Interprocedural Heap Analysis using Access Graphs and Value Contexts

with applications to liveness-based garbage collection

Rohan Padhye

under the guidance of

Prof. Uday Khedker

Department of Computer Science & Engineering
Indian Institute of Technology Bombay

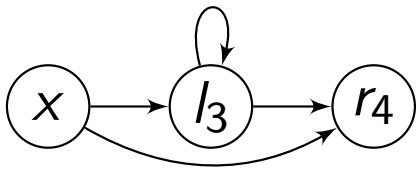
M.Tech Project

Outline

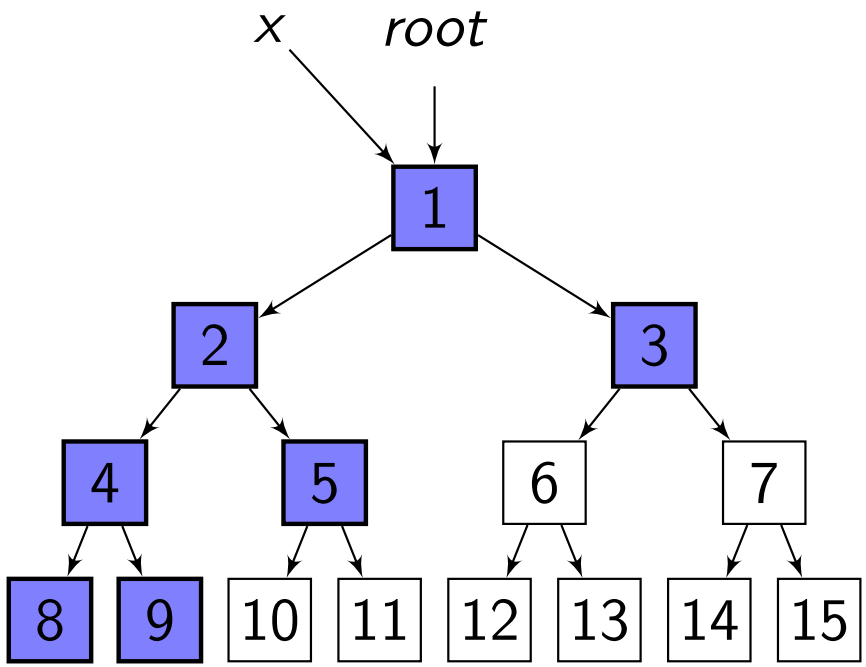
Background

Heap Reference Analysis [Khedker, Sanyal & Karkare, 2007]

```
S1: x = root
S2: while (x.val > M):
S3:     x = x.l
S4: x = x.r
S5: print x.val
S6: EXIT
```



Access graph for x at S_2.



The binary tree in the heap at S_2.
Filled nodes are live objects.

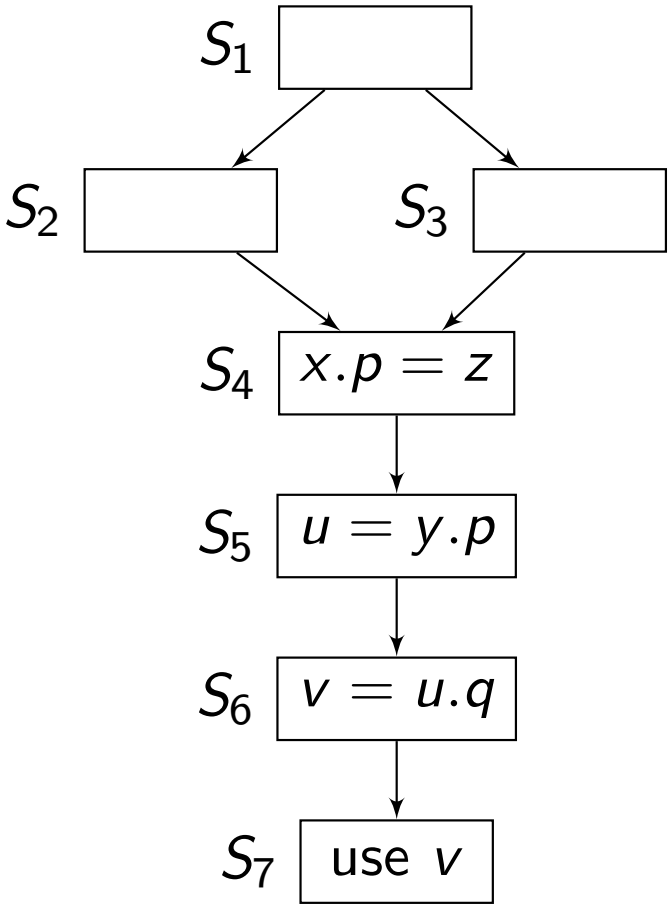
Key Issues

Three main issues in performing Heap Reference Analysis:

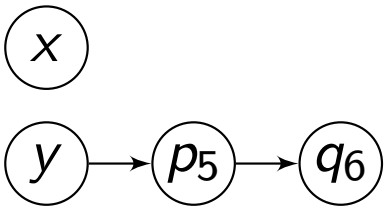
1. How to perform a precise **alias analysis** for arbitrary access paths in the heap?
2. How to implement whole-program heap reference analysis in an **inter-procedural** manner?
3. How to use the resulting access graphs to improve **garbage collection**?

Outline

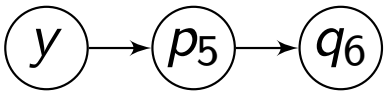
Need for Alias Analysis



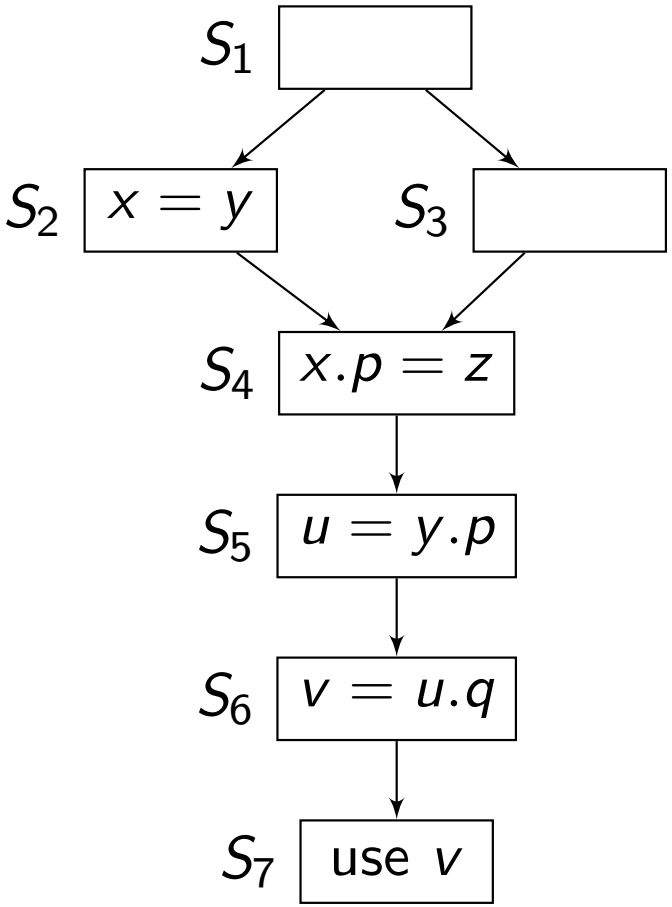
- x and y do not alias at S_4.
- $LVIN_4$



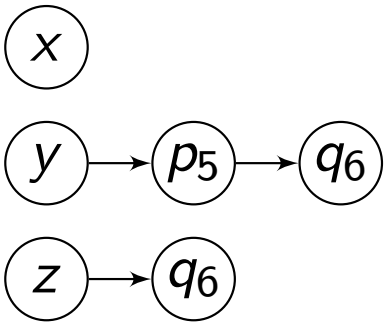
- $LVOUT_4$



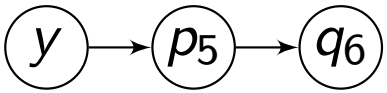
Need for Alias Analysis



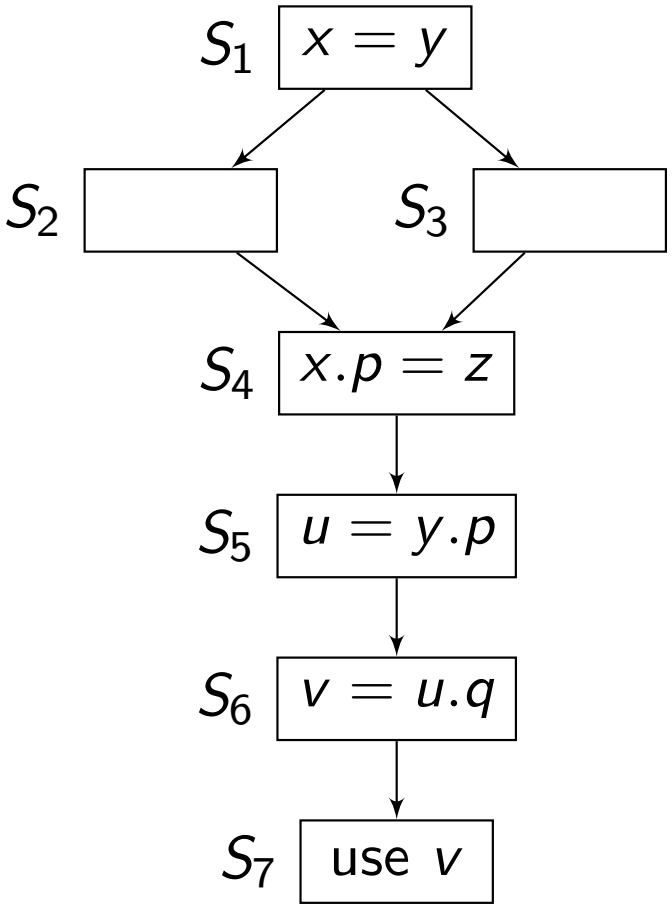
- x *may* alias y at S_4
- $LVIN_4$



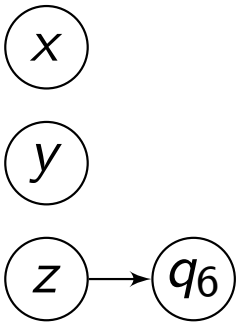
- $LVOUT_4$



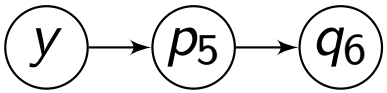
Need for Alias Analysis



- x *must* alias y at S_4
- $LVIN_4$

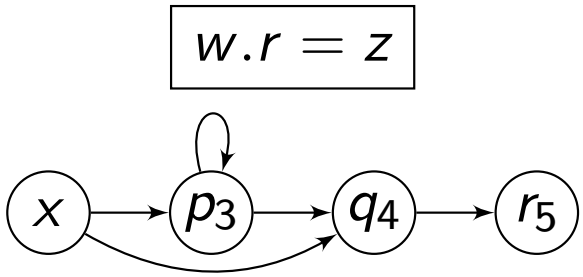


- $LVOUT_4$



Need for Alias Analysis

- May-alias analysis is required for sound heap liveness analysis.
- Must-alias analysis is desirable for performing strong updates.
- In general, alias queries may not be as straightforward as the preceeding examples:



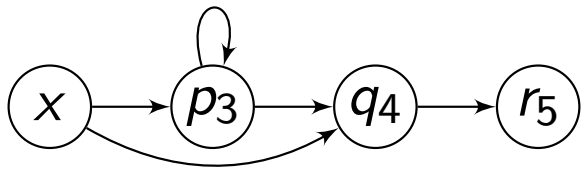
  In the above program, z is live if w may be aliased to any object accessible by the pattern $x(.p)*.q$.
- The key obervation is here is that we need to determine aliases between **live access patterns**.

Access Graphs and Access Patterns

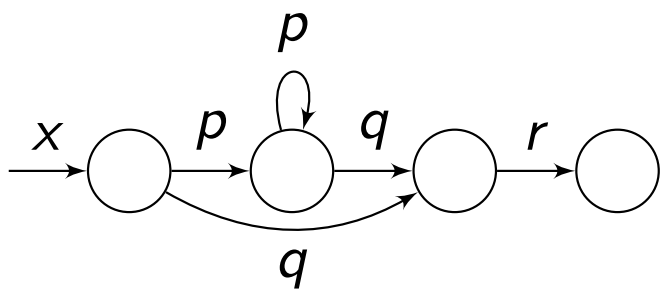
```
S1: w.r = z
S2: while (...):
S3:     x = x.p
S4: x = x.q
S5: x = x.r
S6: use x
S7: EXIT
```

Consider liveness at S_2.

- Access Graph



- Equivalent Automaton



- Access Patterns
  - $x/p_3 : x.p(.p)*$
  - $x/q_4 : x(.p)*.q$
  - $x/r_5 : x(.p)*.q.r$

Approaches to Heap Alias Analysis

Modelling an unbounded number of objects using a finite abstraction:

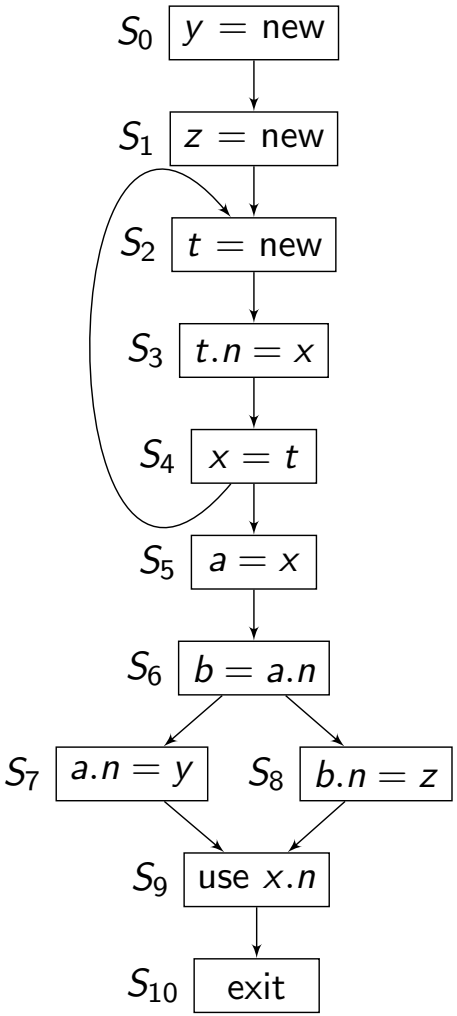
- Muchnick & Jones, 1981: k-limited graph
- Chase, Wegman & Zadeck, 1990: Merge on allocation sites
- Sagiv, Reps & Wilhelm, 1996: "Materialization"
- Sagiv, Reps & Wilhelm, 1999: 3-valued logic

Approaches to Heap Alias Analysis

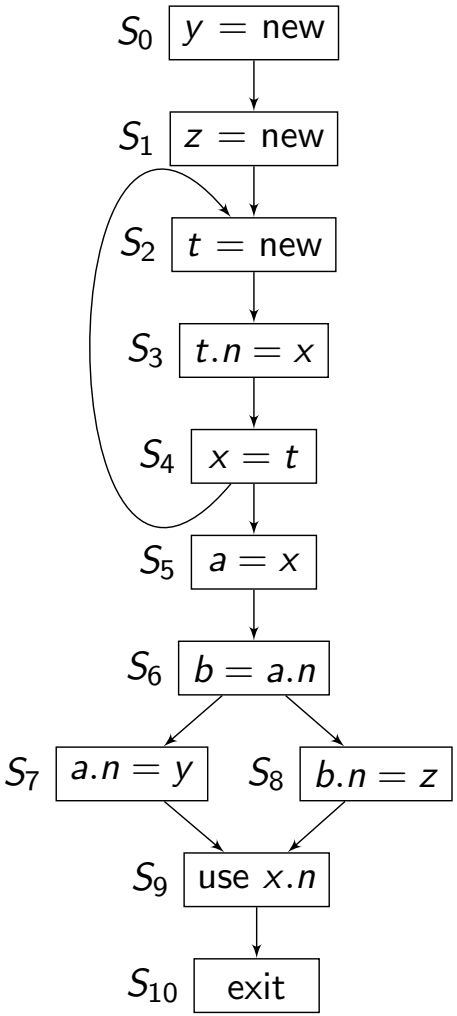
Modelling an unbounded number of objects using a finite abstraction:

- Muchnick & Jones, 1981: k-limited graph
- Chase, Wegman & Zadeck, 1990: Merge on allocation sites
- Sagiv, Reps & Wilhelm, 1996: "Materialization"
- Sagiv, Reps & Wilhelm, 1999: 3-valued logic
- Our approach: Use **access patterns** from liveness graphs to *improve expressibility* of points-to graph

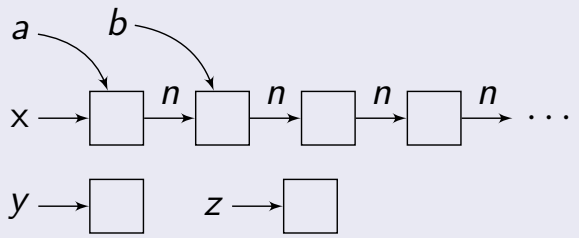
Proposed Approach



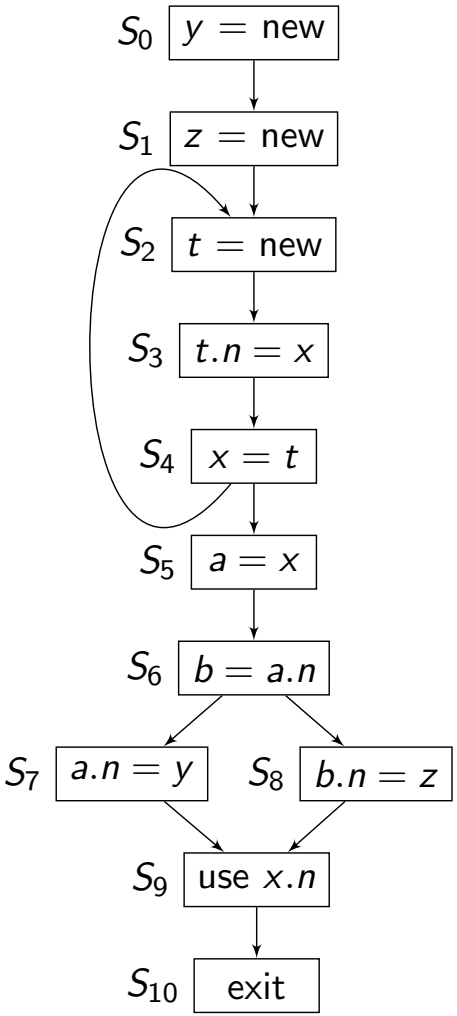
Proposed Approach



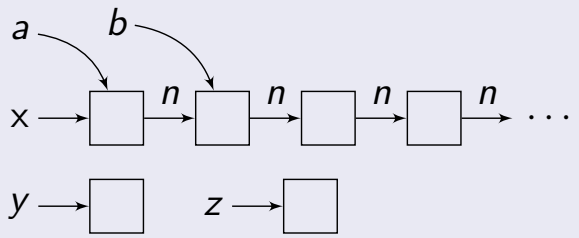
Actual heap layout after S_6



Proposed Approach

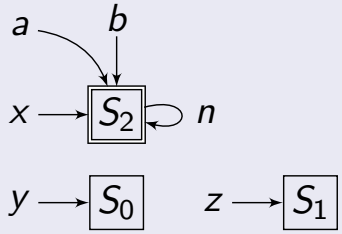


Actual heap layout after S_6

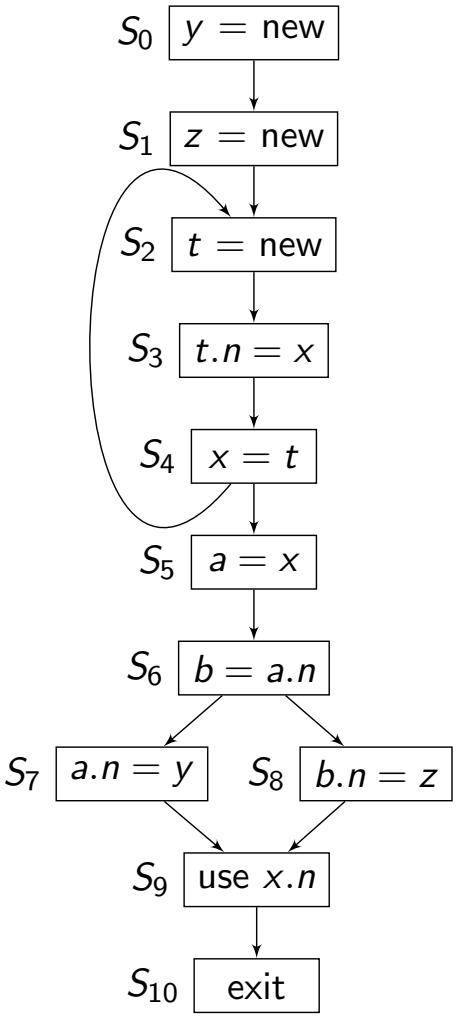


P_0: Initial Points-to Analysis

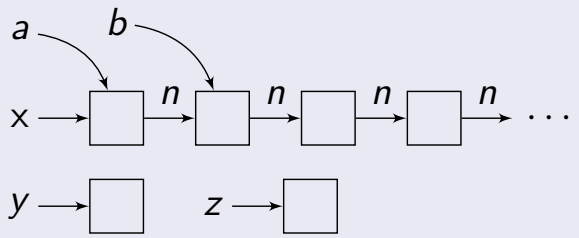
$PTOUT_6 = PTIN_7 = PTIN_8$:



Proposed Approach

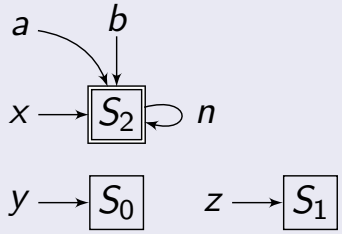


Actual heap layout after S_6



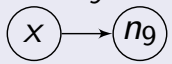
P_0: Initial Points-to Analysis

$PTOUT_6 = PTIN_7 = PTIN_8$:

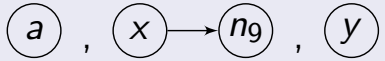


L_1: Liveness after P_0

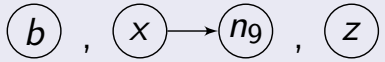
$LVIN_9 = LVOUT_7 = LVOUT_8$:



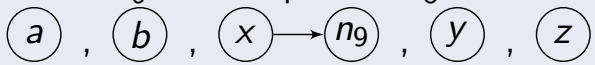
$LVIN_7$ (considering $a \overset{may}{=} x$):



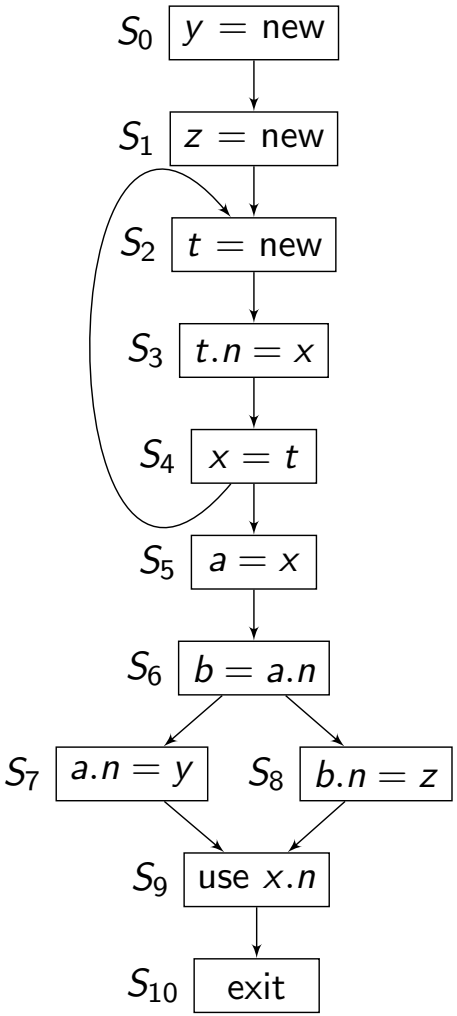
$LVIN_8$ (considering $b \overset{may}{=} x$):



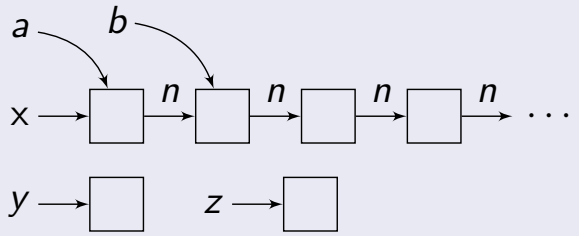
$LVOUT_6 = LVIN_7 \cup LVIN_8$:



