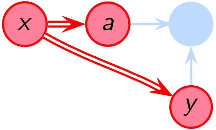


Blue arrows are low level view of memory in terms of classical points-to facts

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

Generalized Points-to Facts: A High Level Abstraction of Memory for Points-to Analysis (SAS-2016)

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

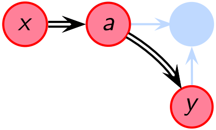


Blue arrows are low level view of memory in terms of classical points-to facts

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

Generalized Points-to Facts: A High Level Abstraction of Memory for Points-to Analysis (SAS-2016)

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

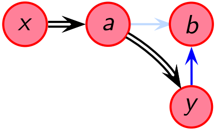


Blue arrows are low level view of memory in terms of classical points-to facts

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

Generalized Points-to Facts: A High Level Abstraction of Memory for Points-to Analysis (SAS-2016)

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

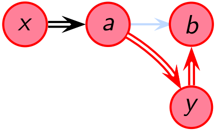


Blue arrows are low level view of memory in terms of classical points-to facts

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

Generalized Points-to Facts: A High Level Abstraction of Memory for Points-to Analysis (SAS-2016)

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

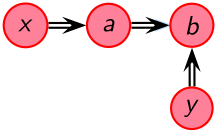


Blue arrows are low level view of memory in terms of classical points-to facts

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

Generalized Points-to Facts: A High Level Abstraction of Memory for Points-to Analysis (SAS-2016)

```
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 Graphs (GPGs) for Points-to Analysis (SAS-2016)

Construction of bottom up summary flow functions using GPGs

- Issues at intraprocedural level

  Flow sensitivity, strong and weak updates, efficiency using SSA form

- Issues at interprocedural level

