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
1. Background and Motivation
   - Heap Reference Analysis
   - Key Issues

2. Heap Alias Analysis
   - Need for Alias Analysis
   - Existing Abstractions
   - Proposed Abstraction: Accessor Relationship Graph

3. Interprocedural Analysis
   - Existing Frameworks
   - Our Framework: Value Contexts
   - The Role of Call Graphs

4. Access Graphs for Garbage Collection
   - Existing Ideas
   - Novel Technique: Dynamic Heap Pruning

5. Summary & Future Work
Background

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

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

Access graph for \( x \) at \( S_2 \).

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

1 Background and Motivation
   - Heap Reference Analysis
   - Key Issues

2 Heap Alias Analysis
   - Need for Alias Analysis
   - Existing Abstractions
   - Proposed Abstraction: Accessor Relationship Graph

3 Interprocedural Analysis
   - Existing Frameworks
   - Our Framework: Value Contexts
   - The Role of Call Graphs

4 Access Graphs for Garbage Collection
   - Existing Ideas
   - Novel Technique: Dynamic Heap Pruning

5 Summary & Future Work
Need for Alias Analysis

- $x.p = z$
- $u = y.p$
- $v = u.q$
- $\text{use } v$

- $x$ and $y$ do not alias at $S_4$.  
- $LVIN_4$
  - $x$
  - $y \rightarrow p_5 \rightarrow q_6$
- $LVOUT_4$
  - $y \rightarrow p_5 \rightarrow q_6$
Need for Alias Analysis

- $x = y$
- $x \cdot p = z$
- $u = y \cdot p$
- $v = u \cdot q$
- use $v$

- $x$ may alias $y$ at $S_4$
- $LVIN_4$
  - $x$
  - $y \rightarrow p_5 \rightarrow q_6$
  - $z \rightarrow q_6$
- $LVOUT_4$
  - $y \rightarrow p_5 \rightarrow q_6$
Need for Alias Analysis

- \( x = y \)  
- \( x \cdot p = z \)  
- \( u = y \cdot p \)  
- \( v = u \cdot q \)  
- \( \text{use } v \)  

- \( x \) must alias \( y \) at \( S_4 \)
- \( \text{LVIN}_4 \)
  - \( x \)  
  - \( y \)  
  - \( z \rightarrow q_6 \)  
- \( \text{LVOUT}_4 \)
  - \( y \rightarrow p_5 \rightarrow q_6 \)
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 preceding examples:

\[ w.r = z \]

In the above program, \( z \) is live if \( w \) may be aliased to any object accessible by the pattern \( x(.p)*.q \).

- The key observation is here is that we need to determine aliases between **live access patterns**.
Access Graphs and Access Patterns

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

Consider liveness at \( S_2 \).

Access Graph

\[
\begin{array}{c}
\text{Access Patterns} \\
\quad \text{S1: } w.r = z \Rightarrow x.p.(p) \\
\quad \text{S2: while (...):} \\
\quad \text{S3: } x = x.p \\
\quad \text{S4: } x = x.q \\
\quad \text{S5: } x = x.r \\
\quad \text{S6: use } x \\
\quad \text{S7: EXIT} \\
\end{array}
\]
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
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

\[ S_0 \quad y = \text{new} \]
\[ S_1 \quad z = \text{new} \]
\[ S_2 \quad t = \text{new} \]
\[ S_3 \quad t.n = x \]
\[ S_4 \quad x = t \]
\[ S_5 \quad a = x \]
\[ S_6 \quad b = a.n \]
\[ S_7 \quad a.n = y \]
\[ S_8 \quad b.n = z \]
\[ S_9 \quad \text{use } x.n \]
\[ S_{10} \quad \text{exit} \]
Proposed Approach

\[ \begin{align*}
S_0 & : y = \text{new} \\
S_1 & : z = \text{new} \\
S_2 & : t = \text{new} \\
S_3 & : t.n = x \\
S_4 & : x = t \\
S_5 & : a = x \\
S_6 & : b = a.n \\
S_7 & : a.n = y \\
S_8 & : b.n = z \\
S_9 & : \text{use } x.n \\
S_{10} & : \text{exit}
\end{align*} \]