Proposed Approach

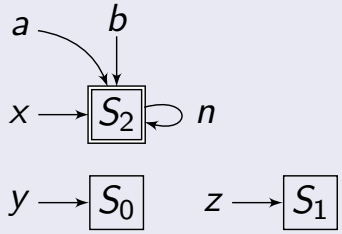


Actual heap layout after S_6



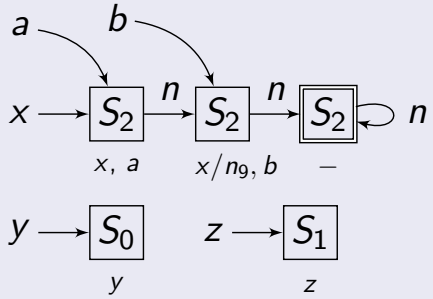
P_0: Initial Points-to Analysis

$PTOUT_6 = PTIN_7 = PTIN_8$:



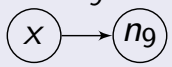
P_1: Points-to Analysis using L_1

$PTOUT_6 = PTIN_7 = PTIN_8$:

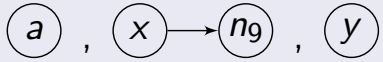


L_1: Liveness after P_0

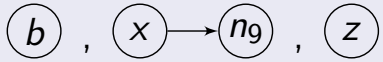
$LVIN_9 = LVOUT_7 = LVOUT_8$:



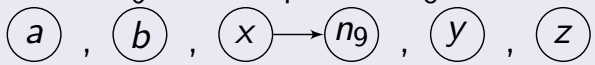
$LVIN_7$ (considering $a \overset{may}{=} x$):



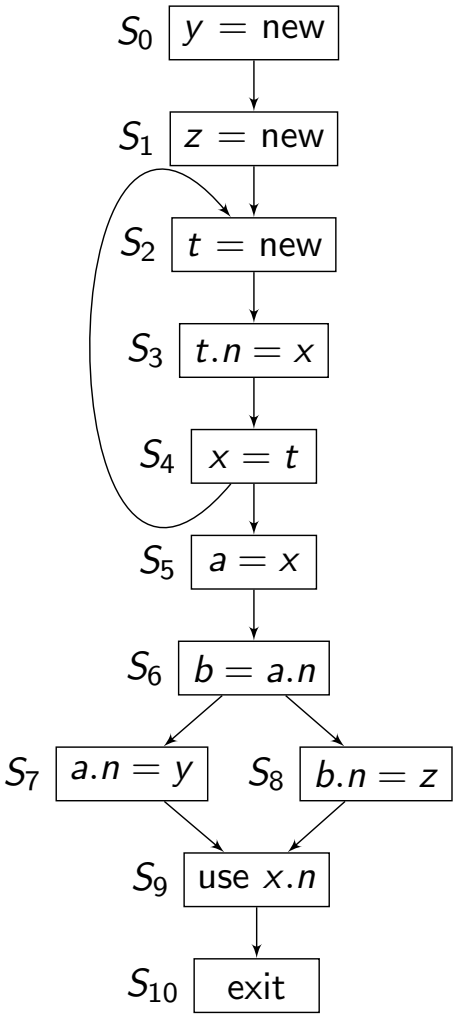
$LVIN_8$ (considering $b \overset{may}{=} x$):



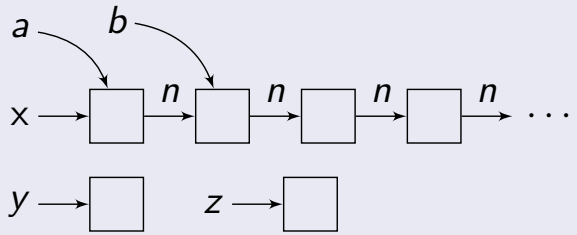
$LVOUT_6 = LVIN_7 \cup LVIN_8$:



Proposed Approach

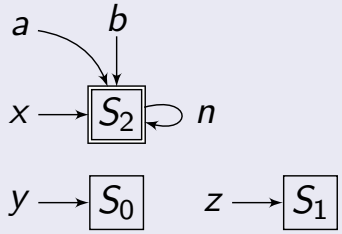


Actual heap layout after S_6



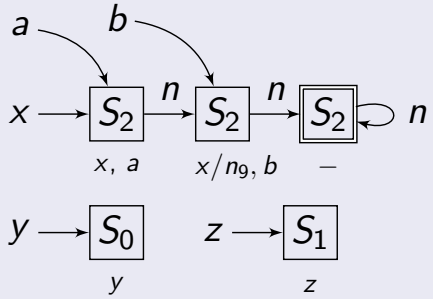
P_0: Initial Points-to Analysis

$PTOUT_6 = PTIN_7 = PTIN_8$:



P_1: Points-to Analysis using L_1

$PTOUT_6 = PTIN_7 = PTIN_8$:



L_1: Liveness after P_0

$LVIN_9 = LVOUT_7 = LVOUT_8$:

$x \rightarrow n_9$

$LVIN_7$ (considering $a \overset{may}{=} x$):

a , $x \rightarrow n_9$, y

$LVIN_8$ (considering $b \overset{may}{=} x$):

b , $x \rightarrow n_9$, z

$LVOUT_6 = LVIN_7 \cup LVIN_8$:

a , b , $x \rightarrow n_9$, y , z

L_2: Liveness after P_1

$LVIN_9 = LVOUT_7 = LVOUT_8$:

$x \rightarrow n_9$

$LVIN_7$ (considering $a \overset{must}{=} x$):

x , y

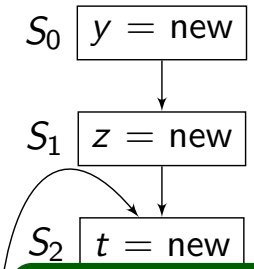
$LVIN_8$ (considering $b \neq x$):

$x \rightarrow n_9$

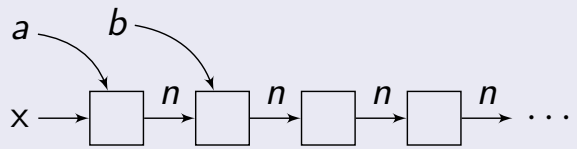
$LVOUT_6 = LVIN_7 \cup LVIN_8$:

$x \rightarrow n_9$, y

Proposed Approach



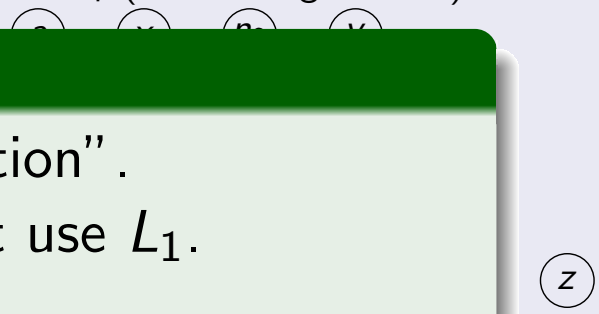
Actual heap layout after S_6



L_1: Liveness after P_0

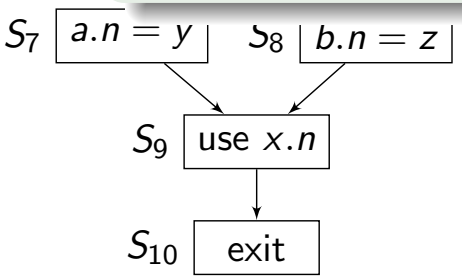
$LVIN_9 = LVOUT_7 = LVOUT_8$:

$LVIN_7$ (considering $a \overset{may}{=} x$):



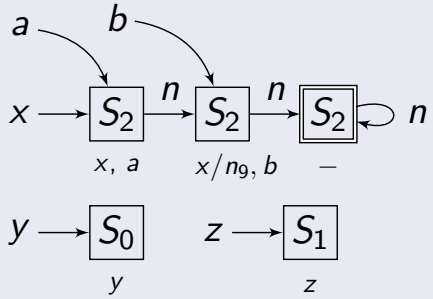
Note

- We are not performing "materialization".
- P_1 does not use P_0 and L_2 does not use L_1.
- These are new passes from scratch!
- L_i uses P_{i-1} for implicit updates.
- P_i uses L_i for expressibility.

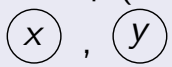


P_1: Points-to Analysis using L_1

$PTOUT_6 = PTIN_7 = PTIN_8$:

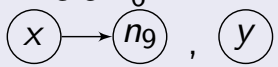


$LVIN_7$ (considering $a \overset{must}{=} x$):



$LVIN_8$ (considering $b \neq x$):

$LVOUT_6 = LVIN_7 \cup LVIN_8$:



Proposed Approach

- Key idea: distinguish between objects accessible by distinct sets of access patterns.
- Thus, our approach is more precise than naive summarization in that:
  1. Unnecessary may-aliases are avoided.
  2. Useful must-aliases are discovered.
- Inter-dependence of liveness and points-to analysis:
  1. Perform naive points-to (summarize on alloc sites).
  2. Backward analysis to get huge liveness info (sound but imprecise).
  3. Again do points-to, distinguishing on access patterns found above.
  4. Another round of backward analysis to get precise liveness info.
  5. Fixed point...?

Accessor Relationship Graph

Symbol	Definition	Cardinality
V	Variables	Proportional to program size
M	Memory allocation sites	Proportional to program size
R	Field dereference points	Proportional to program size
A	Access graph nodes	$\|V\| + \|V\| \times \|R\|$
H	Heap graph nodes	$\|M\| \times 2^{\|A\|}$

Accessor Relationship Graph

Symbol	Definition	Cardinality
V	Variables	Proportional to program size
M	Memory allocation sites	Proportional to program size
R	Field dereference points	Proportional to program size
A	Access graph nodes	$\|V\| + \|V\| \times \|R\|$
H	Heap graph nodes	$\|M\| \times 2^{\|A\|}$

Definition

Accessor Relationship Graph is a 3-tuple $\langle E_v, E_f, summary \rangle$, where:

- $E_v \subseteq V \times H$
- $E_f \subseteq H \times F \times H$
- $summary : H \rightarrow \{true, false\}$

Lattice Representation

Definition

$\langle E_v, E_f, summary \rangle \sqsupseteq \langle E'_v, E'_f, summary' \rangle$ if:

- $E_v \subseteq E'_v$
- $E_f \subseteq E'_f$
- $\forall k \in H : summary(k) \Rightarrow summary'(k)$

Lattice Representation

Definition

$\langle E_v, E_f, summary \rangle \sqsupseteq \langle E'_v, E'_f, summary' \rangle$ if:

- $E_v \subseteq E'_v$
- $E_f \subseteq E'_f$
- $\forall k \in H : summary(k) \Rightarrow summary'(k)$

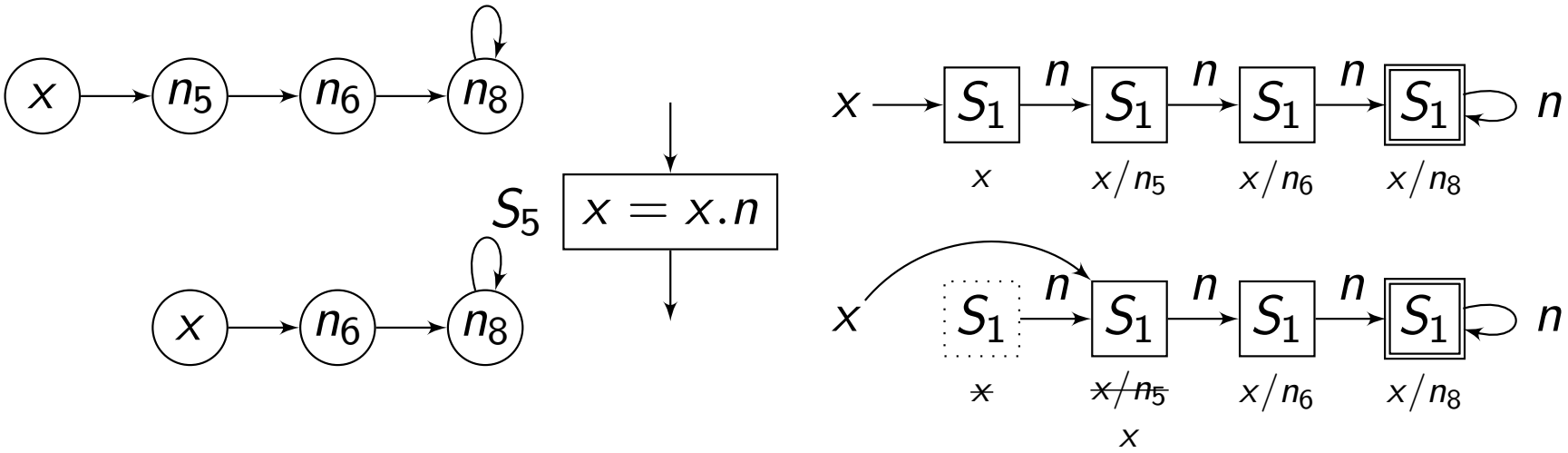
Definition

$\langle E_v, E_f, summary \rangle \sqcap \langle E'_v, E'_f, summary' \rangle = \langle E''_v, E''_f, summary'' \rangle$ such that:

- $E''_v = E_v \cup E'_v$
- $E''_f = E_f \cup E'_f$
- $\forall k \in H : summary''(k) = summary(k) \vee summary'(k)$

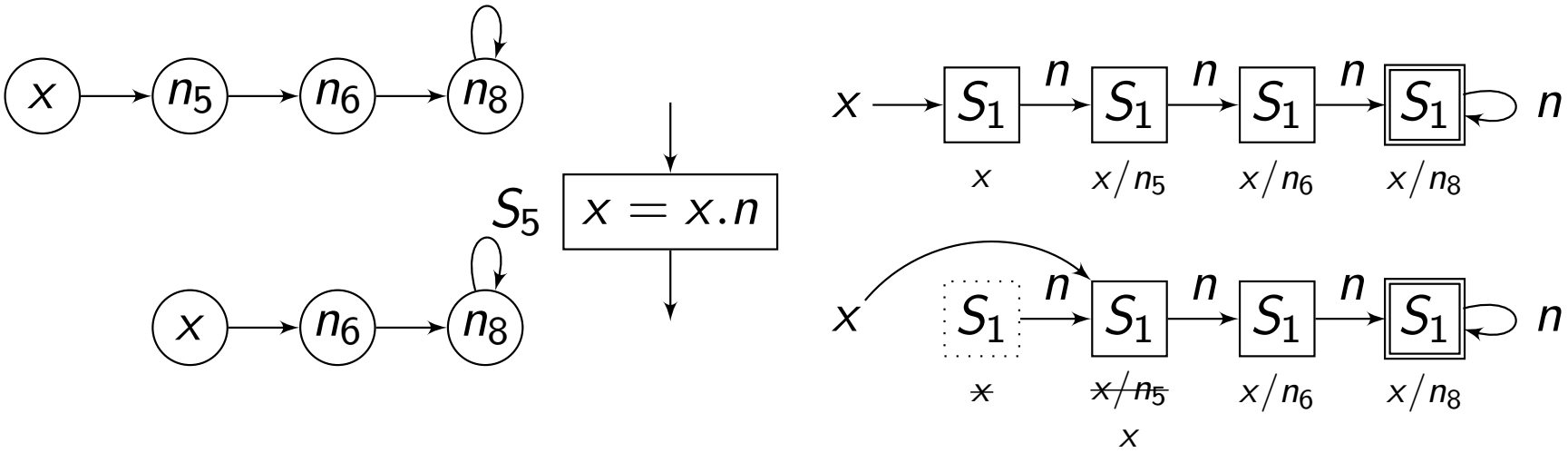
Data Flow Analysis

- Normalization: $\Theta(X, L)$ = Consistency + Reachability



Data Flow Analysis

- Normalization: $\Theta(X, L)$ = Consistency + Reachability



- Data Flow Equations:

$$PTIN_b = \bigsqcap_{p \in pred(b)} \Theta(PTOUT_p, LVIN_b)$$

$$PTOUT_b = \Theta(f_b(PTIN_b), LVOUT_b)$$

System Model

- $\hat{L} : S \times AP \rightarrow \{true, false\}$ (Results of liveness analysis)
- $\hat{P} : S \times AP \times AP \rightarrow \{true, false\}$ (Results of points-to analysis)
- $HLA : \hat{P} \rightarrow \hat{L}$ (Heap Liveness Analysis)
- $PTA : \hat{L} \rightarrow \hat{P}$ (Heap Points-To Analysis)

$$\hat{L}_0 = \lambda s \lambda a.false$$

$$\forall i \geq 0 : \hat{P}_i = PTA(\hat{L}_i)$$

$$\forall i \geq 0 : \hat{L}_{i+1} = HLA(\hat{P}_i)$$

System Model

- $\hat{L} : S \times AP \rightarrow \{true, false\}$ (Results of liveness analysis)
- $\hat{P} : S \times AP \times AP \rightarrow \{true, false\}$ (Results of points-to analysis)
- $HLA : \hat{P} \rightarrow \hat{L}$ (Heap Liveness Analysis)
- $PTA : \hat{L} \rightarrow \hat{P}$ (Heap Points-To Analysis)

$$\hat{L}_0 = \lambda s \lambda a.false$$

$$\forall i \geq 0 : \hat{P}_i = PTA(\hat{L}_i)$$

$$\forall i \geq 0 : \hat{L}_{i+1} = HLA(\hat{P}_i)$$

- $\hat{L}_i \subseteq \hat{L}_j$ iff $\forall s \in S, \forall a \in AP : \hat{L}_i(s, a) \Rightarrow \hat{L}_j(s, a)$
- $\hat{P}_i \subseteq \hat{P}_j$ iff $\forall s \in S, \forall a \in AP, \forall b \in AP : \hat{P}_i(s, a, b) \Rightarrow \hat{P}_j(s, a, b)$

Precision of Liveness

$$\hat{L}_0\ \hat{P}_0\ \hat{L}_1\ \hat{P}_1\ \hat{L}_2\ \hat{P}_2\ \hat{L}_3\ \hat{P}_3\ \hat{L}_4 \cdots$$

Precision of Liveness

$$\hat{L}_0\ \hat{P}_0\ \hat{L}_1\ \hat{P}_1\ \hat{L}_2\ \hat{P}_2\ \hat{L}_3\ \hat{P}_3\ \hat{L}_4 \cdots$$

Theorem

The results of the second round of heap liveness analysis is the most precise result which is also sound. That is, $\forall k > 0 : \hat{L}_2 \subseteq \hat{L}_k$.

Precision of Liveness

$$\hat{L}_0\ \hat{P}_0\ \hat{L}_1\ \hat{P}_1\ \hat{L}_2\ \hat{P}_2\ \hat{L}_3\ \hat{P}_3\ \hat{L}_4 \cdots$$

Theorem

The results of the second round of heap liveness analysis is the most precise result which is also sound. That is, $\forall k > 0 : \hat{L}_2 \subseteq \hat{L}_k$.

Lemma (1)

$\forall i, j : \hat{L}_i \subseteq \hat{L}_j \Rightarrow \hat{P}_i \supseteq \hat{P}_j$

Lemma (2)

$\forall i, j : \hat{P}_i \subseteq \hat{P}_j \Rightarrow \hat{L}_{i+1} \subseteq \hat{L}_{j+1}$

Precision of Liveness

$$\hat{L}_0\ \hat{P}_0\ \hat{L}_1\ \hat{P}_1\ \hat{L}_2\ \hat{P}_2\ \hat{L}_3\ \hat{P}_3\ \hat{L}_4 \cdots$$

Theorem

The results of the second round of heap liveness analysis is the most precise result which is also sound. That is, $\forall k > 0 : \hat{L}_2 \subseteq \hat{L}_k$.

Lemma (1)

$\forall i, j : \hat{L}_i \subseteq \hat{L}_j \Rightarrow \hat{P}_i \supseteq \hat{P}_j$

Lemma (2)

$\forall i, j : \hat{P}_i \subseteq \hat{P}_j \Rightarrow \hat{L}_{i+1} \subseteq \hat{L}_{j+1}$

Proof.

1. $\forall k \geq 0 : \hat{L}_0 \subseteq \hat{L}_k$ (By Definition)
2. $\forall k \geq 0 : \hat{P}_0 \supseteq \hat{P}_k$ (Lemma 1)
3. $\forall k \geq 0 : \hat{L}_1 \supseteq \hat{L}_{k+1}$ (Lemma 2)
4. $\forall k \geq 0 : \hat{P}_1 \subseteq \hat{P}_{k+1}$ (Lemma 1)
5. $\forall k \geq 0 : \hat{L}_2 \subseteq \hat{L}_{k+2}$ (Lemma 2)
6. $\forall k > 0 : \hat{L}_2 \subseteq \hat{L}_k$ (Step 3 and 5)

□

Outline

Interprocedural Analysis

- No existing implementation of inter-procedural heap reference analysis

Interprocedural Analysis

- No existing implementation of inter-procedural heap reference analysis
- Soot has excellent API for data flow analysis - only intraprocedural

Interprocedural Analysis

- No existing implementation of inter-procedural heap reference analysis
- Soot has excellent API for data flow analysis - only intraprocedural
- IFDS/IDE solver [Bodden, SOAP 2012]

Interprocedural Analysis

- No existing implementation of inter-procedural heap reference analysis
- Soot has excellent API for data flow analysis - only intraprocedural
- IFDS/IDE solver [Bodden, SOAP 2012]
  - $f(\{x, y, z\}) = f(\{x\}) \sqcap f(\{y\}) \sqcap f(\{z\})$

Interprocedural Analysis

- No existing implementation of inter-procedural heap reference analysis
- Soot has excellent API for data flow analysis - only intraprocedural
- IFDS/IDE solver [Bodden, SOAP 2012]
  - $f(\{x, y, z\}) = f(\{x\}) \sqcap f(\{y\}) \sqcap f(\{z\})$
  - Functions on 2^D reduced to functions on D

Interprocedural Analysis

- No existing implementation of inter-procedural heap reference analysis
- Soot has excellent API for data flow analysis - only intraprocedural
- IFDS/IDE solver [Bodden, SOAP 2012]
  - $f(\{x, y, z\}) = f(\{x\}) \sqcap f(\{y\}) \sqcap f(\{z\})$
  - Functions on 2^D reduced to functions on D
  - Modelled as a graph reachability problem

Interprocedural Analysis

- No existing implementation of inter-procedural heap reference analysis
- Soot has excellent API for data flow analysis - only intraprocedural
- IFDS/IDE solver [Bodden, SOAP 2012]
  - $f(\{x, y, z\}) = f(\{x\}) \sqcap f(\{y\}) \sqcap f(\{z\})$
  - Functions on 2^D reduced to functions on D
  - Modelled as a graph reachability problem
  - Main limitation: Requires distributive flow functions

Interprocedural Analysis

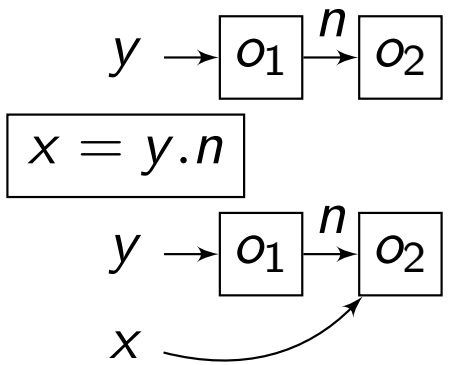
- No existing implementation of inter-procedural heap reference analysis
- Soot has excellent API for data flow analysis - only intraprocedural
- IFDS/IDE solver [Bodden, SOAP 2012]
  - $f(\{x, y, z\}) = f(\{x\}) \sqcap f(\{y\}) \sqcap f(\{z\})$
  - Functions on 2^D reduced to functions on D
  - Modelled as a graph reachability problem
  - Main limitation: Requires distributive flow functions
  - Not suitable for many types of heap analysis

Interprocedural Analysis

- No existing implementation of inter-procedural heap reference analysis
- Soot has excellent API for data flow analysis - only intraprocedural
- IFDS/IDE solver [Bodden, SOAP 2012]
  - $f(\{x, y, z\}) = f(\{x\}) \sqcap f(\{y\}) \sqcap f(\{z\})$
  - Functions on 2^D reduced to functions on D
  - Modelled as a graph reachability problem
  - Main limitation: Requires distributive flow functions
  - Not suitable for many types of heap analysis

Interprocedural Analysis

- No existing implementation of inter-procedural heap reference analysis
- Soot has excellent API for data flow analysis - only intraprocedural
- IFDS/IDE solver [Bodden, SOAP 2012]
  - $f(\{x, y, z\}) = f(\{x\}) \sqcap f(\{y\}) \sqcap f(\{z\})$
  - Functions on 2^D reduced to functions on D
  - Modelled as a graph reachability problem
  - Main limitation: Requires distributive flow functions
  - Not suitable for many types of heap analysis



Interprocedural Analysis

The most general and precise solutions:

Interprocedural Analysis

The most general and precise solutions:

- Call Strings (maintain an abstract call stack) [Sharir & Pnueli, 1981]

Interprocedural Analysis

The most general and precise solutions:

- Call Strings (maintain an abstract call stack) [Sharir & Pnueli, 1981]
- Functional (flow functions for call statements) [Sharir & Pnueli, 1981]

Interprocedural Analysis

The most general and precise solutions:

- Call Strings (maintain an abstract call stack) [Sharir & Pnueli, 1981]
- Functional (flow functions for call statements) [Sharir & Pnueli, 1981]
  - Function composition method
  - Tabulation method

Interprocedural Analysis

The most general and precise solutions:

- Call Strings (maintain an abstract call stack) [Sharir & Pnueli, 1981]
- Functional (flow functions for call statements) [Sharir & Pnueli, 1981]
  - Function composition method
  - Tabulation method
- Modified call-strings method [Khedker & Karkare, 2008]

Interprocedural Analysis

The most general and precise solutions:

- Call Strings (maintain an abstract call stack) [Sharir & Pnueli, 1981]
- Functional (flow functions for call statements) [Sharir & Pnueli, 1981]
  - Function composition method
  - Tabulation method
- Modified call-strings method [Khedker & Karkare, 2008]
  - Value-based termination of call string construction

Interprocedural Analysis

The most general and precise solutions:

- Call Strings (maintain an abstract call stack) [Sharir & Pnueli, 1981]
- Functional (flow functions for call statements) [Sharir & Pnueli, 1981]
  - Function composition method
  - Tabulation method
- Modified call-strings method [Khedker & Karkare, 2008]
  - Value-based termination of call string construction
- Value contexts [Padhye & Khedker, 2013]

Interprocedural Analysis

The most general and precise solutions:

- Call Strings (maintain an abstract call stack) [Sharir & Pnueli, 1981]
- Functional (flow functions for call statements) [Sharir & Pnueli, 1981]
  - Function composition method
  - Tabulation method
- Modified call-strings method [Khedker & Karkare, 2008]
  - Value-based termination of call string construction
- Value contexts [Padhye & Khedker, 2013]
  - Reformulation of tabulation method

Interprocedural Analysis

The most general and precise solutions:

- Call Strings (maintain an abstract call stack) [Sharir & Pnueli, 1981]
- Functional (flow functions for call statements) [Sharir & Pnueli, 1981]
  - Function composition method
  - Tabulation method
- Modified call-strings method [Khedker & Karkare, 2008]
  - Value-based termination of call string construction
- Value contexts [Padhye & Khedker, 2013]
  - Reformulation of tabulation method
  - Suitable for bi-directional interleaved analyses

Interprocedural Analysis

The most general and precise solutions:

- Call Strings (maintain an abstract call stack) [Sharir & Pnueli, 1981]
- Functional (flow functions for call statements) [Sharir & Pnueli, 1981]
  - Function composition method
  - Tabulation method
- Modified call-strings method [Khedker & Karkare, 2008]
  - Value-based termination of call string construction
- Value contexts [Padhye & Khedker, 2013]
  - Reformulation of tabulation method
  - Suitable for bi-directional interleaved analyses
  - Can map arbitrary call string to value context (dynamic optimizations)

Interprocedural Analysis

The most general and precise solutions:

- Call Strings (maintain an abstract call stack) [Sharir & Pnueli, 1981]
- Functional (flow functions for call statements) [Sharir & Pnueli, 1981]
  - Function composition method
  - Tabulation method
- Modified call-strings method [Khedker & Karkare, 2008]
  - Value-based termination of call string construction
- Value contexts [Padhye & Khedker, 2013]
  - Reformulation of tabulation method
  - Suitable for bi-directional interleaved analyses
  - Can map arbitrary call string to value context (dynamic optimizations)
  - Context-sensitive data flow solution (specialization)

Value Contexts

Value contexts:

Value Contexts

Value contexts:

- $X = \langle method, entryValue \rangle$

Value Contexts

Value contexts:

- $X = \langle method, entryValue \rangle$
- $exitValue(X)$

Value Contexts

Value contexts:

- $X = \langle method, entryValue \rangle$
- $exitValue(X)$
- Data Flow Analysis is performed using traditional work-list method

Value Contexts

Value contexts:

- $X = \langle method, entryValue \rangle$
- $exitValue(X)$
- Data Flow Analysis is performed using traditional work-list method
- Work-list contains $\langle context, node \rangle$ pairs

Value Contexts

Value contexts:

- $X = \langle method, entryValue \rangle$
- $exitValue(X)$
- Data Flow Analysis is performed using traditional work-list method
- Work-list contains $\langle context, node \rangle$ pairs
- Call-sites: Find value context $X = \langle method, entryValue \rangle$

Value Contexts

Value contexts:

- $X = \langle method, entryValue \rangle$
- $exitValue(X)$
- Data Flow Analysis is performed using traditional work-list method
- Work-list contains $\langle context, node \rangle$ pairs
- Call-sites: Find value context $X = \langle method, entryValue \rangle$
  - Found: Re-use $exitValue(X)$

Value Contexts

Value contexts:

- $X = \langle method, entryValue \rangle$
- $exitValue(X)$
- Data Flow Analysis is performed using traditional work-list method
- Work-list contains $\langle context, node \rangle$ pairs
- Call-sites: Find value context $X = \langle method, entryValue \rangle$
  - Found: Re-use $exitValue(X)$
  - Not found: Create new X and add all nodes to work-list

Value Contexts

Value contexts:

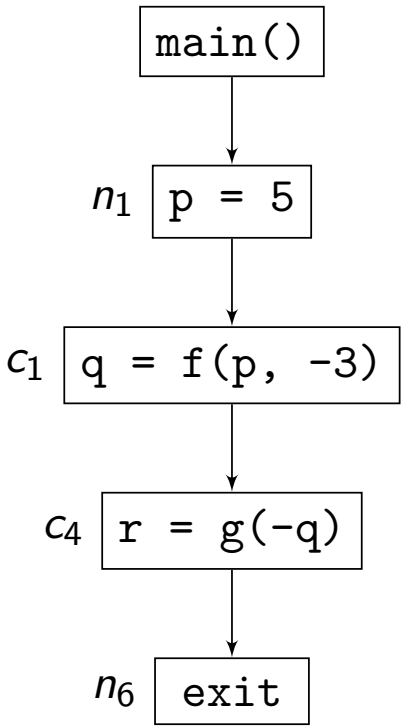
- $X = \langle method, entryValue \rangle$
- $exitValue(X)$
- Data Flow Analysis is performed using traditional work-list method
- Work-list contains $\langle context, node \rangle$ pairs
- Call-sites: Find value context $X = \langle method, entryValue \rangle$
  - Found: Re-use $exitValue(X)$
  - Not found: Create new X and add all nodes to work-list
  - Record transition from this call-site to X

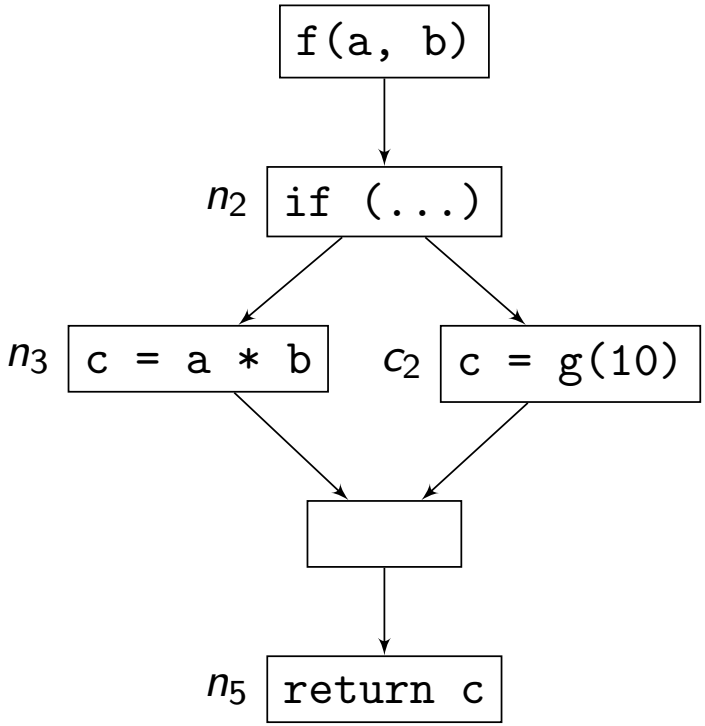
Value Contexts

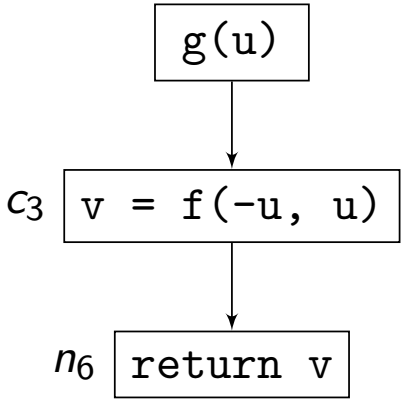
Value contexts:

- $X = \langle method, entryValue \rangle$
- $exitValue(X)$
- Data Flow Analysis is performed using traditional work-list method
- Work-list contains $\langle context, node \rangle$ pairs
- Call-sites: Find value context $X = \langle method, entryValue \rangle$
  - Found: Re-use $exitValue(X)$
  - Not found: Create new X and add all nodes to work-list
  - Record transition from this call-site to X
- Exit-sites: Set $exitValue(X)$ and add callers to work-list

Example - Sign Analysis

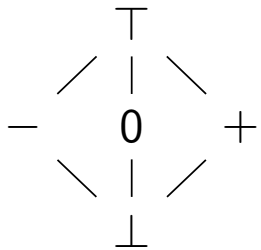






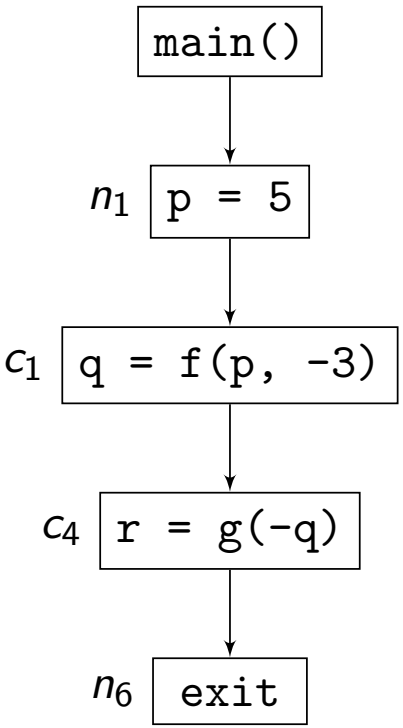
Context	Proc.	Entry	Exit

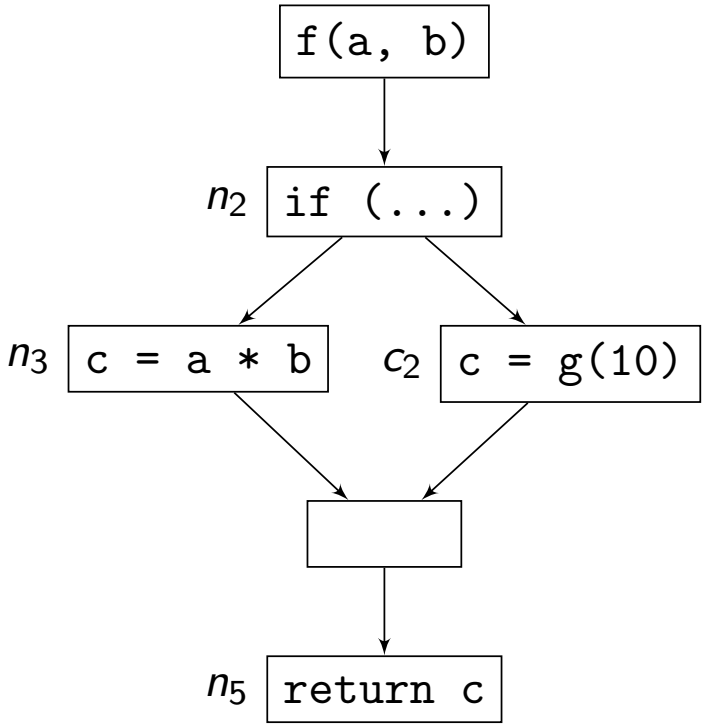
Value Contexts

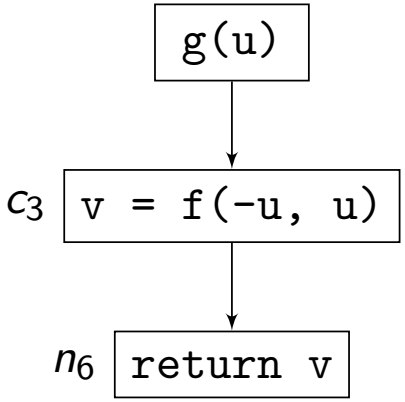


Component Lattice

Example - Sign Analysis

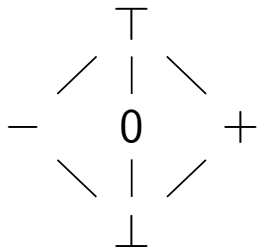






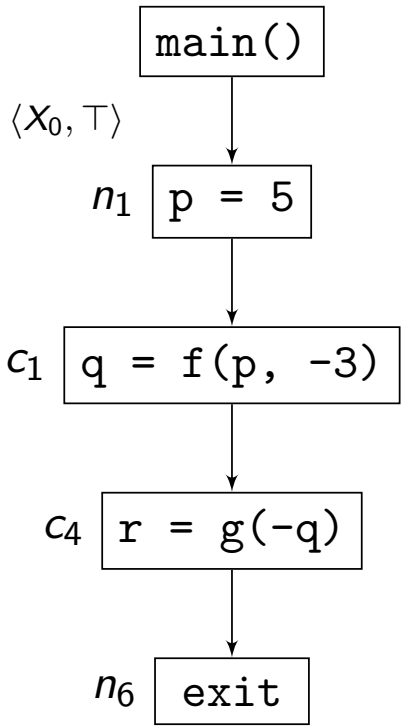
Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$

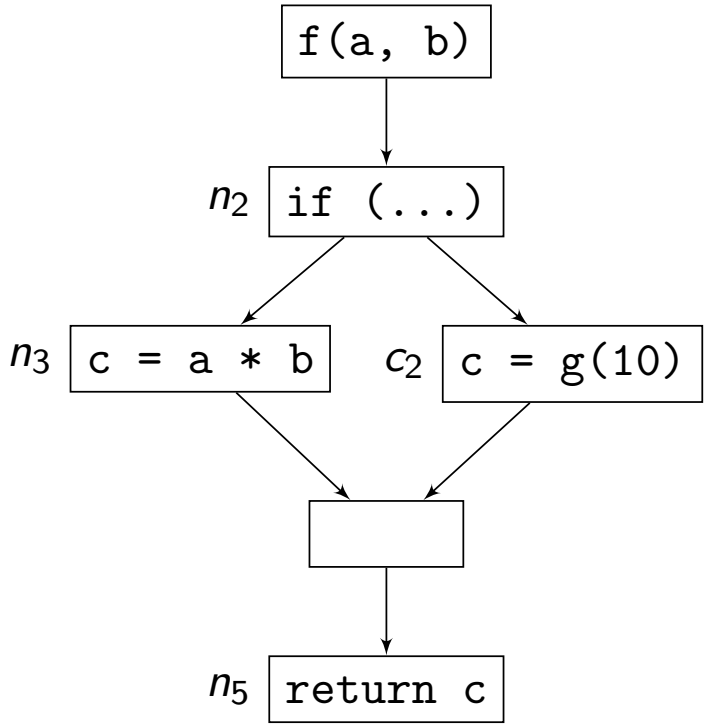
Value Contexts

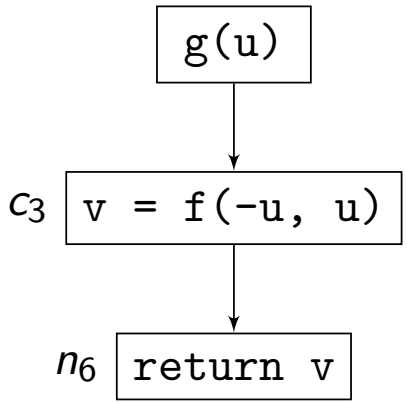


Component Lattice

Example - Sign Analysis

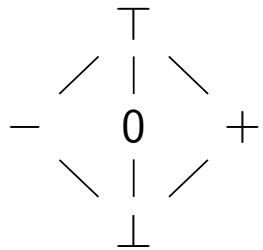






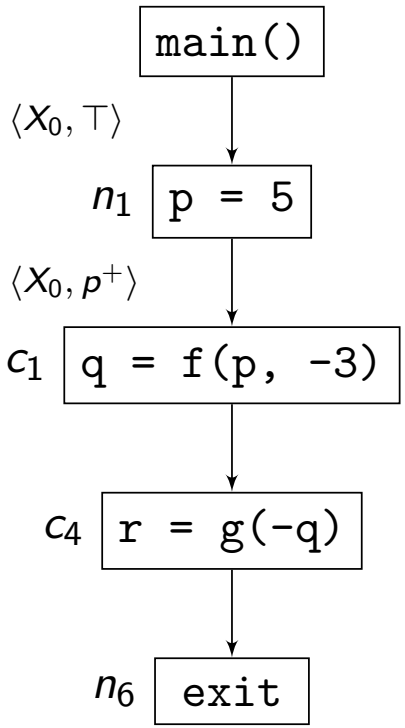
Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$

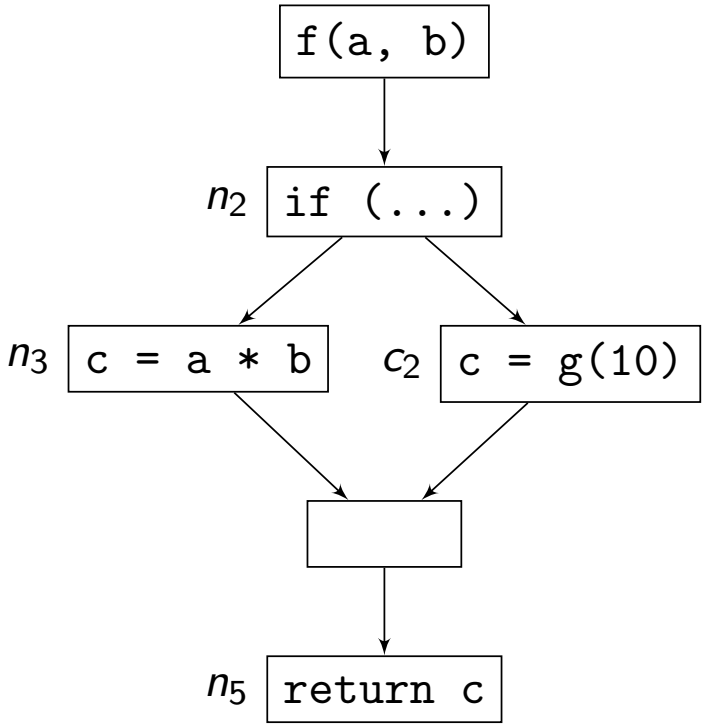
Value Contexts

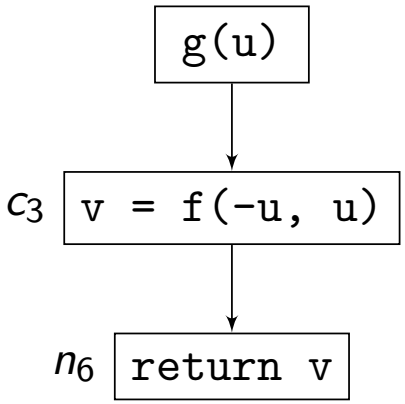


Component Lattice

Example - Sign Analysis

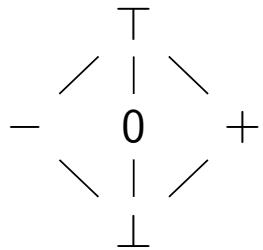






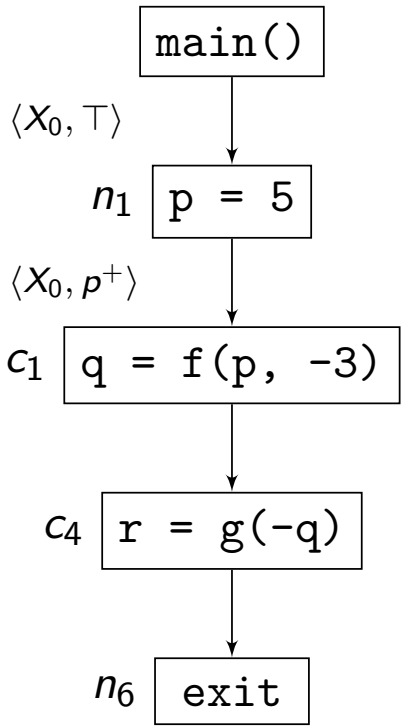
Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$

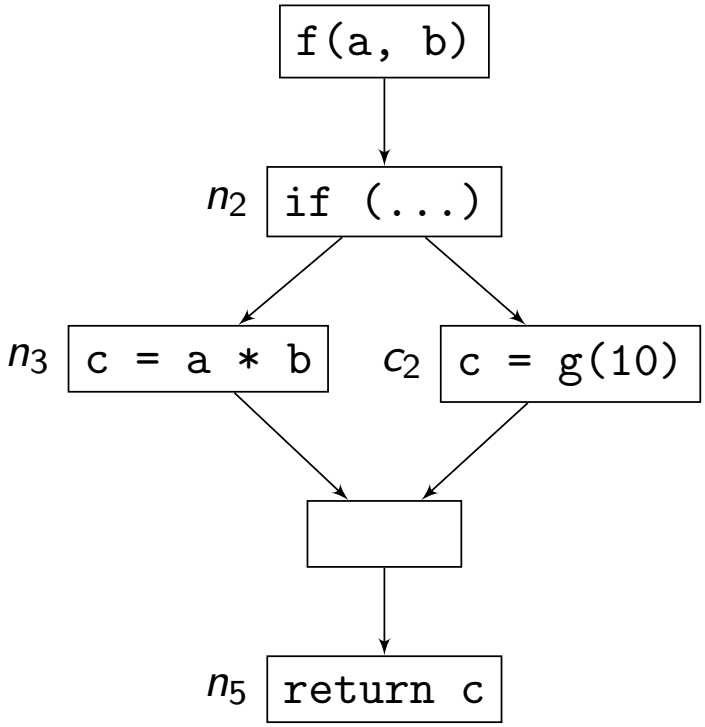
Value Contexts

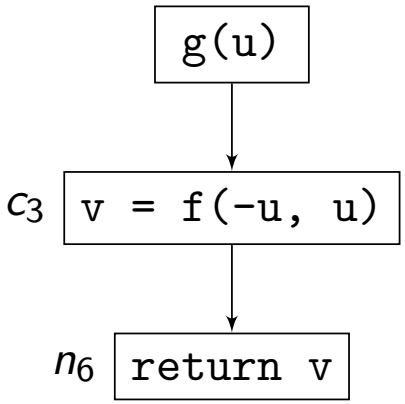


Component Lattice

Example - Sign Analysis

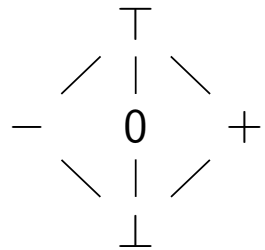






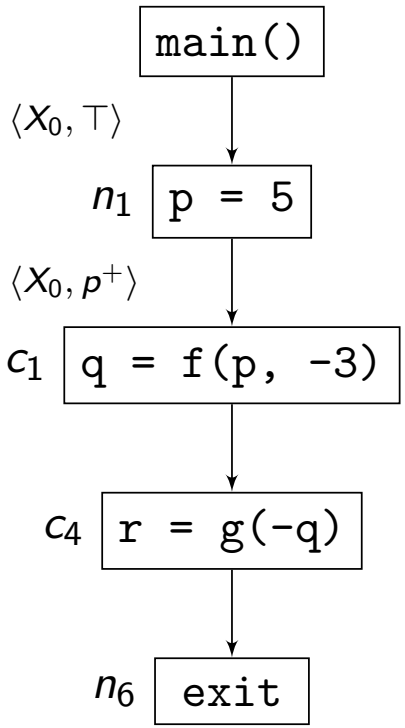
Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	$a^+ b^-$	$\top$