  Context sensitivity: Composition of callee's GPGs within callers

  Efficiency using bypassing of irrelevant information

- Handling advanced features

  Function Pointers, Heap, Structures, Union, Arrays, Pointer Arithmetic

- Theoretical issues. Soundness and complexity

- Implementation and measurements

  Using LTO framework in GCC 4.7.2 scaling to 158 KLoC

Generalized Points-to Graphs (GPGs) 1

A GPG is a graph with

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

Generalized Points-to Graphs (GPGs) 1

A GPG is a graph with

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

A GPG is analogous to a CFG of a procedure

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

Generalized Points-to Graphs (GPGs) 1

A GPG

- N
- E

First difference:

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

A GPG is analogous to a CFG of a procedure

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

Generalized Points-to Graphs (GPGs) 1

A GPG

- N
- E

Second difference:

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

A GPG is analogous to a CFG of a procedure

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

Generalized Points-to Graphs (GPGs) 2

Construction of Initial GPGs:

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

GPGs then undergo extensive optimizations

GPGs Across Optimizations

CFG of proc f

$x = \&a;$

$\downarrow$

$g();$

$\downarrow$

$x = \&b;$

CFG of proc g

$y = x;$

GPGs Across Optimizations

CFG of proc f

$x = \&a;$

$\downarrow$

$g();$

$\downarrow$

$x = \&b;$

Initial GPG of proc f

$x \xrightarrow[1]{1|0} a$

$\downarrow$

$g();$

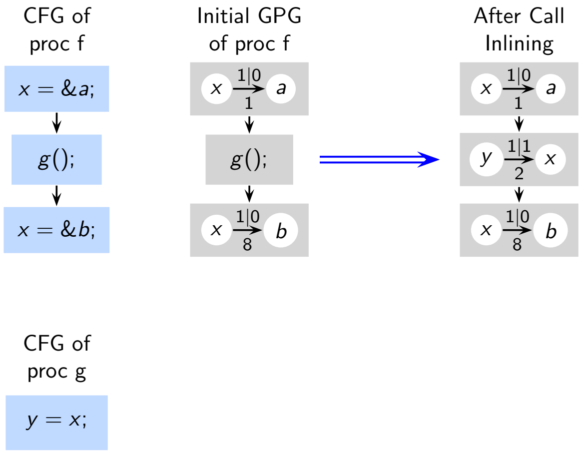
$\downarrow$

$x \xrightarrow[8]{1|0} b$

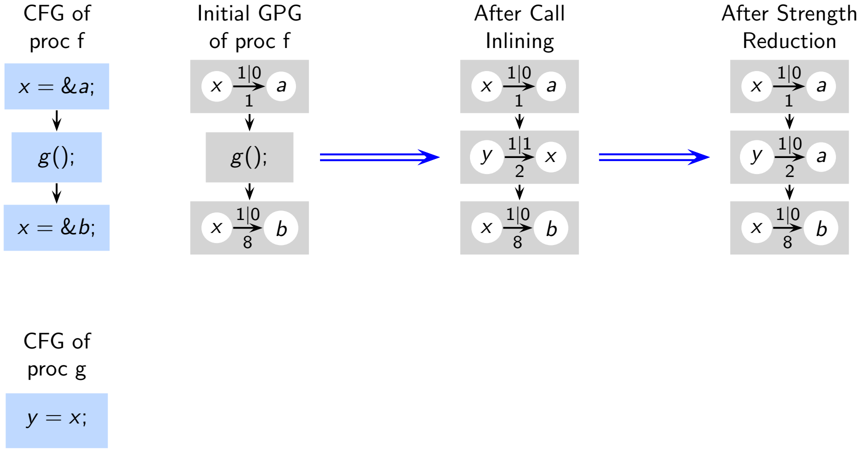
CFG of proc g

$y = x;$

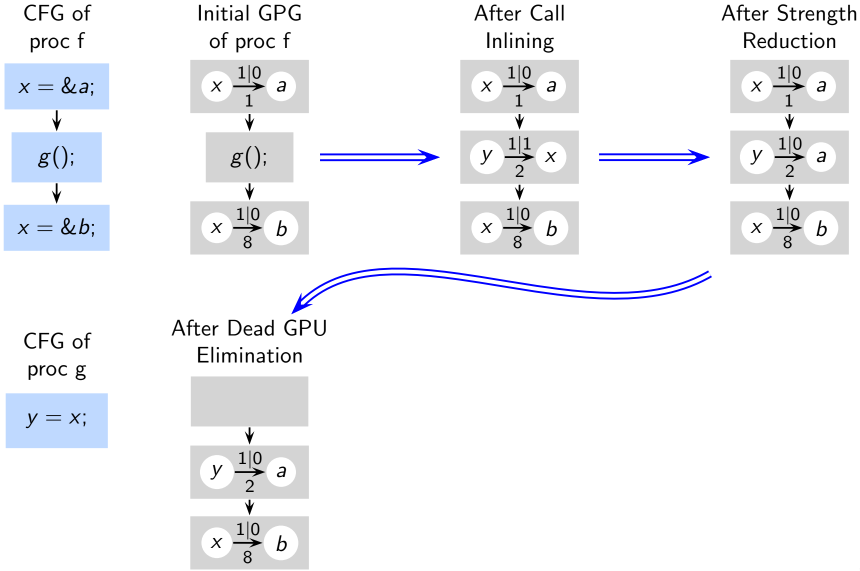
GPGs Across Optimizations



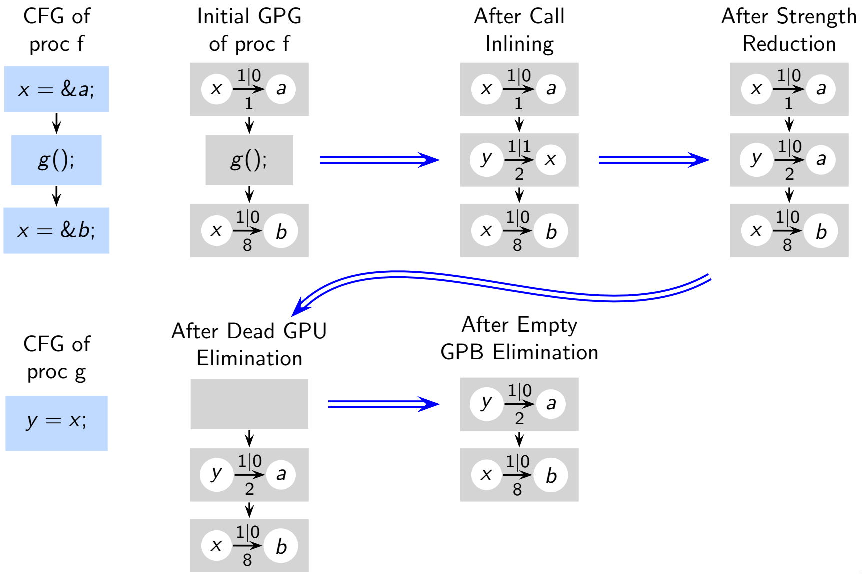
GPGs Across Optimizations



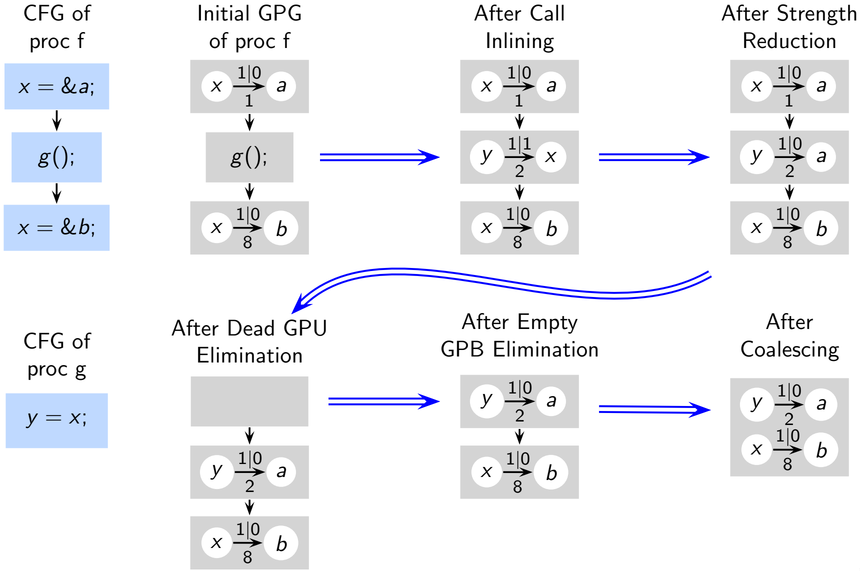
GPGs Across Optimizations



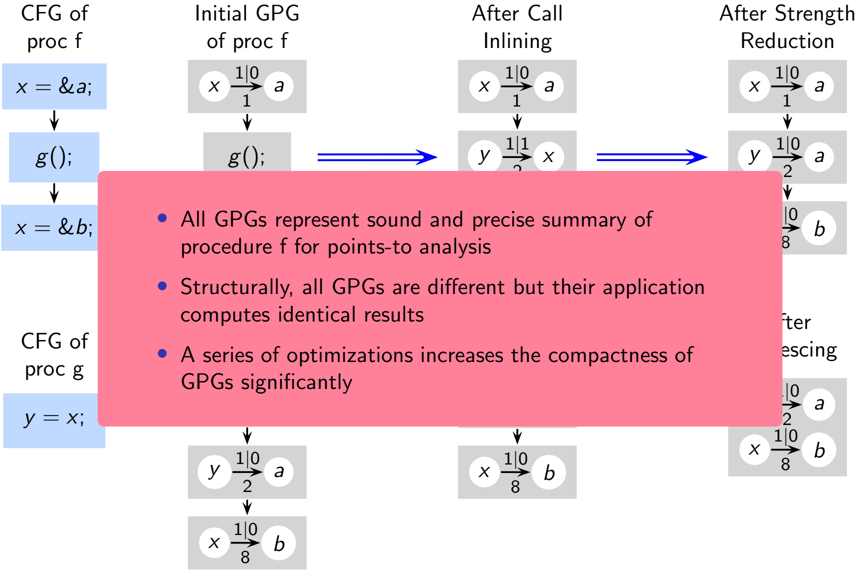
GPGs Across Optimizations



GPGs Across Optimizations



GPGs Across Optimizations



- All GPGs represent sound and precise summary of procedure f for points-to analysis
- Structurally, all GPGs are different but their application computes identical results
- A series of optimizations increases the compactness of GPGs significantly

Factors affecting Scalability

Three issues that cause non-scalability

- Modelling indirect accesses of pointees that are defined in callers without examining their code
  - GPUs track indirection levels that relate (transitively indirect) pointees of a variable with those of other variables
- Preserving data dependence between memory updates
  - Maintain minimal control flow between memory updates ensuring soundness and precision
- Incorporating the effect of summaries of the callee procedures transitively
  - Series of GPG optimizations gives compactness that mitigate the impact of transitive inlining

Part 3

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

What should be computed?

Maximum Computation

Minimum Computation

When should it be computed?

Early Computation

Late Computation

A Spectrum of Possible Ways of Performing Computation

exhaustive computation

computation restricted to usable information

avoiding redundant computation

demand driven computation

What should be computed?

Maximum Computation ⟷ Minimum Computation

When should it be computed?

Early Computation ⟷ Late Computation

Do not compute what you don't need!

Who defines what is needed?

A Spectrum of Possible Ways of Performing Computation

exhaustive computation

computation restricted to usable information

avoiding redundant computation

demand driven computation

What should be computed?

Maximum Computation — Minimum Computation