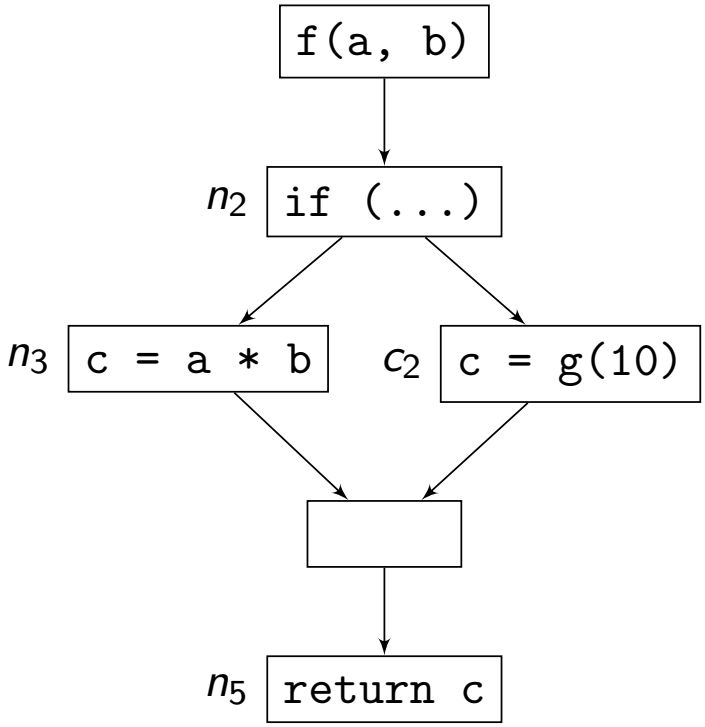
Value Contexts

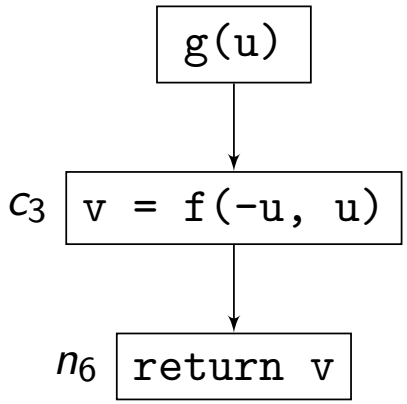


Component Lattice

Example - Sign Analysis

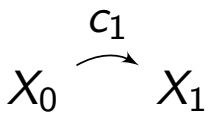




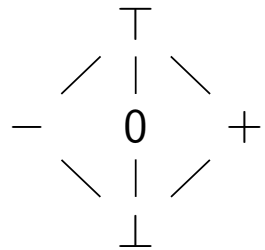


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	$a^+ b^-$	$\top$

Value Contexts

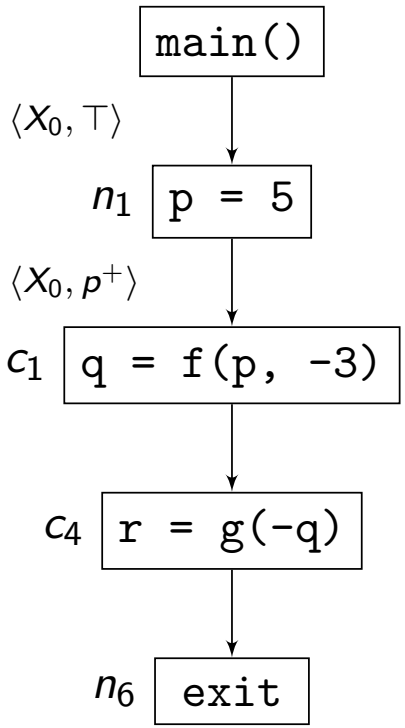


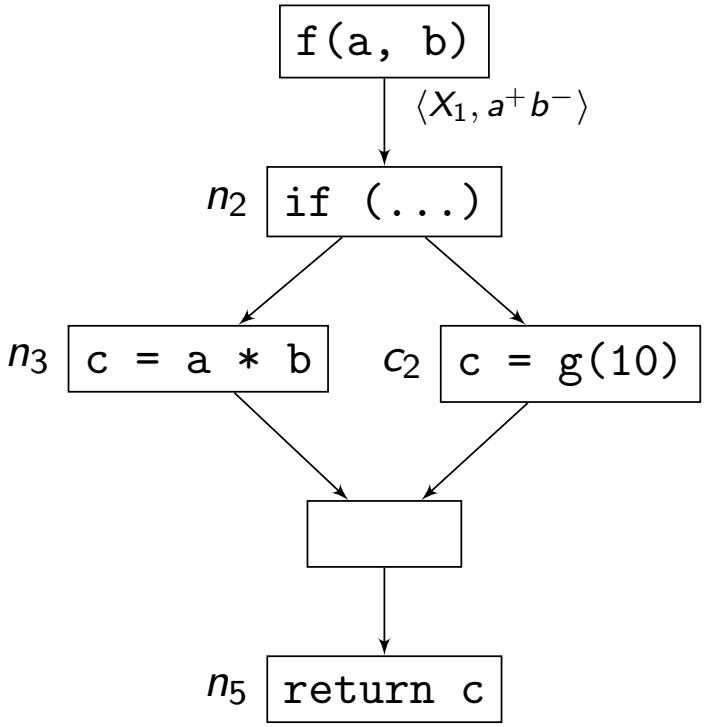
Context Transitions

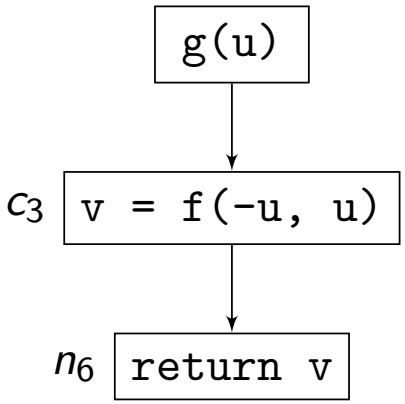


Component Lattice

Example - Sign Analysis

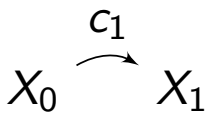




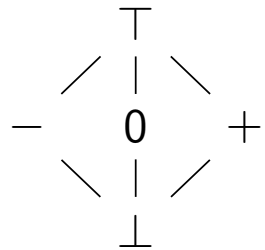


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$

Value Contexts

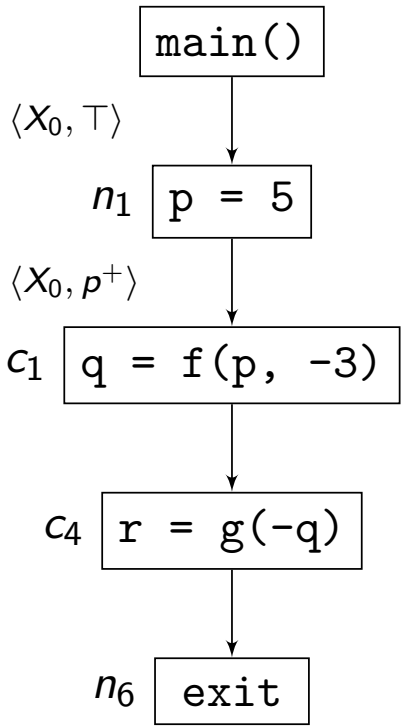


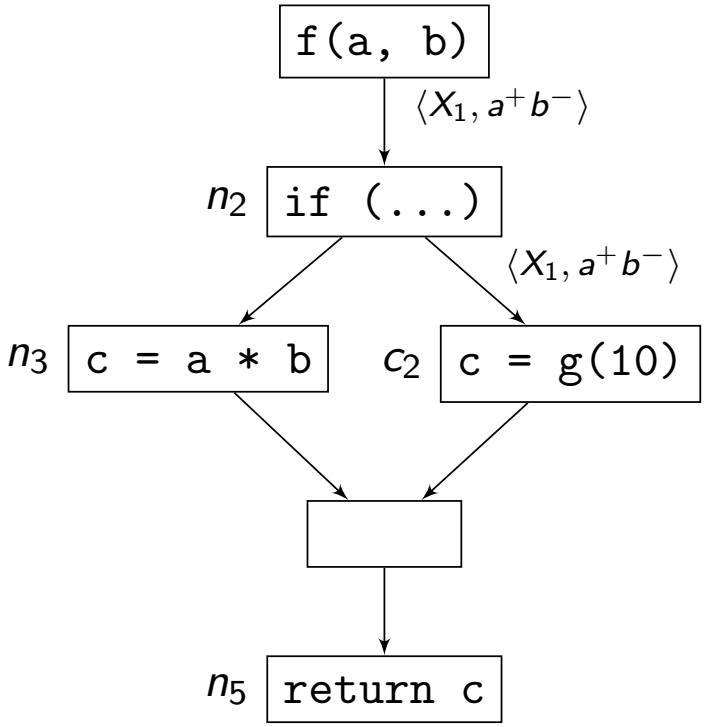
Context Transitions

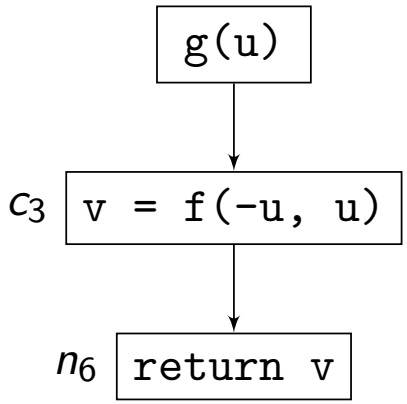


Component Lattice

Example - Sign Analysis

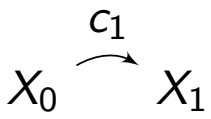




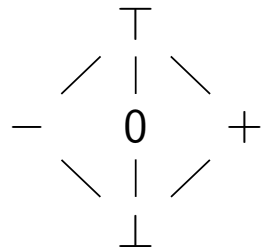


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$

Value Contexts

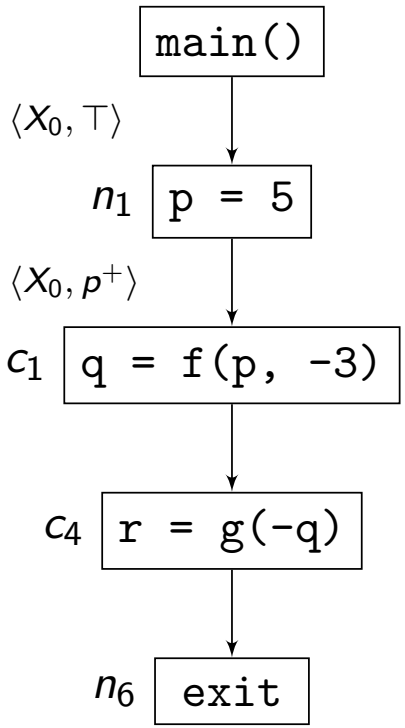


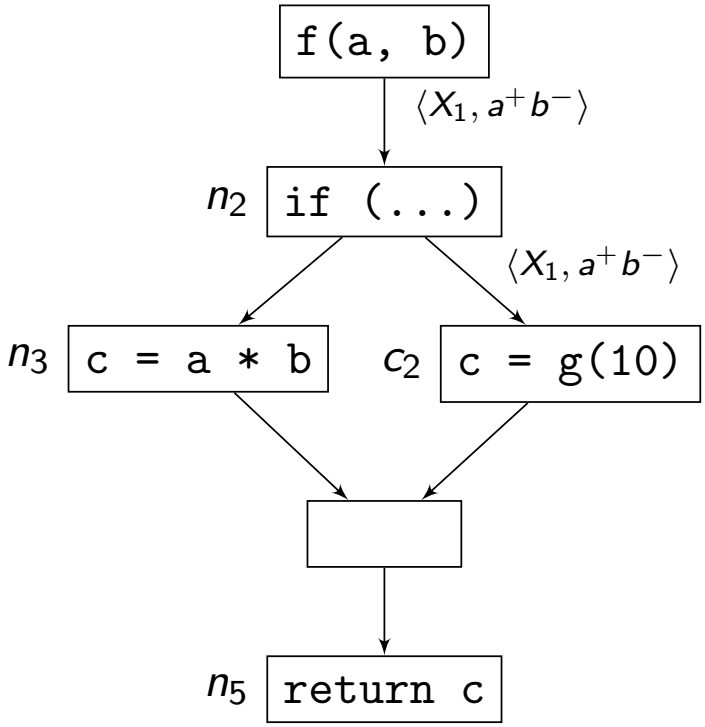
Context Transitions

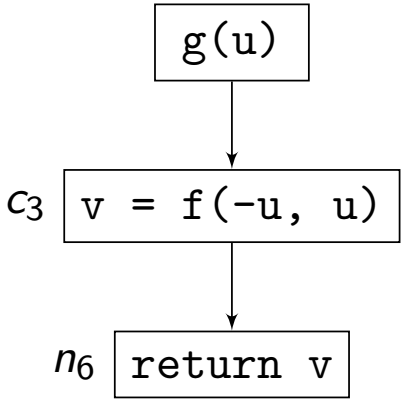


Component Lattice

Example - Sign Analysis

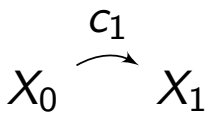




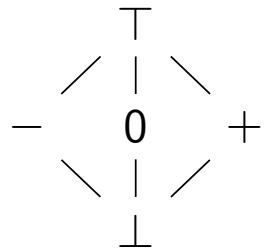


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$
X_2	g	u^+	$\top$

Value Contexts

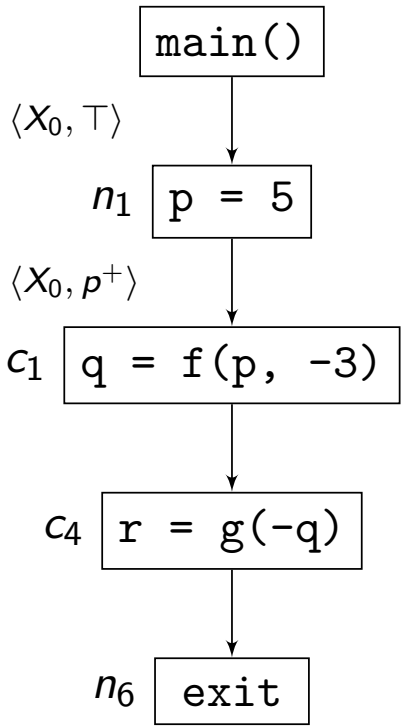


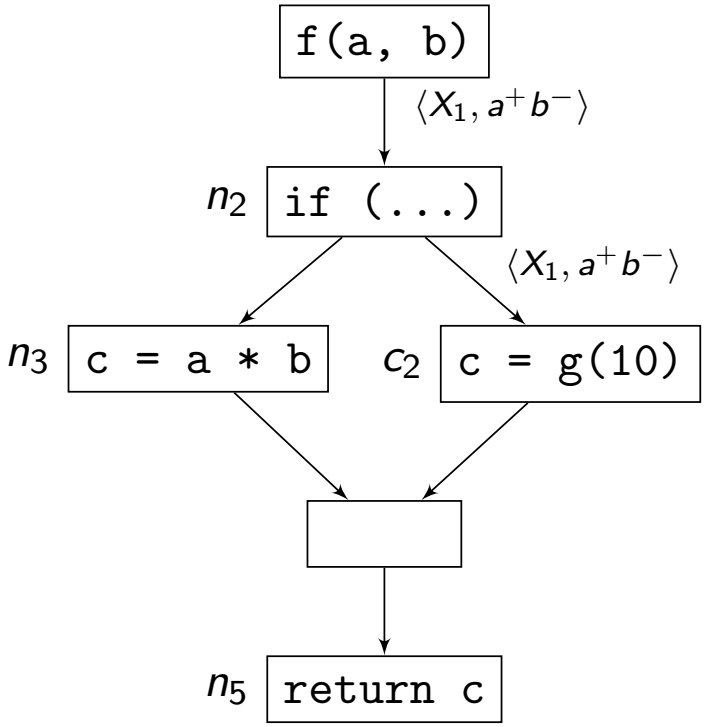
Context Transitions

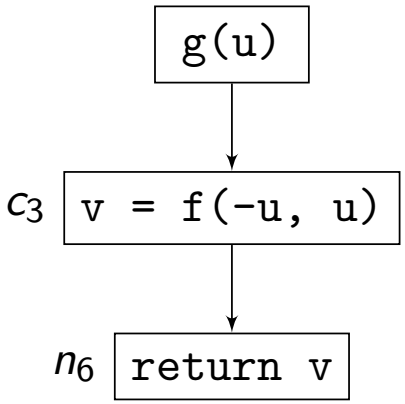


Component Lattice

Example - Sign Analysis

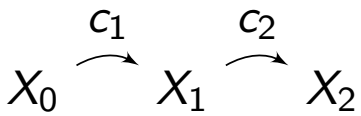




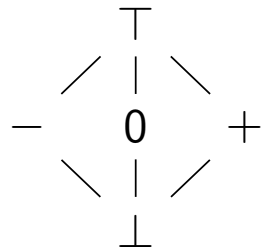


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$
X_2	g	u^+	$\top$

Value Contexts

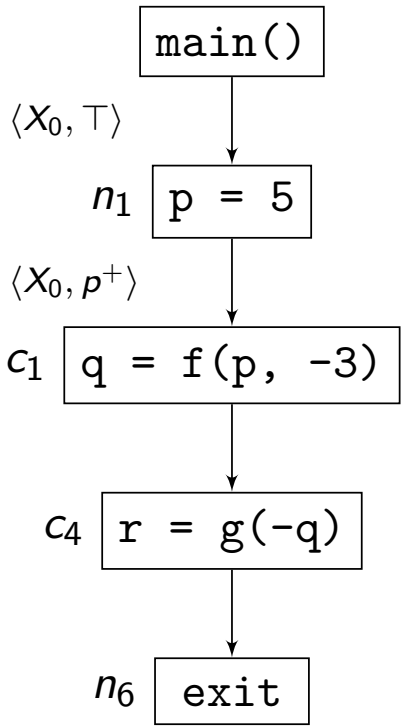


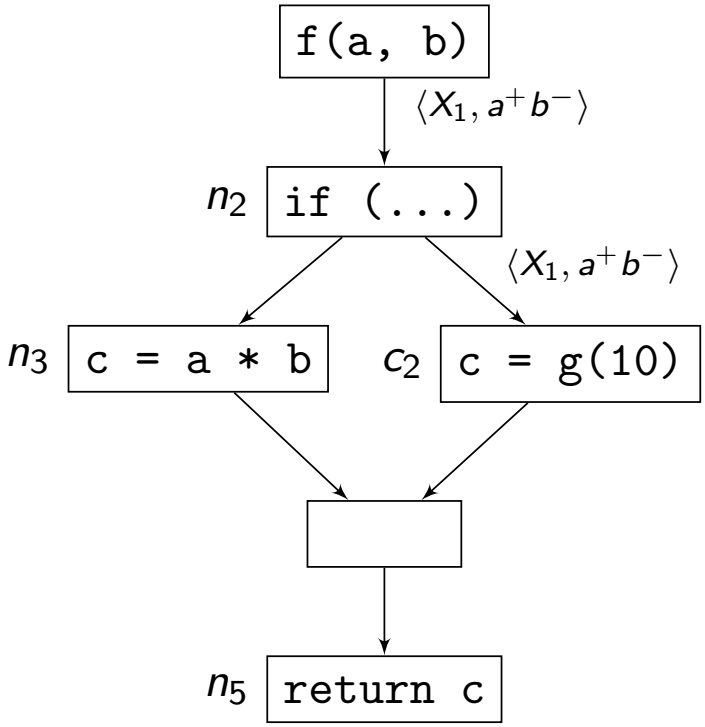
Context Transitions

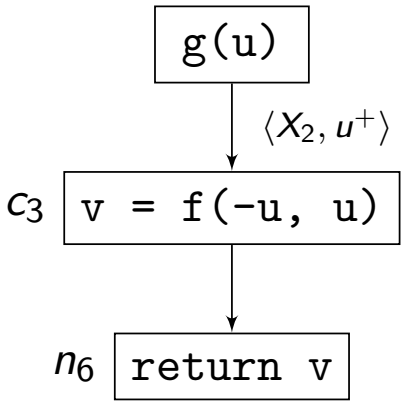


Component Lattice

Example - Sign Analysis

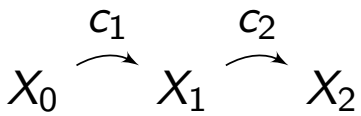




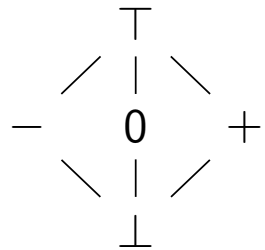


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$
X_2	g	u^+	$\top$

Value Contexts

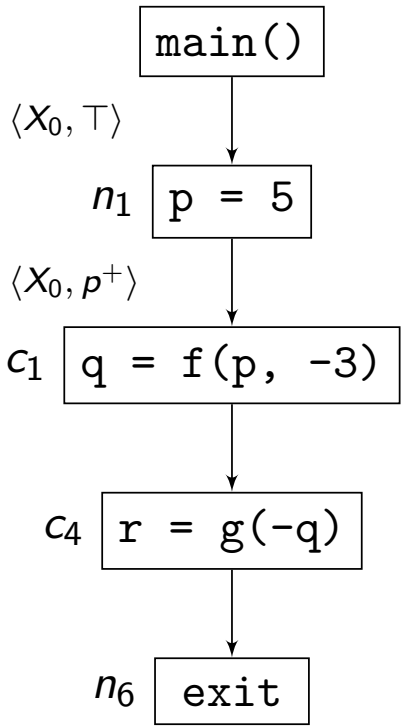


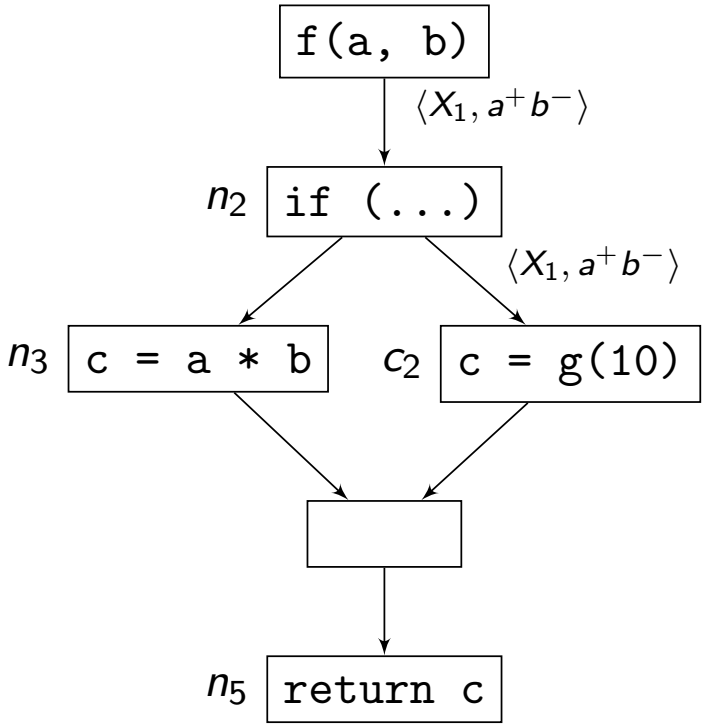
Context Transitions

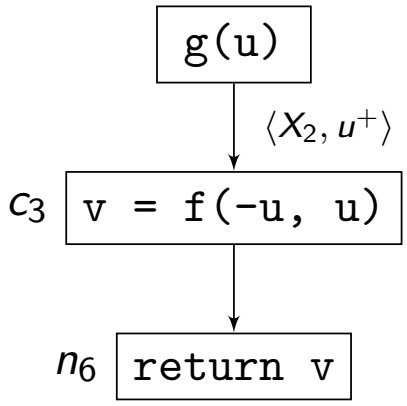


Component Lattice

Example - Sign Analysis

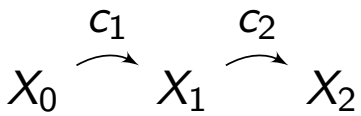




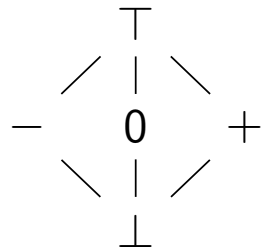


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$
X_2	g	u^+	$\top$
X_3	f	a^-b^+	$\top$

Value Contexts

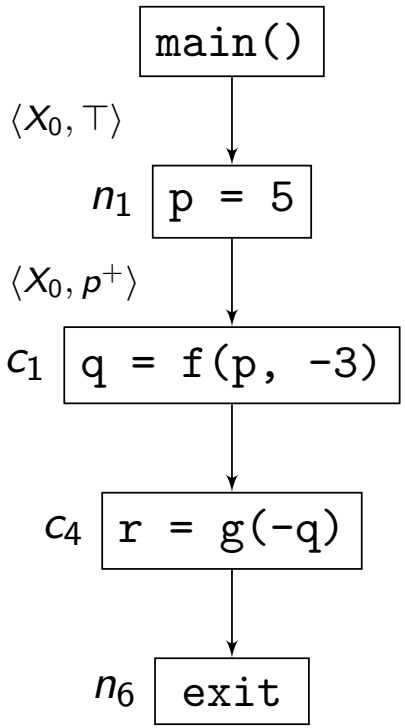


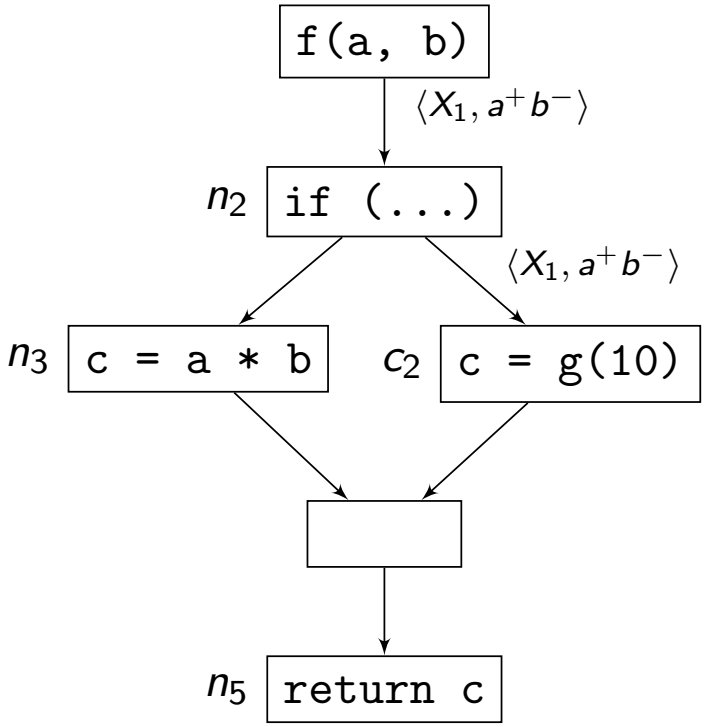
Context Transitions

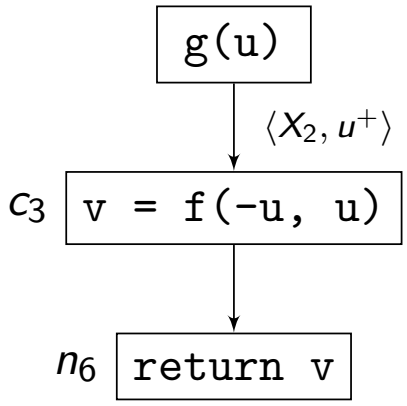


Component Lattice

Example - Sign Analysis

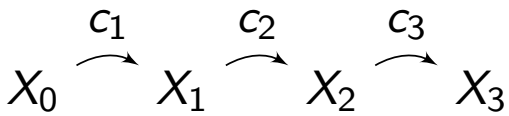




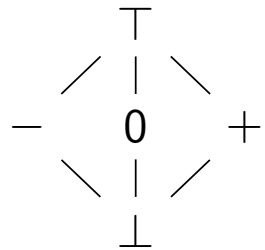


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$
X_2	g	u^+	$\top$
X_3	f	a^-b^+	$\top$

Value Contexts

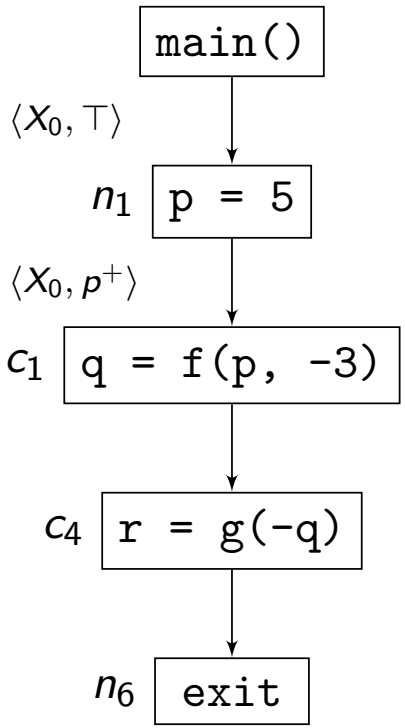


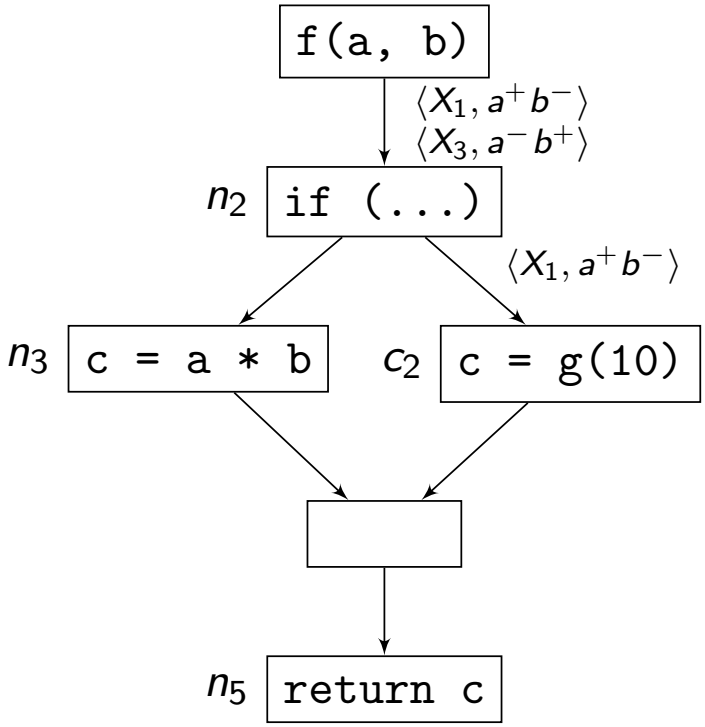
Context Transitions

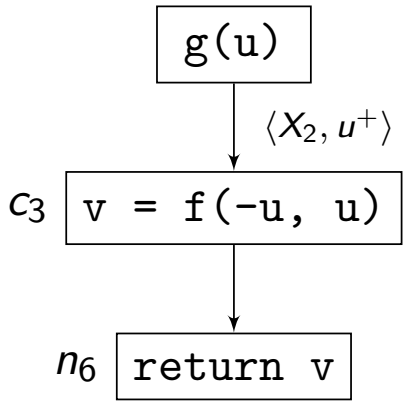


Component Lattice

Example - Sign Analysis

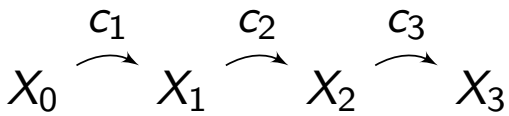




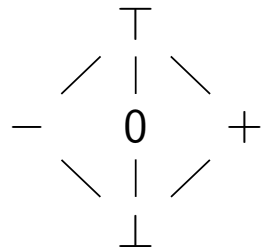


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$
X_2	g	u^+	$\top$
X_3	f	a^-b^+	$\top$

Value Contexts

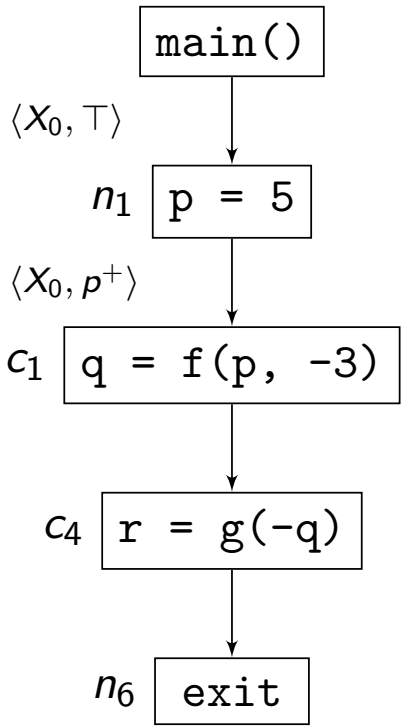


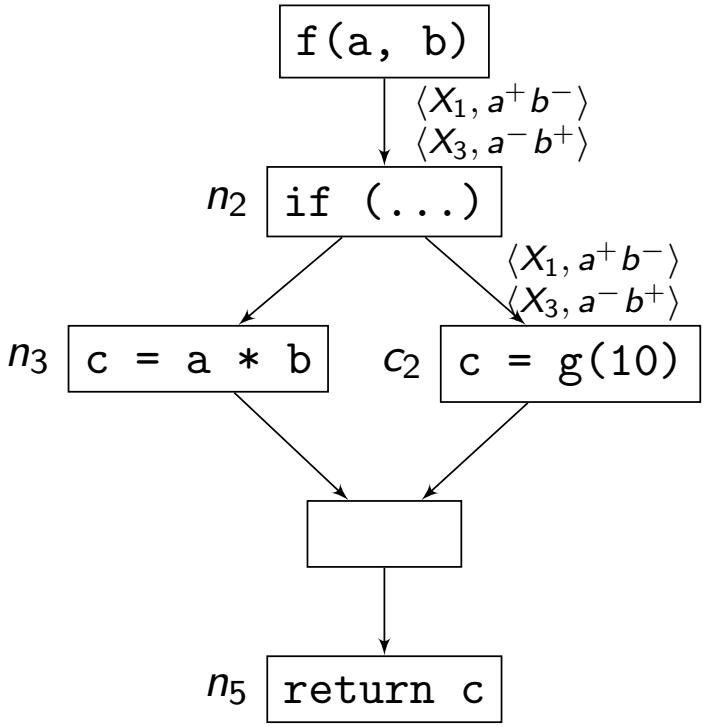
Context Transitions

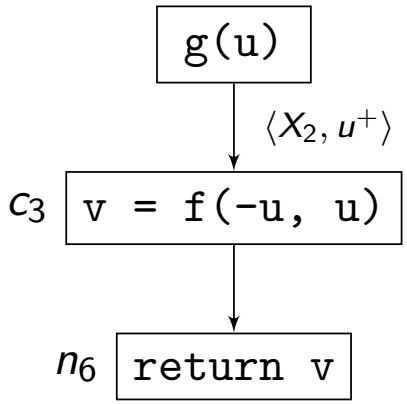


Component Lattice

Example - Sign Analysis

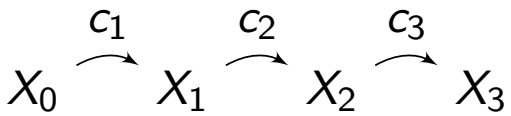




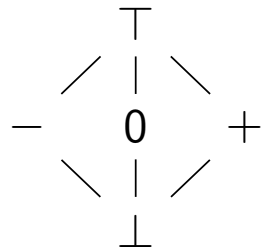


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$
X_2	g	u^+	$\top$
X_3	f	a^-b^+	$\top$

Value Contexts

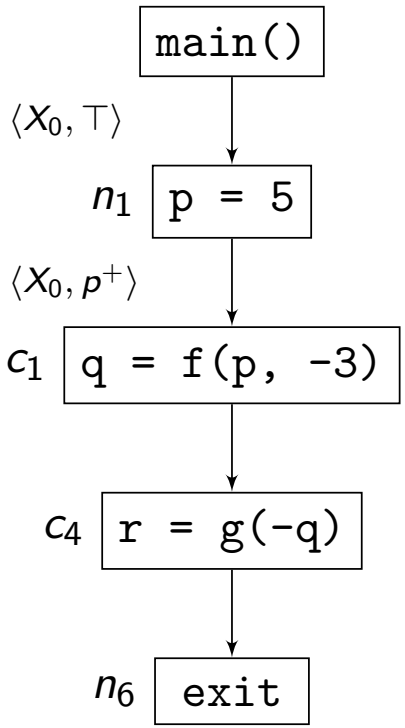


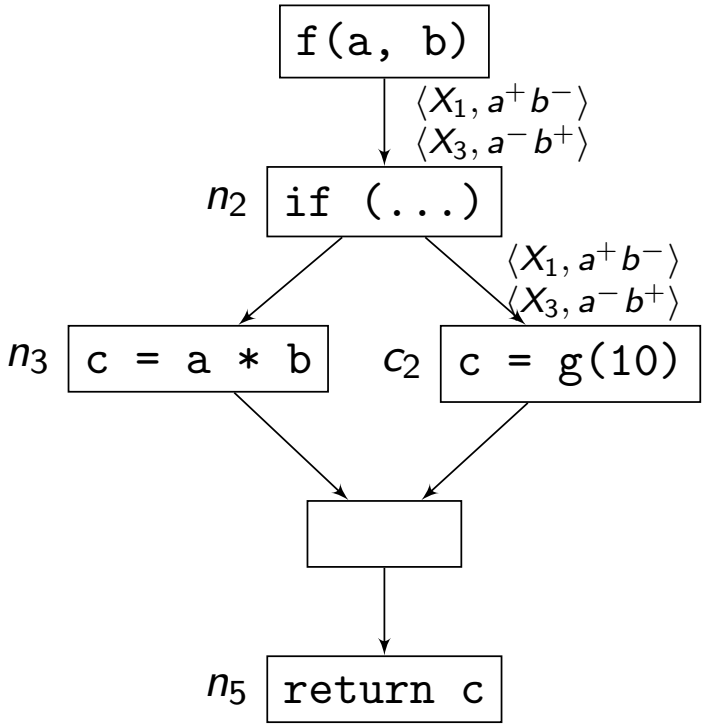
Context Transitions

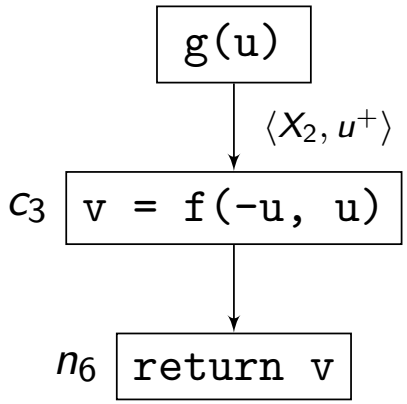


Component Lattice

Example - Sign Analysis

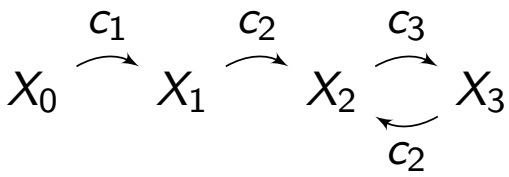




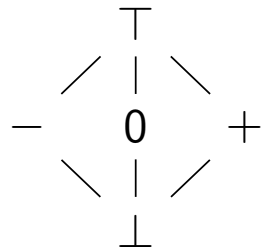


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$
X_2	g	u^+	$\top$
X_3	f	a^-b^+	$\top$

Value Contexts

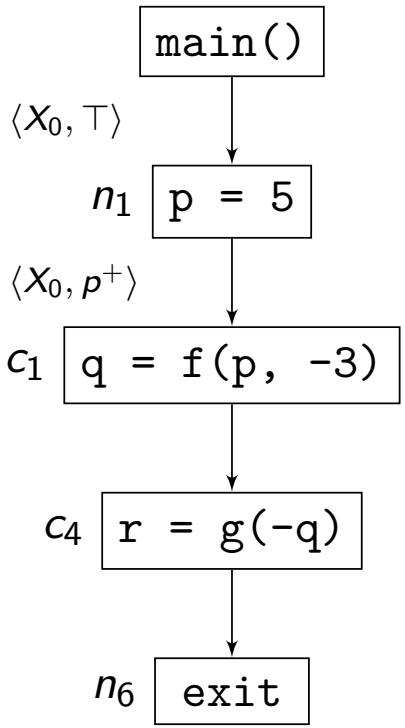


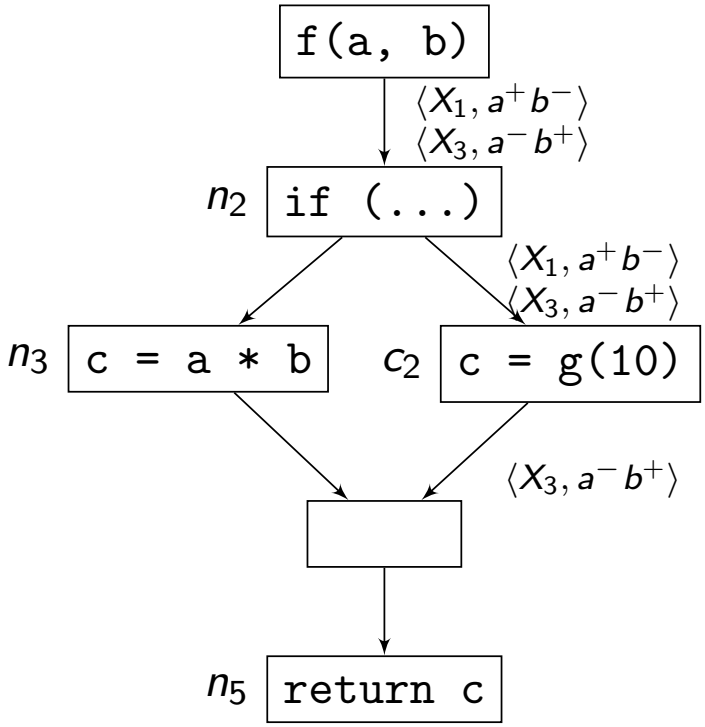
Context Transitions

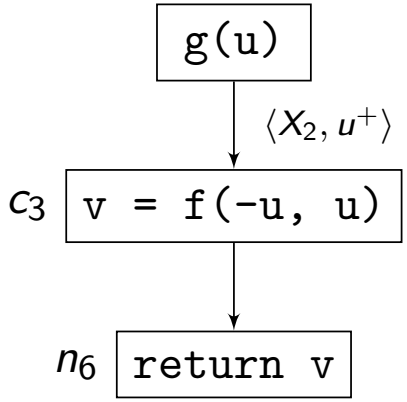


Component Lattice

Example - Sign Analysis

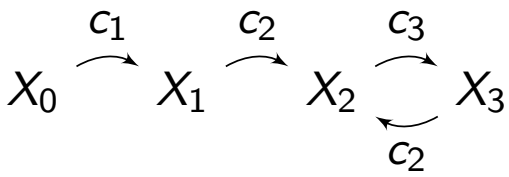




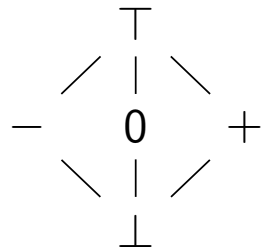


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$
X_2	g	u^+	$\top$
X_3	f	a^-b^+	$\top$

Value Contexts

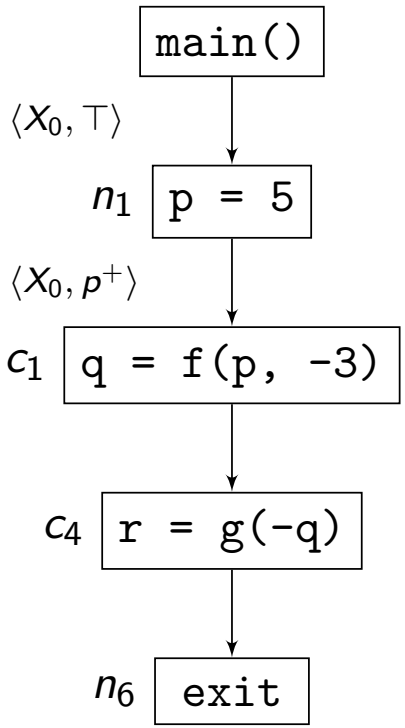


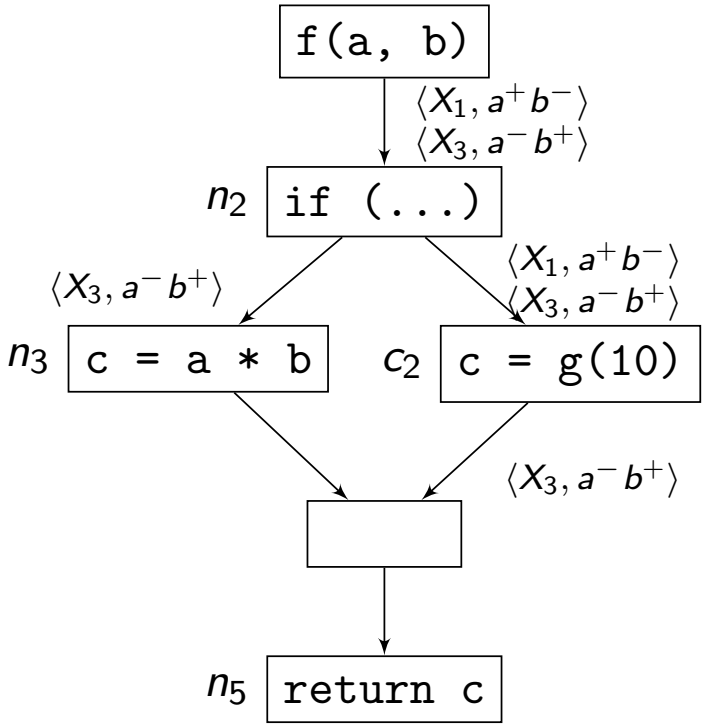
Context Transitions

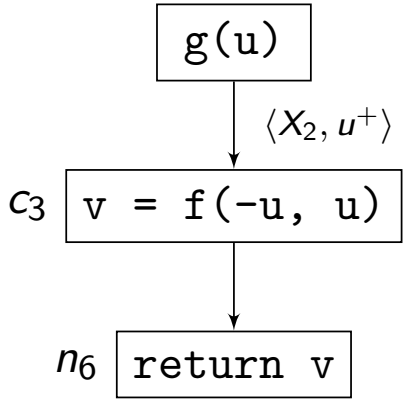


Component Lattice

Example - Sign Analysis

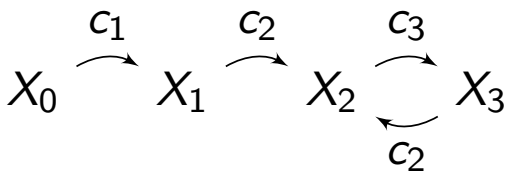




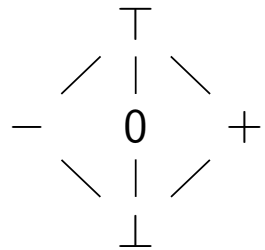


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$
X_2	g	u^+	$\top$
X_3	f	a^-b^+	$\top$

Value Contexts

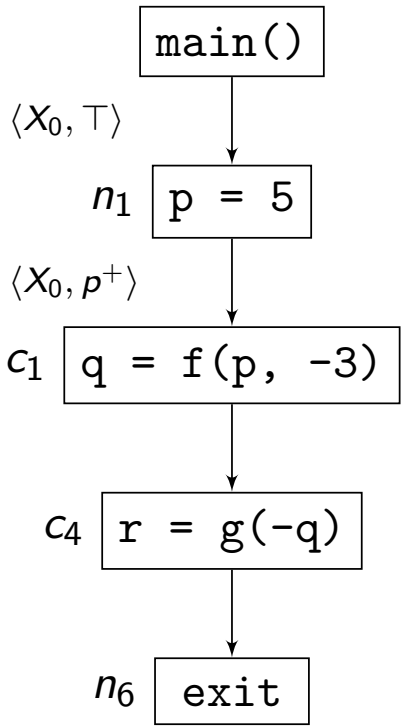


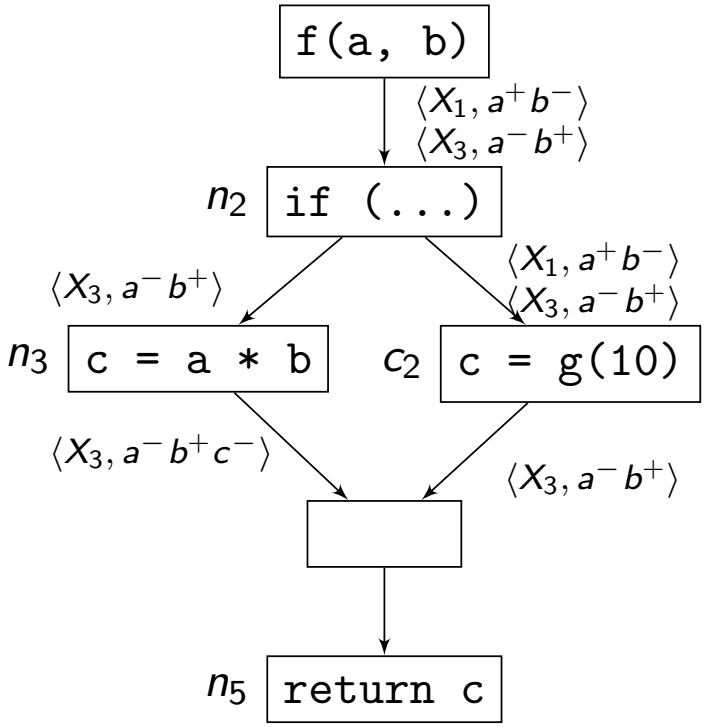
Context Transitions

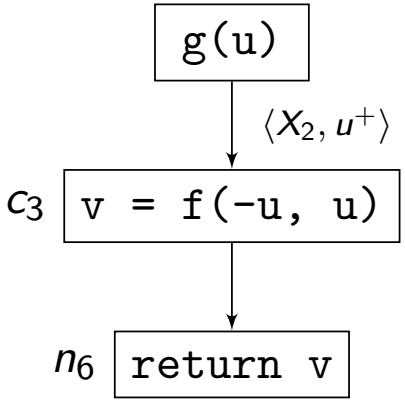


Component Lattice

Example - Sign Analysis

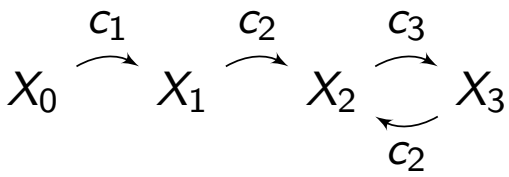




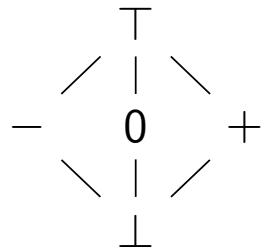


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$
X_2	g	u^+	$\top$
X_3	f	a^-b^+	$\top$