When should it be computed?

Early Computation — Late Computation

Do not compute what you don't need!

Who defines what is needed? Client

A Spectrum of Possible Ways of Performing Computation

incremental computation

splitting into pre and post computation

exhaustive computation

computation restricted to usable information

avoiding redundant computation

demand driven computation

Maximum Computation ← What should be computed? → Minimum Computation

Early Computation ← When should it be computed? → Late Computation

Do not compute what you don't need!

Who defines what is needed? Algorithm, Data Structure

A Spectrum of Possible Ways of Performing Computation

incremental computation

splitting into pre and post computation

exhaustive computation

computation restricted to usable information

avoiding redundant computation

demand driven computation

What should

Maximum Computation

When should

Early Computation

Do not compute what you don't need

Who defines what is needed?

Algorithm, Data Structure

Other examples:

- Bottom up summary flow functions
- Value contexts
- Work list based methods
- BDDs

A Spectrum of Possible Ways of Performing Computation

incremental computation

splitting into pre and post computation

exhaustive computation

computation restricted to usable information

avoiding redundant computation

demand driven computation

Maximum Computation ← What should be computed? → Minimum Computation

Early Computation ← When should it be computed? → Late Computation

Do not compute what you don't need!

Who defines what is needed? Definition of Analysis

A Spectrum of Possible Ways of Performing Computation

incremental computation

splitting into pre and post computation

exhaustive computation

computation restricted to usable information

avoiding redundant computation

demand driven computation

Maximum Computation ← What should be computed? → Minimum Computation