Value Contexts

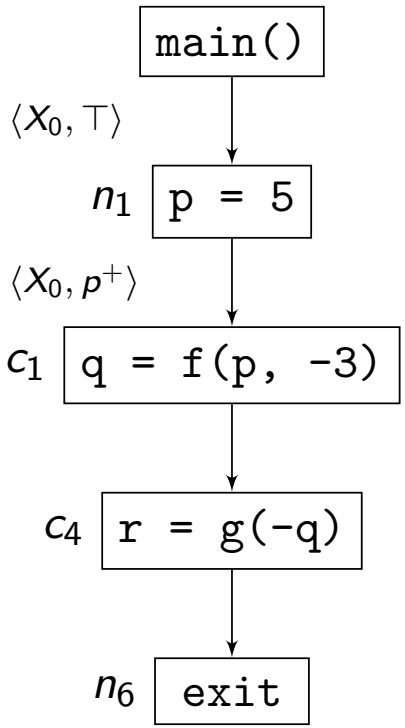


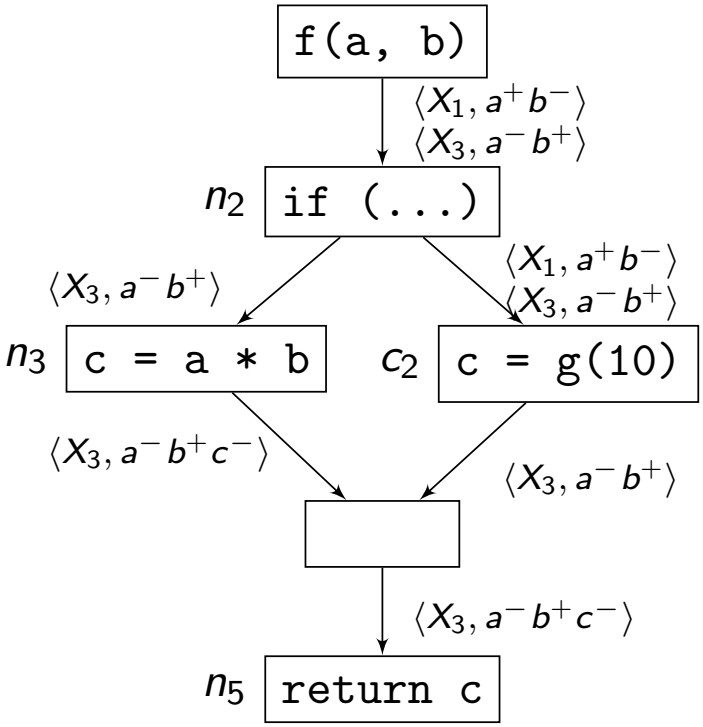
Context Transitions

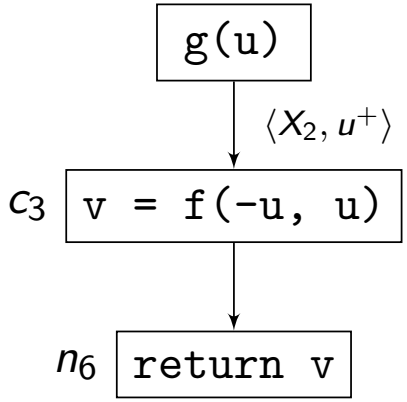


Component Lattice

Example - Sign Analysis

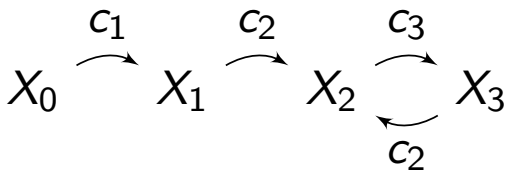




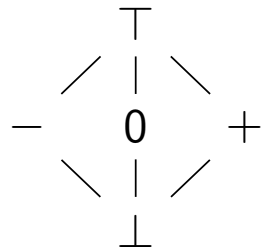


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$
X_2	g	u^+	$\top$
X_3	f	a^-b^+	$\top$

Value Contexts

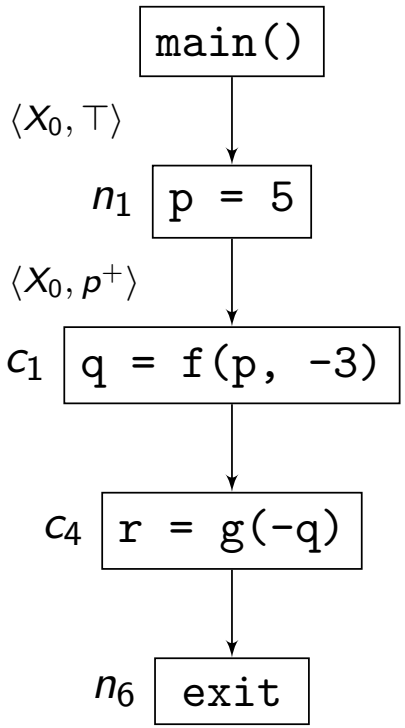


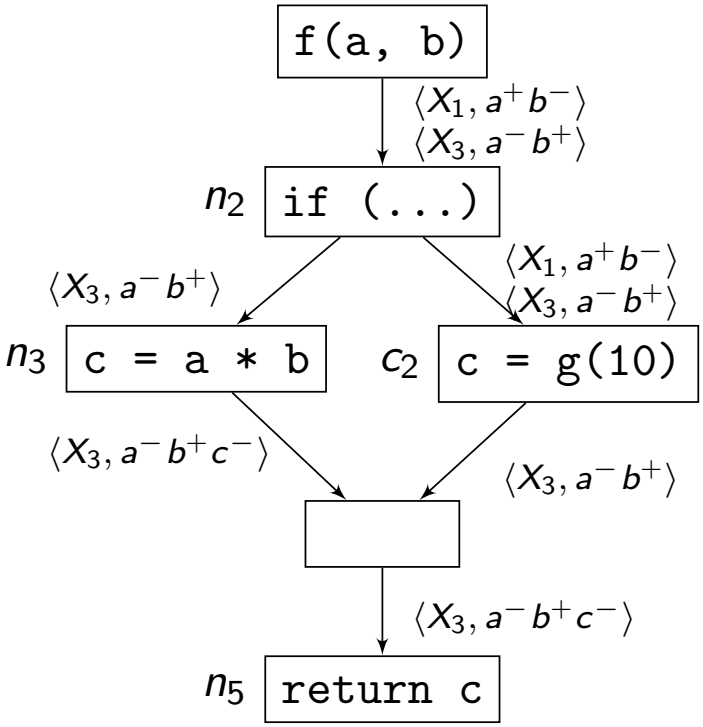
Context Transitions

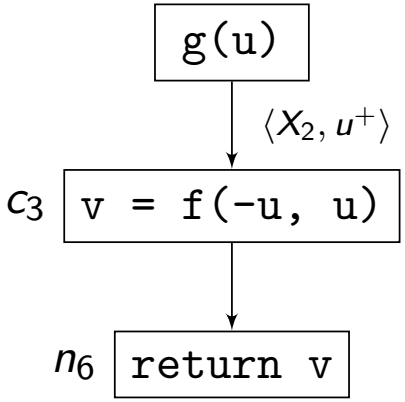


Component Lattice

Example - Sign Analysis

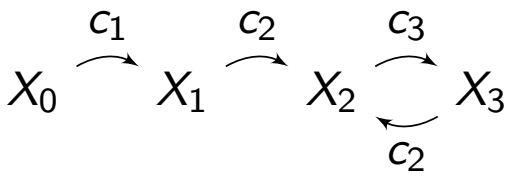




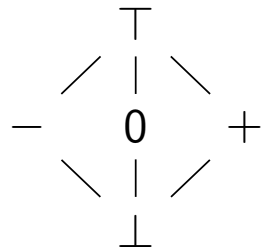


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$
X_2	g	u^+	$\top$
X_3	f	a^-b^+	$a^-b^+c^-$

Value Contexts

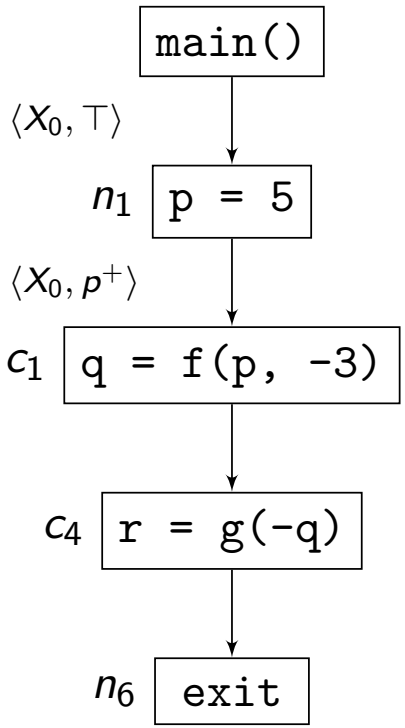


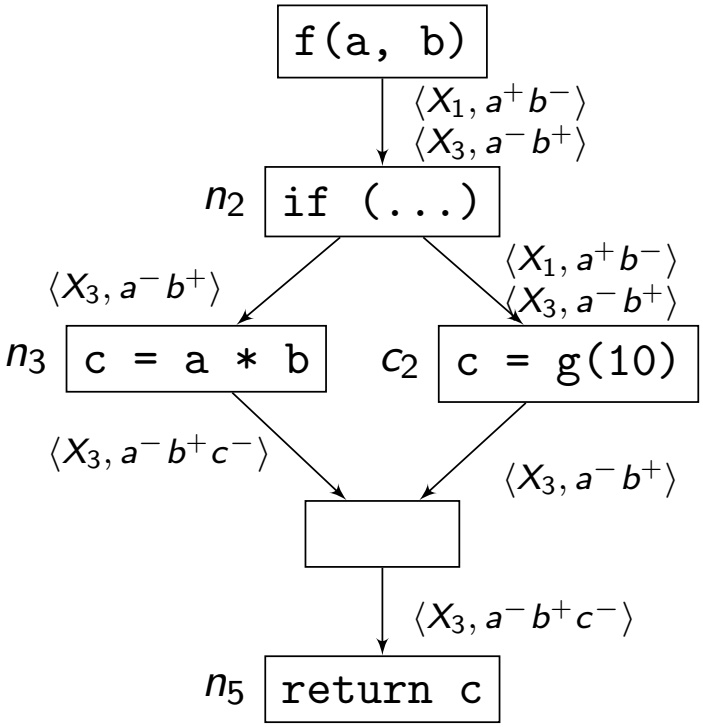
Context Transitions

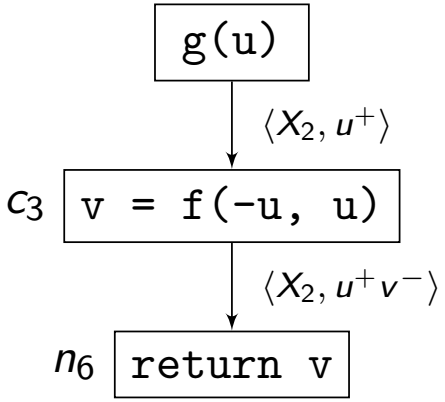


Component Lattice

Example - Sign Analysis

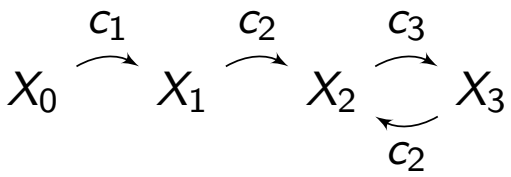




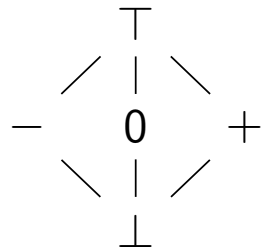


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$
X_2	g	u^+	$\top$
X_3	f	a^-b^+	$a^-b^+c^-$

Value Contexts

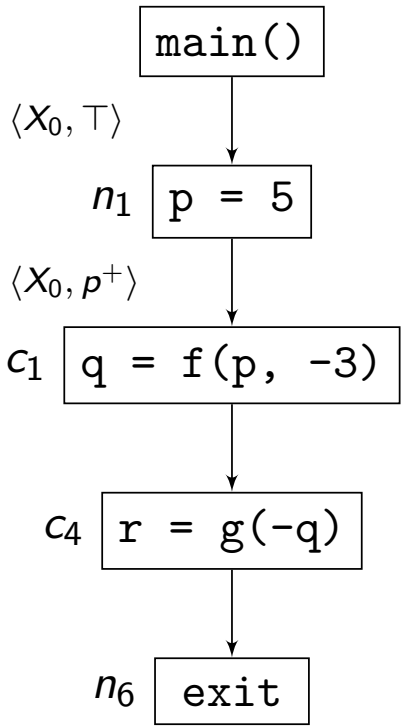


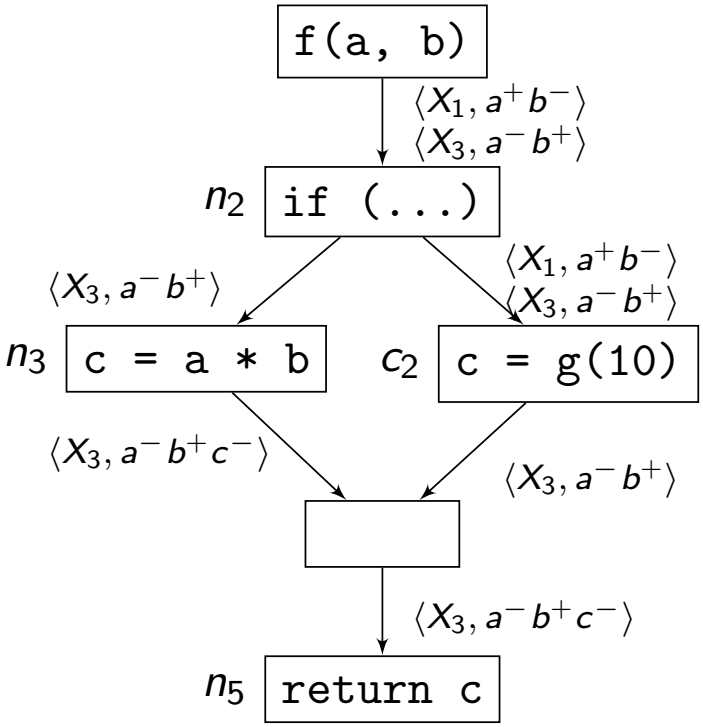
Context Transitions

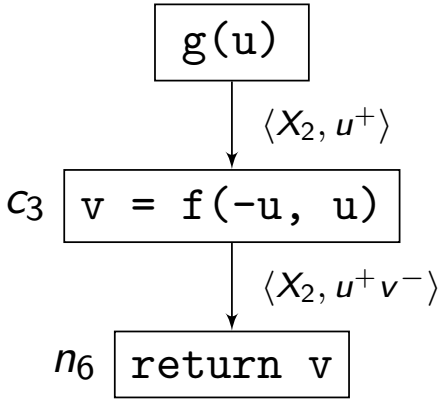


Component Lattice

Example - Sign Analysis

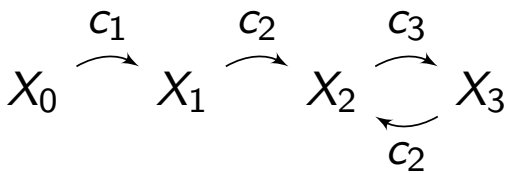




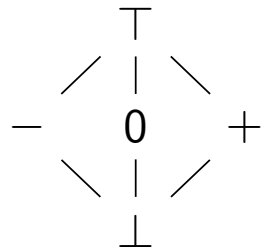


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$
X_2	g	u^+	u^+v^-
X_3	f	a^-b^+	$a^-b^+c^-$

Value Contexts

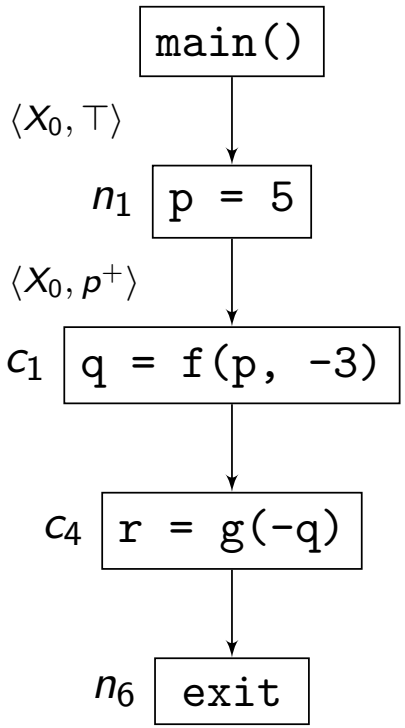


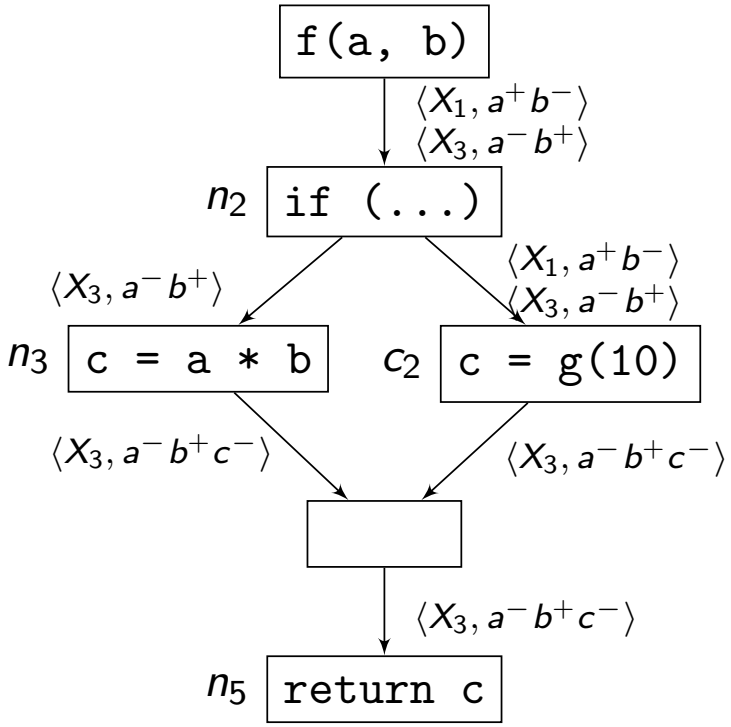
Context Transitions

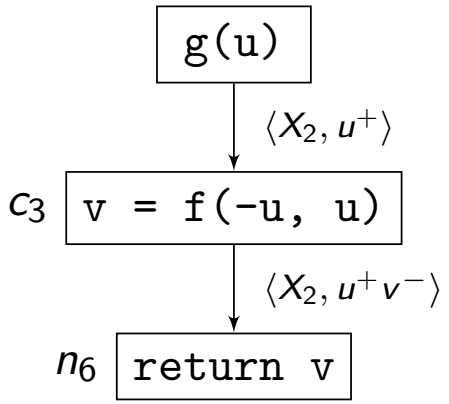


Component Lattice

Example - Sign Analysis

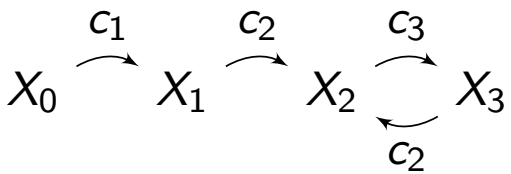




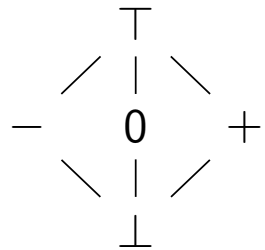


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$
X_2	g	u^+	u^+v^-
X_3	f	a^-b^+	$a^-b^+c^-$

Value Contexts

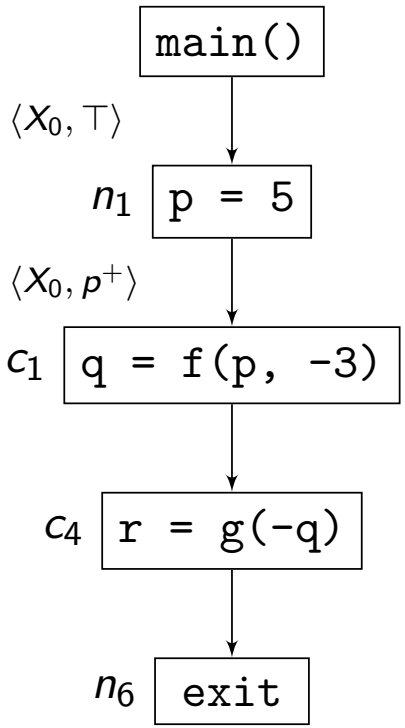


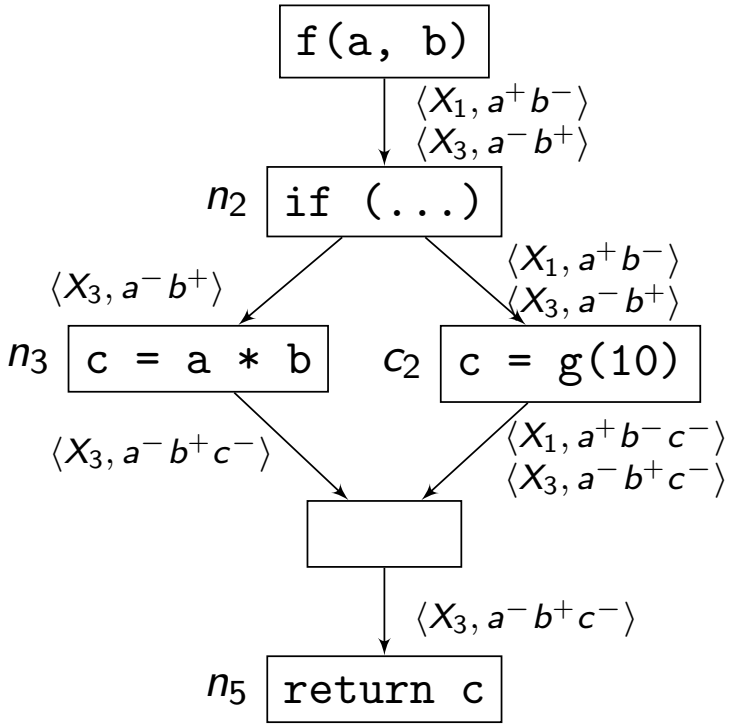
Context Transitions

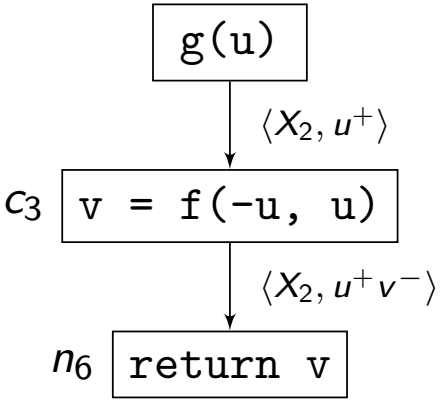


Component Lattice

Example - Sign Analysis

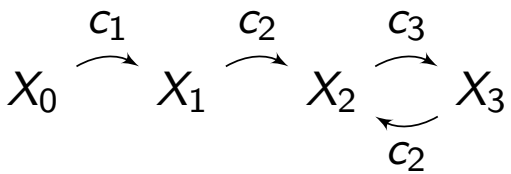




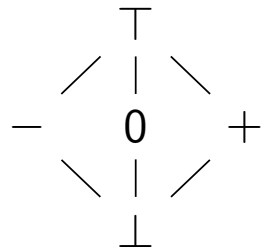


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$
X_2	g	u^+	u^+v^-
X_3	f	a^-b^+	$a^-b^+c^-$

Value Contexts

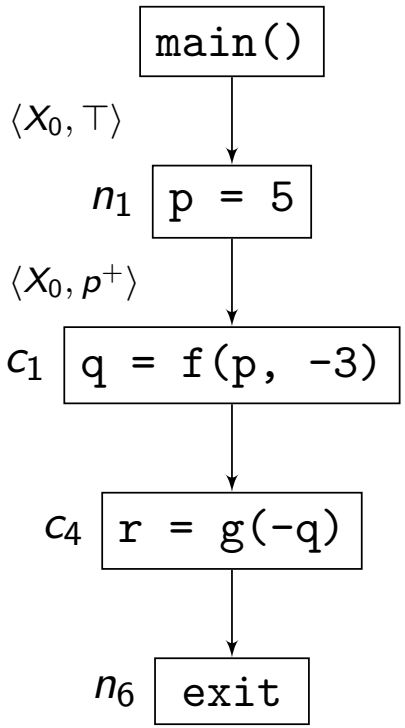


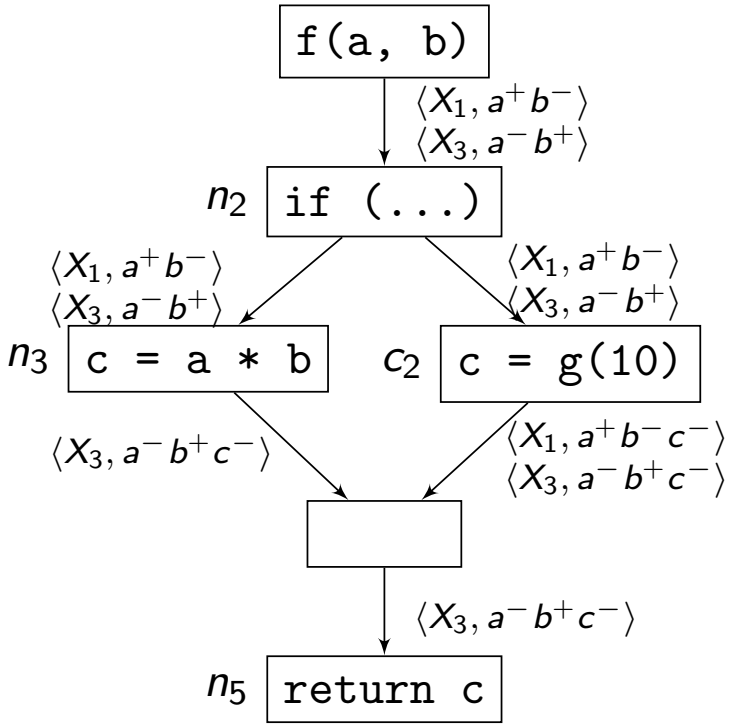
Context Transitions

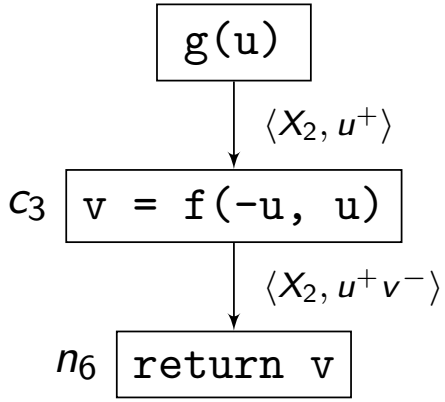


Component Lattice

Example - Sign Analysis

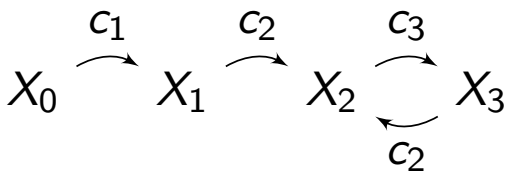




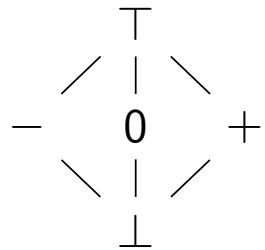


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$
X_2	g	u^+	u^+v^-
X_3	f	a^-b^+	$a^-b^+c^-$

Value Contexts

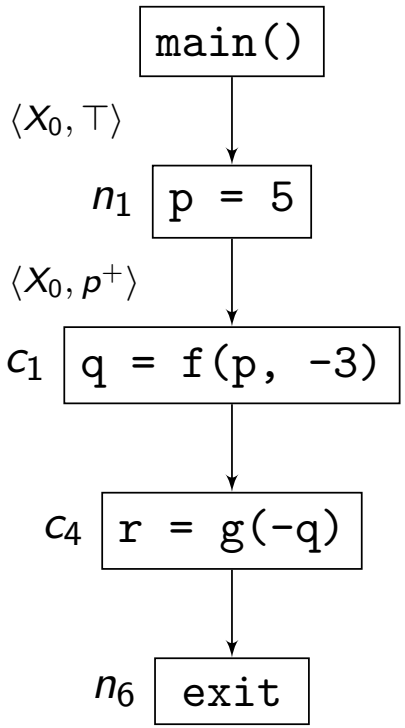


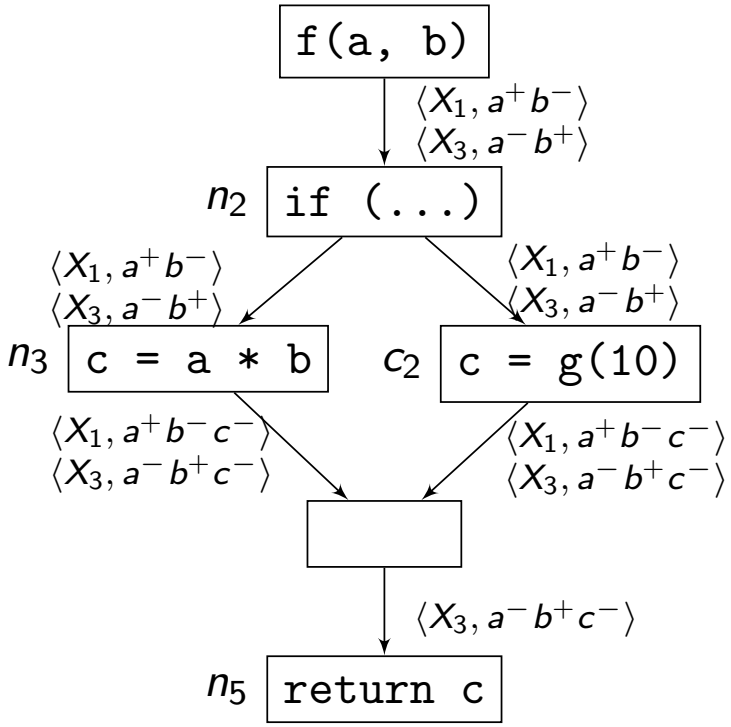
Context Transitions

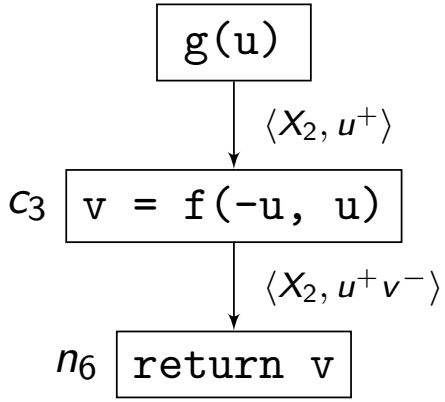


Component Lattice

Example - Sign Analysis

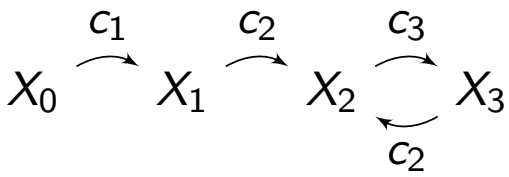




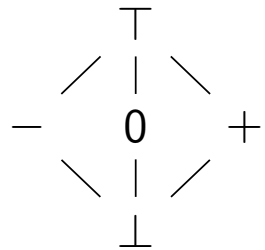


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$
X_2	g	u^+	u^+v^-
X_3	f	a^-b^+	$a^-b^+c^-$

Value Contexts

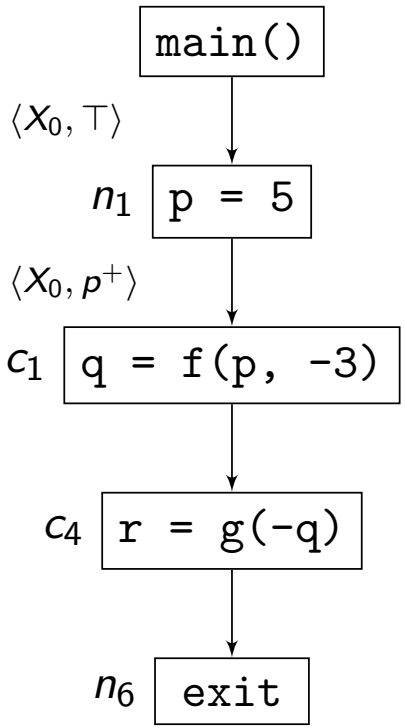


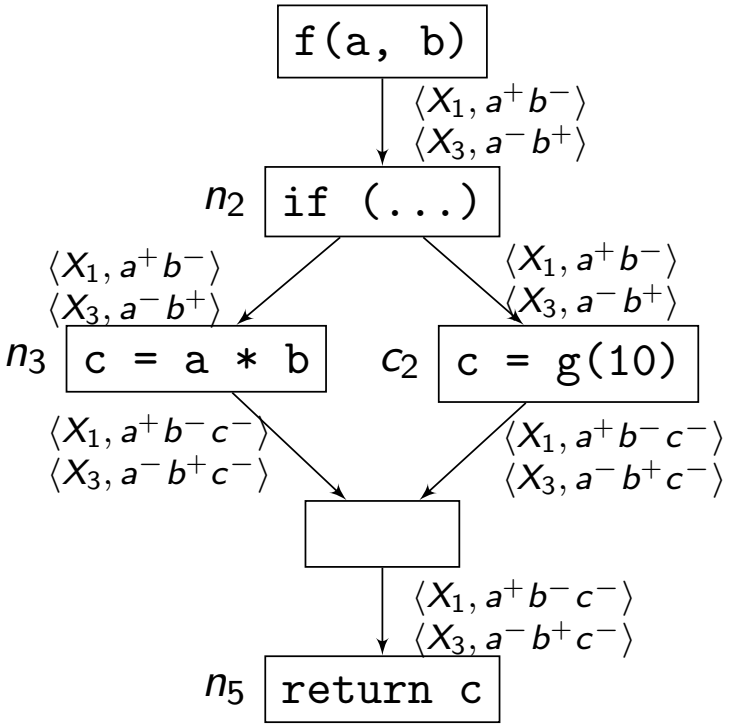
Context Transitions

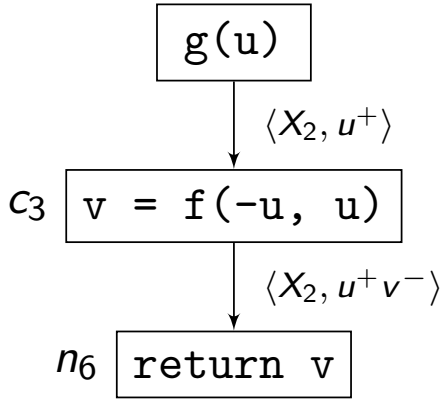


Component Lattice

Example - Sign Analysis

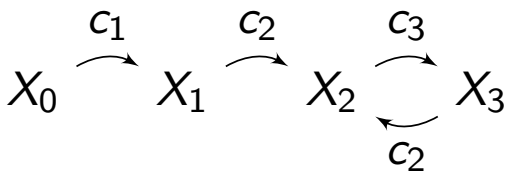




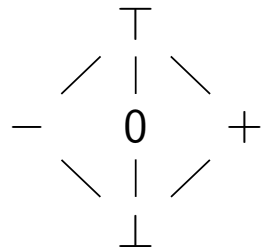


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$\top$
X_2	g	u^+	u^+v^-
X_3	f	a^-b^+	$a^-b^+c^-$

Value Contexts

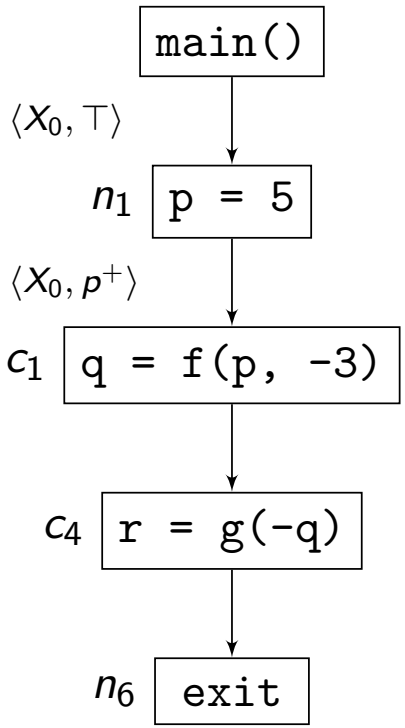


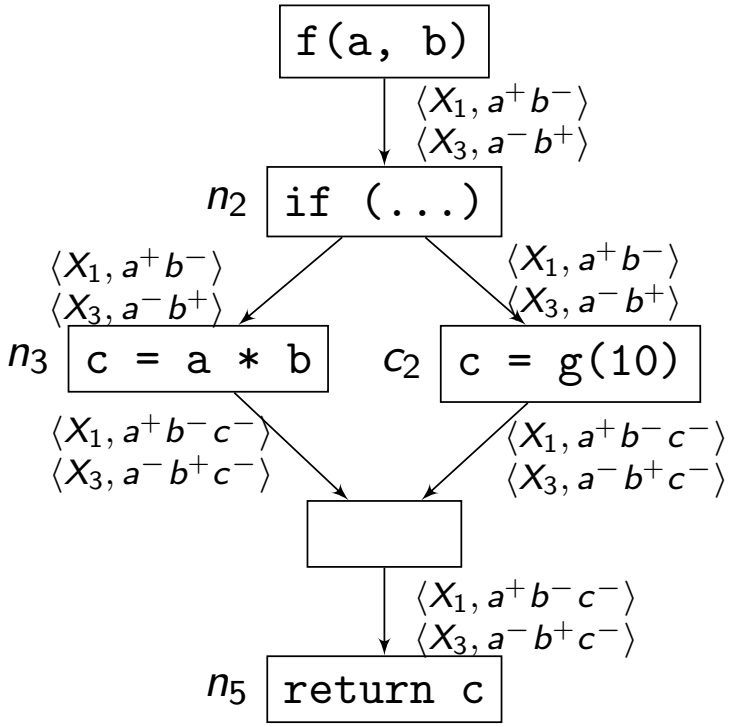
Context Transitions

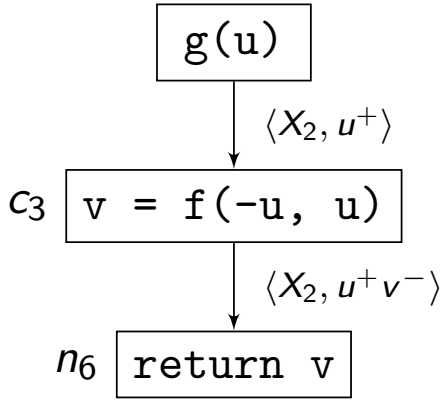


Component Lattice

Example - Sign Analysis

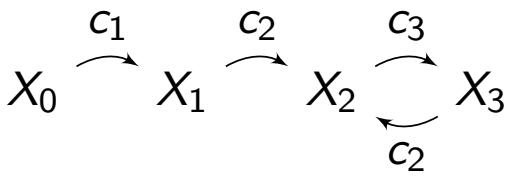




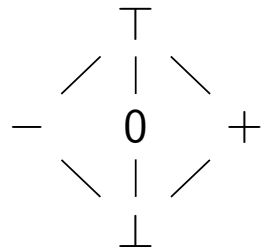


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$a^+b^-c^-$
X_2	g	u^+	u^+v^-
X_3	f	a^-b^+	$a^-b^+c^-$

Value Contexts

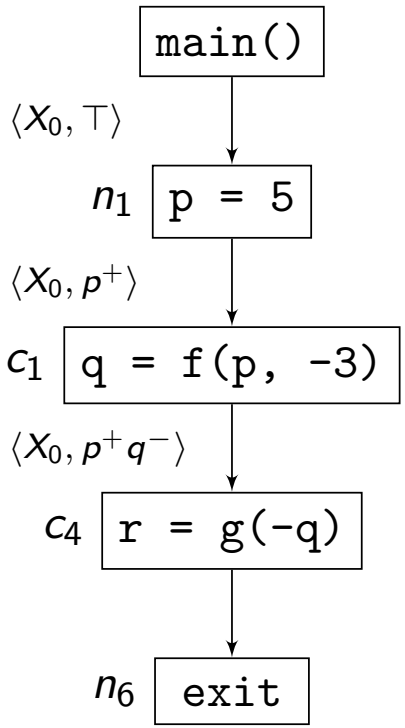


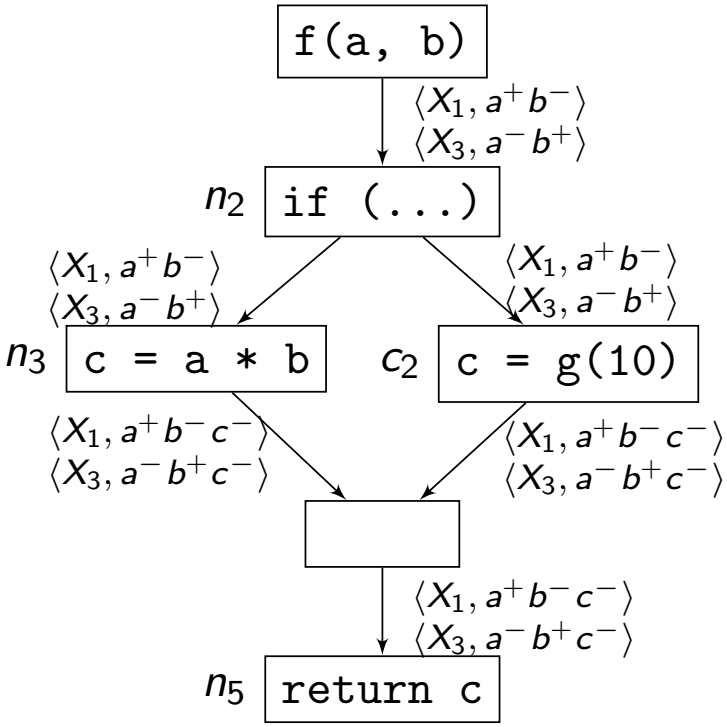
Context Transitions

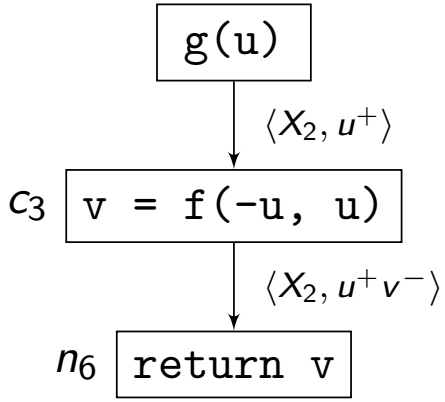


Component Lattice

Example - Sign Analysis

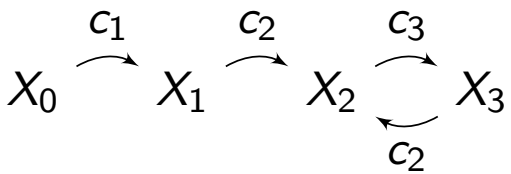




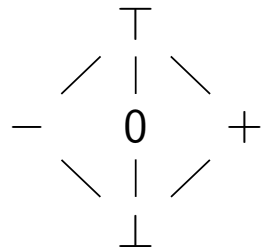


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$a^+b^-c^-$
X_2	g	u^+	u^+v^-
X_3	f	a^-b^+	$a^-b^+c^-$

Value Contexts

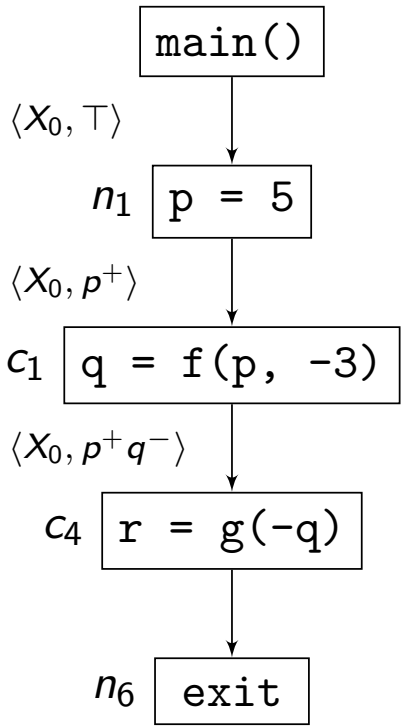


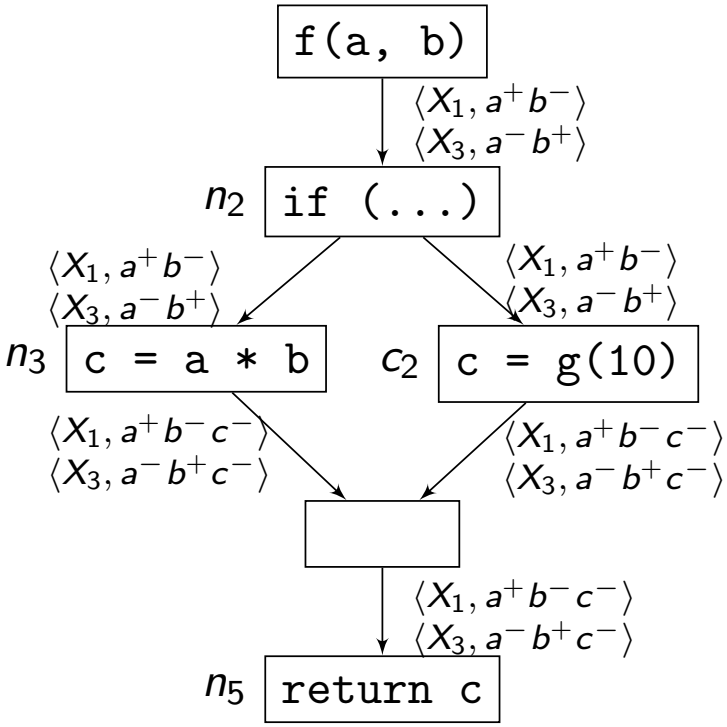
Context Transitions

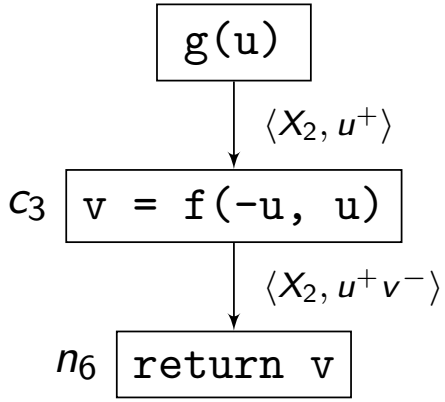


Component Lattice

Example - Sign Analysis

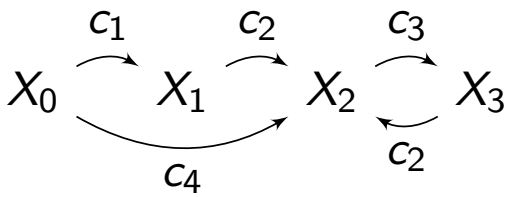




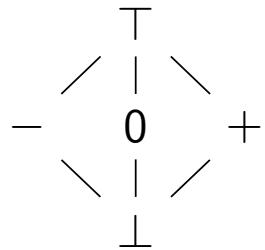


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$a^+b^-c^-$
X_2	g	u^+	u^+v^-
X_3	f	a^-b^+	$a^-b^+c^-$