Early Computation ← When should it be computed? → Late Computation

Do not compute what you don't need!

Who defines what is needed? No One!

A Spectrum of Possible Ways of Performing Computation

incremental computation

splitting into pre and post computation

exhaustive computation

computation restricted to usable information

avoiding redundant computation

demand driven computation

What should be computed?

Maximum Computation

Minimum Computation

When should it be computed?

Early Computation

Late Computation

Do not compute what you don't need!

Who defines what is needed?

These seem orthogonal and may be used together

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?

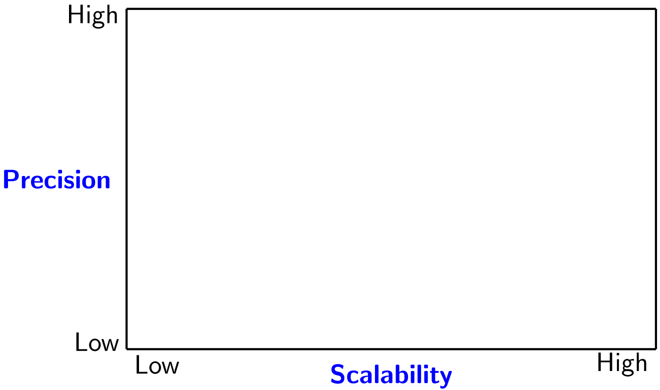
Under-approximated Control Flow	Original Control Flow	Over-approximated Control Flow
Missing control flow paths		Flow insensitivity Context insensitivity

The Next Holy Grail in Search of Scalability?

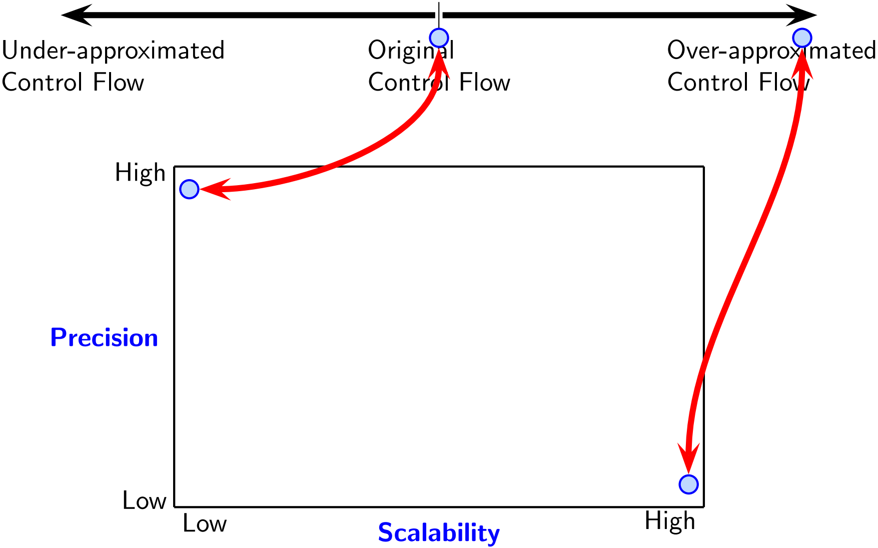
Under-approximated Control Flow	Original Control Flow	Over-approximated Control Flow
Missing control flow paths	Sound and Precise	Flow insensitivity Context insensitivity
Unsound		Sound and Imprecise

The Next Holy Grail in Search of Scalability?

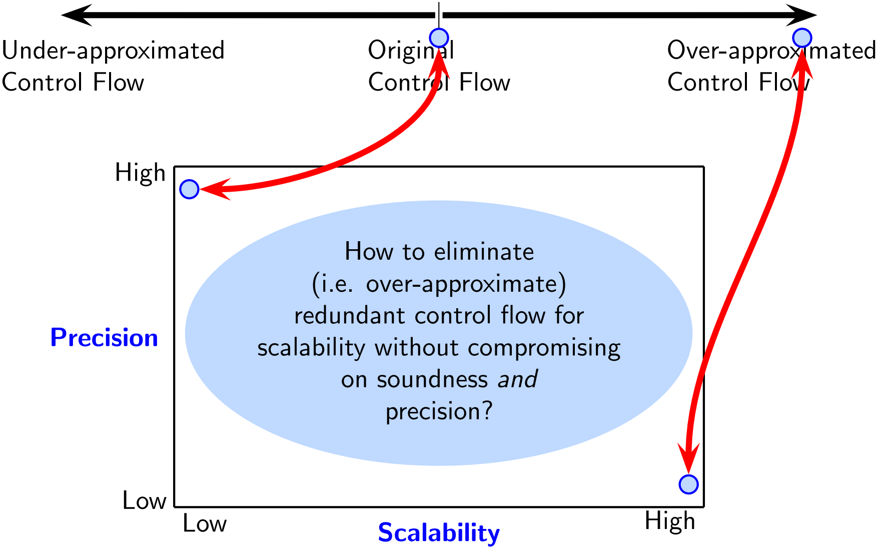
Under-approximated Control Flow ⟵ Original Control Flow ⟶ Over-approximated Control Flow



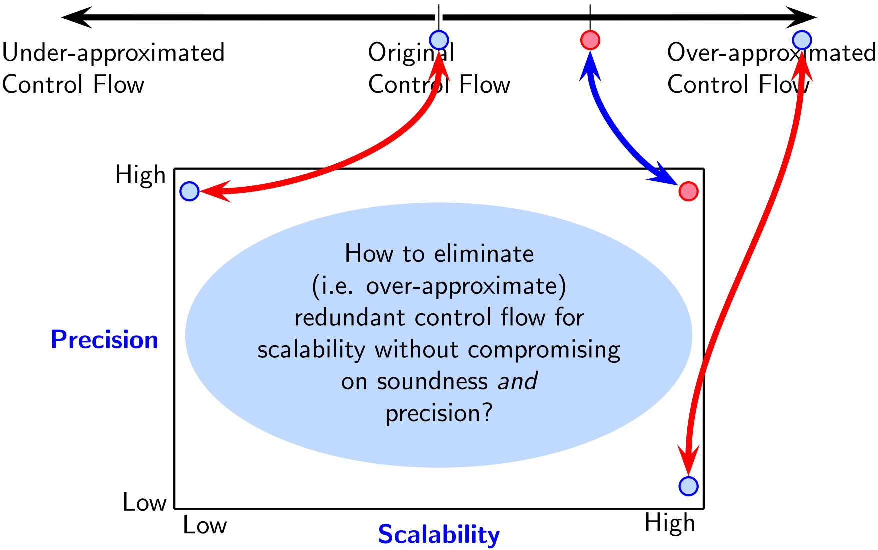
The Next Holy Grail in Search of Scalability?



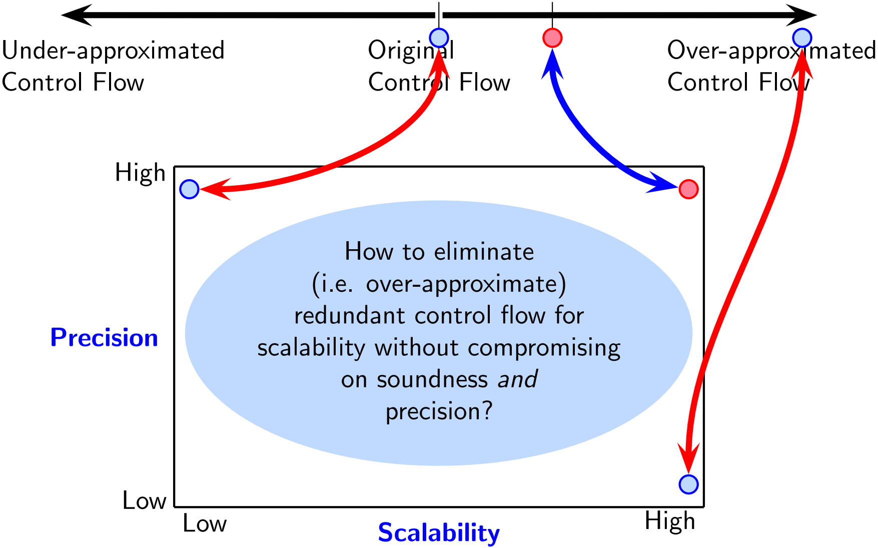
The Next Holy Grail in Search of Scalability?



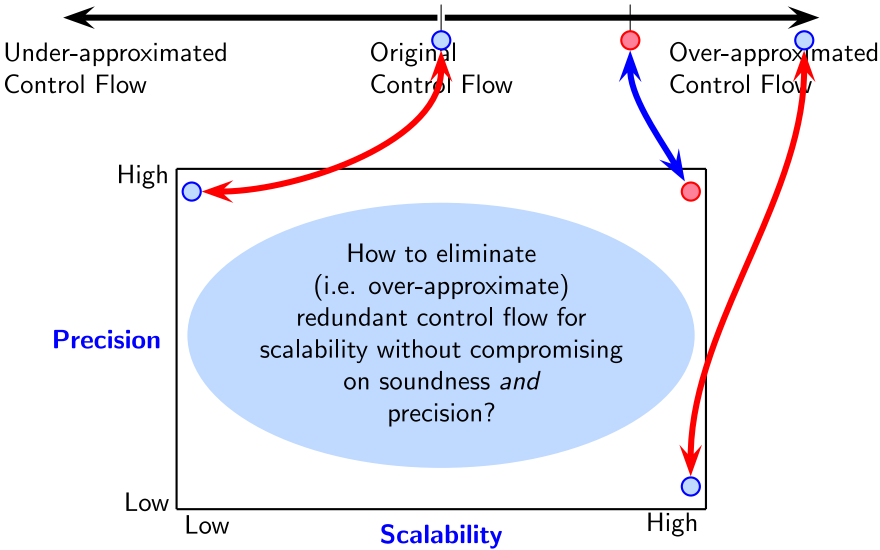
The Next Holy Grail in Search of Scalability?



The Next Holy Grail in Search of Scalability?



The Next Holy Grail in Search of Scalability?



Conclusions

- Building quick approximations and compromising on precision may not be necessary for efficiency
- Building clean abstractions to separate the necessary information from redundant information is much more significant

Conclusions

- Building quick approximations and compromising on precision may not be necessary for efficiency
- Building clean abstractions to separate the necessary information from redundant information is much more significant

  Our experience of points-to analysis shows that

  - Use of liveness reduced the pointer information ...
  - which reduced the number of contexts required ...
  - which reduced the liveness and pointer information ...

Conclusions

- Building quick approximations and compromising on precision may not be necessary for efficiency
- Building clean abstractions to separate the necessary information from redundant information is much more significant

  Our experience of points-to analysis shows that

  - Use of liveness reduced the pointer information ...
  - which reduced the number of contexts required ...
  - which reduced the liveness and pointer information ...

  This encouraged us to explore bottom summary flow functions for points-to analysis

  - which reduced the number of times a procedure is processed and ...
  - gave rise to generalized points-to facts...
  - which reduced the size of intermediate points-to graphs...

Conclusions

- Building quick approximations and compromising on precision may not be necessary for efficiency
- Building clean abstractions to separate the necessary information from redunda

  Our ex

  - U
  - w
  - w

  This en for points-to analysis

  - which reduced the number of times a procedure is processed and ...
  - gave rise to generalized points-to facts...
  - which reduced the size of intermediate points-to graphs...

Approximations should come *after* building abstractions and *not before*

Acknowledgements

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

. . . and many more

Last But Not the Least

Thank You!