Value Contexts

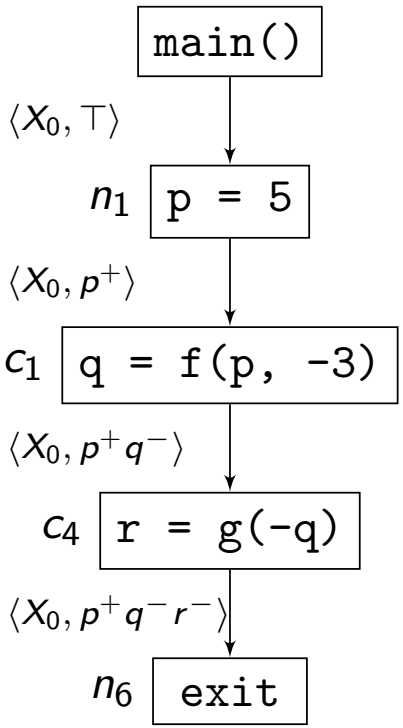


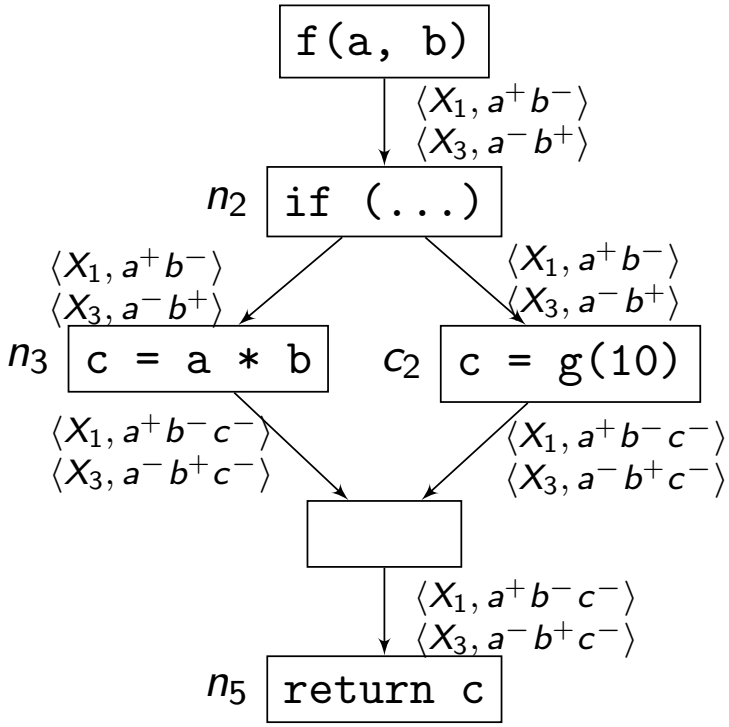
Context Transitions

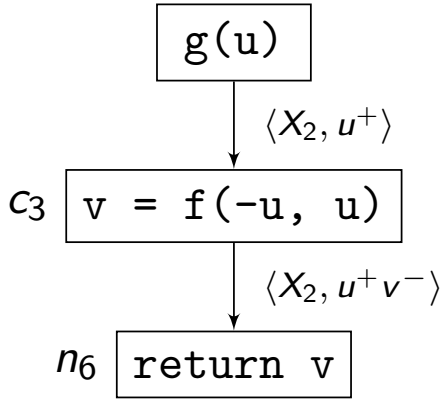


Component Lattice

Example - Sign Analysis

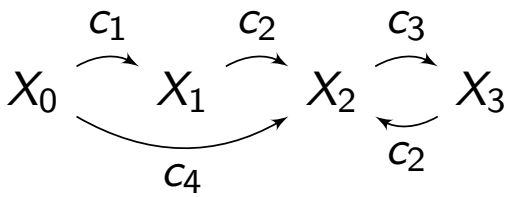




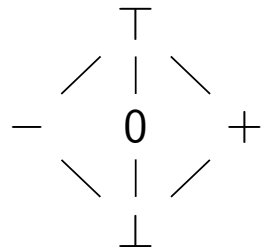


Context	Proc.	Entry	Exit
X_0	main	$\top$	$\top$
X_1	f	a^+b^-	$a^+b^-c^-$
X_2	g	u^+	u^+v^-
X_3	f	a^-b^+	$a^-b^+c^-$

Value Contexts

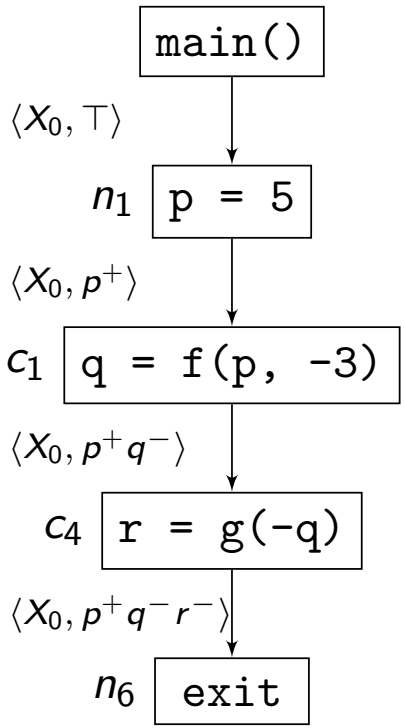


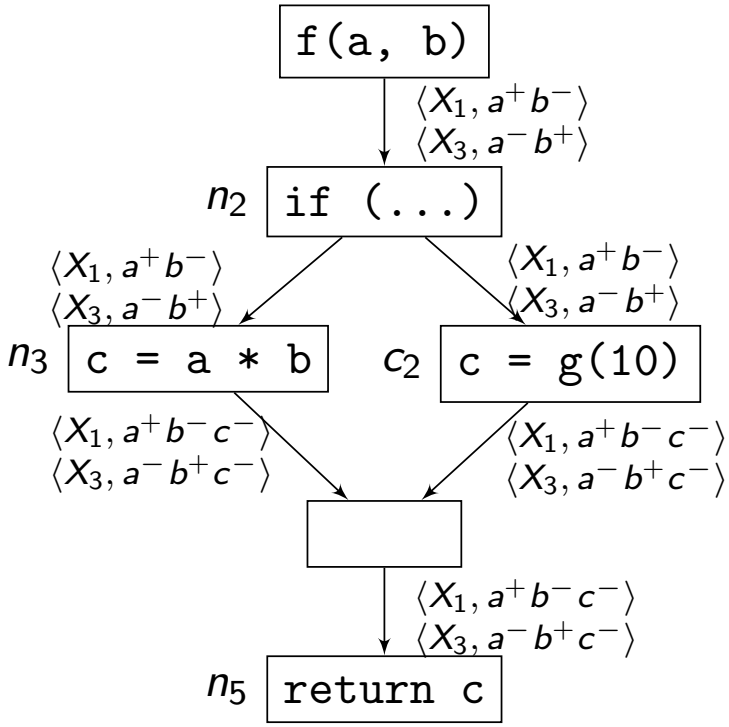
Context Transitions

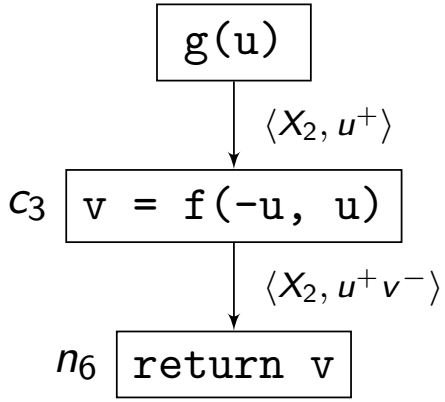


Component Lattice

Example - Sign Analysis

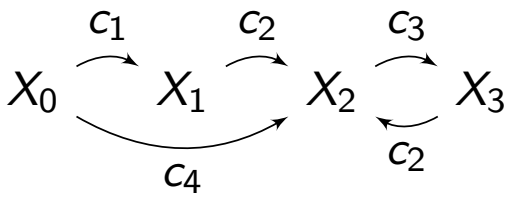




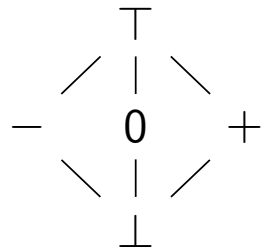


Context	Proc.	Entry	Exit
X_0	main	$\top$	$p^+q^-r^-$
X_1	f	a^+b^-	$a^+b^-c^-$
X_2	g	u^+	u^+v^-
X_3	f	a^-b^+	$a^-b^+c^-$

Value Contexts

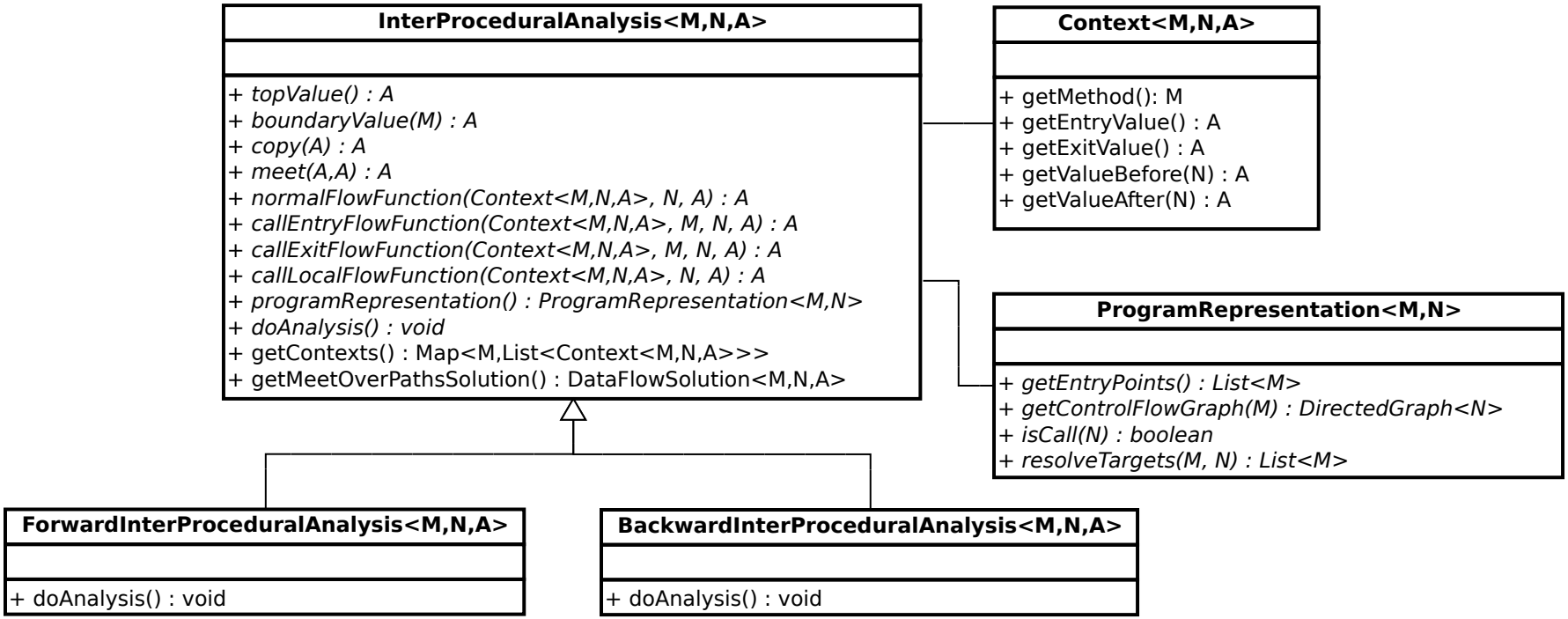


Context Transitions



Component Lattice

Implementation Framework



https://github.com/rohanpadhye/vasco

The Role of Call Graphs

- Context-sensitivity only useful if call graph is precise

The Role of Call Graphs

- Context-sensitivity only useful if call graph is precise
- OOP: Use points-to analysis to resolve virtual calls

The Role of Call Graphs

- Context-sensitivity only useful if call graph is precise
- OOP: Use points-to analysis to resolve virtual calls
- Imprecise points-to analysis $\Rightarrow$ "spurious" edges

The Role of Call Graphs

- Context-sensitivity only useful if call graph is precise
- OOP: Use points-to analysis to resolve virtual calls
- Imprecise points-to analysis $\Rightarrow$ "spurious" edges
- SPARK: Thousands of spurious edges even for small programs

The Role of Call Graphs

- Context-sensitivity only useful if call graph is precise
- OOP: Use points-to analysis to resolve virtual calls
- Imprecise points-to analysis $\Rightarrow$ "spurious" edges
- SPARK: Thousands of spurious edges even for small programs
  - e.g. Over 250 targets for `x.hashCode()` in `HashSet`

The Role of Call Graphs

- Context-sensitivity only useful if call graph is precise
- OOP: Use points-to analysis to resolve virtual calls
- Imprecise points-to analysis $\Rightarrow$ "spurious" edges
- SPARK: Thousands of spurious edges even for small programs
  - e.g. Over 250 targets for `x.hashCode()` in `HashSet`
- Affects efficiency and precision of interprocedural analysis

The Role of Call Graphs

- Context-sensitivity only useful if call graph is precise
- OOP: Use points-to analysis to resolve virtual calls
- Imprecise points-to analysis $\Rightarrow$ "spurious" edges
- SPARK: Thousands of spurious edges even for small programs
  - e.g. Over 250 targets for `x.hashCode()` in `HashSet`
- Affects efficiency and precision of interprocedural analysis
- Points-to Analysis using Value Contexts

The Role of Call Graphs

- Context-sensitivity only useful if call graph is precise
- OOP: Use points-to analysis to resolve virtual calls
- Imprecise points-to analysis $\Rightarrow$ "spurious" edges
- SPARK: Thousands of spurious edges even for small programs
  - e.g. Over 250 targets for `x.hashCode()` in `HashSet`
- Affects efficiency and precision of interprocedural analysis
- Points-to Analysis using Value Contexts
  - Flow and context-sensitive points-to analysis (FCPA)

The Role of Call Graphs

- Context-sensitivity only useful if call graph is precise
- OOP: Use points-to analysis to resolve virtual calls
- Imprecise points-to analysis $\Rightarrow$ "spurious" edges
- SPARK: Thousands of spurious edges even for small programs
  - e.g. Over 250 targets for `x.hashCode()` in `HashSet`
- Affects efficiency and precision of interprocedural analysis
- Points-to Analysis using Value Contexts
  - Flow and context-sensitive points-to analysis (FCPA)
  - Context-sensitive call graph constructed on-the-fly

Results of Points-To Analysis

- Tested on 7 benchmarks from SPEC JVM98 and DaCapo 2006

Results of Points-To Analysis

- Tested on 7 benchmarks from SPEC JVM98 and DaCapo 2006
- Time to analyze: 1.15 sec (compress) to 697.4 sec (antlr)

Results of Points-To Analysis

- Tested on 7 benchmarks from SPEC JVM98 and DaCapo 2006
- Time to analyze: 1.15 sec (compress) to 697.4 sec (antlr)
- Average contexts per method: 4.24 (mpegaudio) to 25.04 (jess)

Results of Points-To Analysis

- Tested on 7 benchmarks from SPEC JVM98 and DaCapo 2006
- Time to analyze: 1.15 sec (compress) to 697.4 sec (antlr)
- Average contexts per method: 4.24 (mpegaudio) to 25.04 (jess)
- Number of interprocedural paths in resulting call graph (for $k = 10$):

Results of Points-To Analysis

- Tested on 7 benchmarks from SPEC JVM98 and DaCapo 2006
- Time to analyze: 1.15 sec (compress) to 697.4 sec (antlr)
- Average contexts per method: 4.24 (mpegaudio) to 25.04 (jess)
- Number of interprocedural paths in resulting call graph (for $k = 10$):
  - Over 96% less paths in FCPA over SPARK for 3 benchmarks

Results of Points-To Analysis

- Tested on 7 benchmarks from SPEC JVM98 and DaCapo 2006
- Time to analyze: 1.15 sec (compress) to 697.4 sec (antlr)
- Average contexts per method: 4.24 (mpegaudio) to 25.04 (jess)
- Number of interprocedural paths in resulting call graph (for $k = 10$):
  - Over 96% less paths in FCPA over SPARK for 3 benchmarks
  - 62-92% less paths in FCPA over SPARK for remaining benchmarks

Outline

Access Graphs for Garbage Collection

How to use access graphs for improving garbage collection?

Access Graphs for Garbage Collection

How to use access graphs for improving garbage collection?

1. Assign `null` to dead access paths.

Access Graphs for Garbage Collection

How to use access graphs for improving garbage collection?

1. Assign `null` to dead access paths.
   - Requires availability and anticipability analysis to prevent exceptions.

Access Graphs for Garbage Collection

How to use access graphs for improving garbage collection?

1. Assign `null` to dead access paths.
   - Requires availability and anticipability analysis to prevent exceptions.
   - Cannot `null`ify access paths that are not provably safe to dereference.

Access Graphs for Garbage Collection

How to use access graphs for improving garbage collection?

1. Assign `null` to dead access paths.
   - Requires availability and anticipability analysis to prevent exceptions.
   - Cannot `null`ify access paths that are not provably safe to dereference.
   - The safety analyses themselves depend on alias information.

Access Graphs for Garbage Collection

How to use access graphs for improving garbage collection?

1. Assign `null` to dead access paths.
   - Requires availability and anticipability analysis to prevent exceptions.
   - Cannot `null`ify access paths that are not provably safe to dereference.
   - The safety analyses themselves depend on alias information.
   - Increase in code size and possible performance penalty.

Access Graphs for Garbage Collection

How to use access graphs for improving garbage collection?

1. Assign `null` to dead access paths.
   - Requires availability and anticipability analysis to prevent exceptions.
   - Cannot `null`ify access paths that are not provably safe to dereference.
   - The safety analyses themselves depend on alias information.
   - Increase in code size and possible performance penalty.
   - Redundant `null`ification of same reference from aliased access paths.

Access Graphs for Garbage Collection

How to use access graphs for improving garbage collection?

1. Assign `null` to dead access paths.
   - Requires availability and anticipability analysis to prevent exceptions.
   - Cannot `null`ify access paths that are not provably safe to dereference.
   - The safety analyses themselves depend on alias information.
   - Increase in code size and possible performance penalty.
   - Redundant `null`ification of same reference from aliased access paths.
2. Augment garbage collector to traverse access graphs.

Access Graphs for Garbage Collection

How to use access graphs for improving garbage collection?

1. Assign `null` to dead access paths.
   - Requires availability and anticipability analysis to prevent exceptions.
   - Cannot `null`ify access paths that are not provably safe to dereference.
   - The safety analyses themselves depend on alias information.
   - Increase in code size and possible performance penalty.
   - Redundant `null`ification of same reference from aliased access paths.
2. Augment garbage collector to traverse access graphs.
   - No need of safety analysis.

Access Graphs for Garbage Collection

How to use access graphs for improving garbage collection?

1. Assign `null` to dead access paths.
   - Requires availability and anticipability analysis to prevent exceptions.
   - Cannot `null`ify access paths that are not provably safe to dereference.
   - The safety analyses themselves depend on alias information.
   - Increase in code size and possible performance penalty.
   - Redundant `null`ification of same reference from aliased access paths.
2. Augment garbage collector to traverse access graphs.
   - No need of safety analysis.
   - Perfect alias information available at run-time.

Access Graphs for Garbage Collection

How to use access graphs for improving garbage collection?

1. Assign `null` to dead access paths.
   - Requires availability and anticipability analysis to prevent exceptions.
   - Cannot `null`ify access paths that are not provably safe to dereference.
   - The safety analyses themselves depend on alias information.
   - Increase in code size and possible performance penalty.
   - Redundant `null`ification of same reference from aliased access paths.
2. Augment garbage collector to traverse access graphs.
   - No need of safety analysis.
   - Perfect alias information available at run-time.
   - Difficult to map named variables and fields to run-time offsets.

Access Graphs for Garbage Collection

How to use access graphs for improving garbage collection?

1. Assign `null` to dead access paths.
   - Requires availability and anticipability analysis to prevent exceptions.
   - Cannot `null`ify access paths that are not provably safe to dereference.
   - The safety analyses themselves depend on alias information.
   - Increase in code size and possible performance penalty.
   - Redundant `null`ification of same reference from aliased access paths.
2. Augment garbage collector to traverse access graphs.
   - No need of safety analysis.
   - Perfect alias information available at run-time.
   - Difficult to map named variables and fields to run-time offsets.
   - Optimizations after HRA (static or JIT) invalidate access graphs.

Access Graphs for Garbage Collection

How to use access graphs for improving garbage collection?

1. Assign `null` to dead access paths.
   - Requires availability and anticipability analysis to prevent exceptions.
   - Cannot `null`ify access paths that are not provably safe to dereference.
   - The safety analyses themselves depend on alias information.
   - Increase in code size and possible performance penalty.
   - Redundant `null`ification of same reference from aliased access paths.
2. Augment garbage collector to traverse access graphs.
   - No need of safety analysis.
   - Perfect alias information available at run-time.
   - Difficult to map named variables and fields to run-time offsets.
   - Optimizations after HRA (static or JIT) invalidate access graphs.
3. Dynamic heap pruning - a hybrid approach.

Dynamic Heap Pruning

Manipulate the heap using a debugger!

Dynamic Heap Pruning

Manipulate the heap using a debugger!

1. Pause a running program when pruning has to be performed.

Dynamic Heap Pruning

Manipulate the heap using a debugger!

1. Pause a running program when pruning has to be performed.
2. For each frame on the call stack do:

Dynamic Heap Pruning

Manipulate the heap using a debugger!

1. Pause a running program when pruning has to be performed.
2. For each frame on the call stack do:
    1. Find the paused program point P using return address of next frame (or PC for top-of-stack).

Dynamic Heap Pruning

Manipulate the heap using a debugger!

1. Pause a running program when pruning has to be performed.
2. For each frame on the call stack do:
   1. Find the paused program point P using return address of next frame (or PC for top-of-stack).
   2. Construct the call string σ using the sequence of return addresses from the bottom-of-stack.

Dynamic Heap Pruning

Manipulate the heap using a debugger!

1. Pause a running program when pruning has to be performed.
2. For each frame on the call stack do:
   1. Find the paused program point P using return address of next frame (or PC for top-of-stack).
   2. Construct the call string σ using the sequence of return addresses from the bottom-of-stack.
   3. Determine the value context X by traversing σ in the context transition graph.

Dynamic Heap Pruning

Manipulate the heap using a debugger!

1. Pause a running program when pruning has to be performed.
2. For each frame on the call stack do:
   1. Find the paused program point P using return address of next frame (or PC for top-of-stack).
   2. Construct the call string σ using the sequence of return addresses from the bottom-of-stack.
   3. Determine the value context X by traversing σ in the context transition graph.
   4. Retrieve the access graphs for point P in context X.

Dynamic Heap Pruning

Manipulate the heap using a debugger!

1. Pause a running program when pruning has to be performed.
2. For each frame on the call stack do:
   1. Find the paused program point P using return address of next frame (or PC for top-of-stack).
   2. Construct the call string σ using the sequence of return addresses from the bottom-of-stack.
   3. Determine the value context X by traversing σ in the context transition graph.
   4. Retrieve the access graphs for point P in context X.
   5. Traverse the access graphs from the root variables (stack locals) and label heap objects with the set of accessor nodes that reach them.

Dynamic Heap Pruning

Manipulate the heap using a debugger!

1. Pause a running program when pruning has to be performed.
2. For each frame on the call stack do:
    1. Find the paused program point P using return address of next frame (or PC for top-of-stack).
    2. Construct the call string σ using the sequence of return addresses from the bottom-of-stack.
    3. Determine the value context X by traversing σ in the context transition graph.
    4. Retrieve the access graphs for point P in context X.
    5. Traverse the access graphs from the root variables (stack locals) and label heap objects with the set of accessor nodes that reach them.
3. For each labelled object in the heap do:

Dynamic Heap Pruning

Manipulate the heap using a debugger!

1. Pause a running program when pruning has to be performed.
2. For each frame on the call stack do:
   1. Find the paused program point P using return address of next frame (or PC for top-of-stack).
   2. Construct the call string σ using the sequence of return addresses from the bottom-of-stack.
   3. Determine the value context X by traversing σ in the context transition graph.
   4. Retrieve the access graphs for point P in context X.
   5. Traverse the access graphs from the root variables (stack locals) and label heap objects with the set of accessor nodes that reach them.
3. For each labelled object in the heap do:
   1. Find the set of live fields by looking at the edges out of every accessor that reaches it.

Dynamic Heap Pruning

Manipulate the heap using a debugger!

1. Pause a running program when pruning has to be performed.
2. For each frame on the call stack do:
   1. Find the paused program point P using return address of next frame (or PC for top-of-stack).
   2. Construct the call string σ using the sequence of return addresses from the bottom-of-stack.
   3. Determine the value context X by traversing σ in the context transition graph.
   4. Retrieve the access graphs for point P in context X.
   5. Traverse the access graphs from the root variables (stack locals) and label heap objects with the set of accessor nodes that reach them.
3. For each labelled object in the heap do:
   1. Find the set of live fields by looking at the edges out of every accessor that reaches it.
   2. Set the value of all other fields (which are dead) to `null`.

Dynamic Heap Pruning

Manipulate the heap using a debugger!

1. Pause a running program when pruning has to be performed.
2. For each frame on the call stack do:
   1. Find the paused program point P using return address of next frame (or PC for top-of-stack).
   2. Construct the call string σ using the sequence of return addresses from the bottom-of-stack.
   3. Determine the value context X by traversing σ in the context transition graph.
   4. Retrieve the access graphs for point P in context X.
   5. Traverse the access graphs from the root variables (stack locals) and label heap objects with the set of accessor nodes that reach them.
3. For each labelled object in the heap do:
   1. Find the set of live fields by looking at the edges out of every accessor that reaches it.
   2. Set the value of all other fields (which are dead) to `null`.
4. Resume the program. Let garbage collection run as normal.

Outline

Summary

The following were the main contributions of this project:

1. A liveness-driven heap abstraction for precise alias analysis.
2. A generic access graph library implemented in Java.
3. A generic inter-procedural data flow analysis framework implemented in Java.
4. A flow- and context-sensitive points-to analysis implemented in Soot that constructs precise call graphs.
5. A technique for performing dynamic heap pruning implemented using the Java Debug Interface (JDI).

Future Work

1. Implementation of an inter-procedural liveness-driven heap points-to analysis.
2. Performance analysis of dynamic heap pruning on real benchmarks.
3. Shape analysis using accessor relationship graphs.