Actual heap layout after \( S_6 \):

\[ \begin{array}{c}
\text{Actual heap layout after } S_6 \\
\begin{tikzpicture}
  \node (a) at (0,0) {$a$};
  \node (b) at (1,0) {$b$};
  \node (n) at (0,-1) {$n$};
  \node (n2) at (1,-1) {$n$};
  \node (n3) at (2,-1) {$n$};
  \node (n4) at (3,-1) {$n$};
  \node (n5) at (4,-1) {$n$};
  \node (n6) at (5,-1) {$\ldots$};
  \node (x) at (0,-2) {$x$};
  \draw (a) -- (n);
  \draw (n) -- (n2);
  \draw (n2) -- (n3);
  \draw (n3) -- (n4);
  \draw (n4) -- (n5);
  \draw (n5) -- (n6);
  \draw (x) -- (n);
  \node (y) at (1,-2) {$y$};
  \node (z) at (2,-2) {$z$};
  \draw (y) -- (n);
  \draw (z) -- (n2);
\end{tikzpicture}
\end{array} \]
Proposed Approach

Actual heap layout after $S_6$

$P_0$: Initial Points-to Analysis

$PTOUT_6 = PTIN_7 = PTIN_8$:
Proposed Approach

$S_0 \ y = \text{new}$

$S_1 \ z = \text{new}$

$S_2 \ t = \text{new}$

$S_3 \ t.n = x$

$S_4 \ x = t$

$S_5 \ a = x$

$S_6 \ b = a.n$

$S_7 \ a.n = y$

$S_8 \ b.n = z$

$S_9 \ \text{use } x.n$

$S_{10} \ \text{exit}$

Actual heap layout after $S_6$

$L_1$: Liveness after $P_0$

$PVIN_9 = PVOUT_7 = PVOUT_8$:

- $x \rightarrow n_9$
- $L_7$ (considering $a \equiv x$):
  - $a$, $x \rightarrow n_9$, $y$
- $L_8$ (considering $b \equiv x$):
  - $b$, $x \rightarrow n_9$, $z$
- $PVOUT_6 = PVIN_7 \cup PVIN_8$:
  - $a$, $b$, $x \rightarrow n_9$, $y$, $z$

$P_0$: Initial Points-to Analysis

$PTOUT_6 = PTIN_7 = PTIN_8$:

- $x \rightarrow S_2$
- $y \rightarrow S_0$
- $z \rightarrow S_1$
Proposed Approach

Actual heap layout after $S_6$

$P_0$: Initial Points-to Analysis

$L_1$: Liveness after $P_0$

$P_1$: Points-to Analysis using $L_1$
Proposed Approach

**S0**: $y = \text{new}

**S1**: $z = \text{new}

**S2**: $t = \text{new}

**S3**: $t.n = x$

**S4**: $x = t$

**S5**: $a = x$

**S6**: $b = a.n$

**S7**: $a.n = y$

**S8**: $b.n = z$

**S9**: use $x.n$

**S10**: exit

---

**Actual heap layout after S6**

- $a$
- $b$
- $n$
- $y$
- $z$

---

**P0**: Initial Points-to Analysis

$PTOUT_6 = PTIN_7 = PTIN_8$:

- $a$
- $b$
- $n$

$y \rightarrow S_0$

$z \rightarrow S_1$

---

**L1**: Liveness after $P_0$

$LVIN_9 = LVOUT_7 = LVOUT_8$:

- $x \rightarrow n_9$

$LVIN_7$ (considering $a \equiv x$):

- $a$
- $x \rightarrow n_9$
- $y$

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

- $b$
- $x \rightarrow n_9$
- $z$

$LVOUT_6 = LVIN_7 \cup LVIN_8$:

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

---

**P1**: Points-to Analysis using $L_1$

$PTOUT_6 = PTIN_7 = PTIN_8$:

- $a$
- $b$
- $n$

$x \rightarrow S_2$

$x.n \rightarrow n$

$y \rightarrow S_0$

$z \rightarrow S_1$

---

**L2**: Liveness after $P_1$

$LVIN_9 = LVOUT_7 = LVOUT_8$:

- $x \rightarrow n_9$

$LVIN_7$ (considering $a \equiv x$):

- $x$
- $y$

$LVIN_8$ (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 $S_6$

![Heap Layout Diagram]

$L_1$: Liveness after $P_0$

$LVIN_9 = LVOUT_7 = LVOUT_8$: 

![Liveness Diagram 1]

$LVIN_7$ (considering $a \equiv x$):

![Liveness Diagram 2]

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

![Liveness Diagram 3]

$LVOUT_6 = LVIN_7 \cup LVIN_8$:

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.
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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V$</td>
<td>Variables</td>
<td>Proportional to program size</td>
</tr>
<tr>
<td>$M$</td>
<td>Memory allocation sites</td>
<td>Proportional to program size</td>
</tr>
<tr>
<td>$R$</td>
<td>Field dereference points</td>
<td>Proportional to program size</td>
</tr>
<tr>
<td>$A$</td>
<td>Access graph nodes</td>
<td>$</td>
</tr>
<tr>
<td>$H$</td>
<td>Heap graph nodes</td>
<td>$</td>
</tr>
</tbody>
</table>

**Definition**

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

- $E_v \subseteq V \times H$
- $E_f \subseteq H \times F \times H$
- $\text{summary} : H \rightarrow \{\text{true}, \text{false}\}$
### Accessor Relationship Graph

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V$</td>
<td>Variables</td>
<td>Proportional to program size</td>
</tr>
<tr>
<td>$M$</td>
<td>Memory allocation sites</td>
<td>Proportional to program size</td>
</tr>
<tr>
<td>$R$</td>
<td>Field dereference points</td>
<td>Proportional to program size</td>
</tr>
<tr>
<td>$A$</td>
<td>Access graph nodes</td>
<td>$</td>
</tr>
<tr>
<td>$H$</td>
<td>Heap graph nodes</td>
<td>$</td>
</tr>
</tbody>
</table>

#### Definition

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

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

\[ \langle E_v, E_f, \text{summary} \rangle \sqsupseteq \langle E'_v, E'_f, \text{summary}' \rangle \text{ if:} \]

- \( E_v \subseteq E'_v \)
- \( E_f \subseteq E'_f \)
- \( \forall k \in H : \text{summary}(k) \Rightarrow \text{summary}'(k) \)
Lattice Representation

Definition

\[ \langle E_v, E_f, \text{summary} \rangle \sqsubseteq \langle E'_v, E'_f, \text{summary}' \rangle \text{ if:} \]

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

Definition

\[ \langle E_v, E_f, \text{summary} \rangle \cap \langle E'_v, E'_f, \text{summary}' \rangle = \langle E''_v, E''_f, \text{summary}'' \rangle \text{ such that:} \]

- \( E''_v = E_v \cup E'_v \)
- \( E''_f = E_f \cup E'_f \)
- \( \forall k \in H : \text{summary}''(k) = \text{summary}(k) \vee \text{summary}'(k) \)
Data Flow Analysis

Normalization: $\Theta(X, L) = \text{Consistency} + \text{Reachability}$
Data Flow Analysis

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

- Data Flow Equations:

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

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

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

\[
\hat{L}_0 = \lambda s \lambda a. \text{false}
\]

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

\[
\forall i \geq 0 : \hat{L}_{i+1} = HLA(\hat{P}_i)
\]
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 $\forall s \in S, \forall a \in AP, \forall b \in AP : \hat{P}_i(s, a, b) \Rightarrow \hat{P}_j(s, a, b)$
Theorem
The results of the second round of heap liveness analysis is the most precise result which is also sound. That is,

∀ k > 0 : \( \hat{L}_k \subseteq \hat{L}_{k+2} \) (Lemma 2)

Proof.
1. ∀ k ≥ 0 : \( \hat{L}_0 \subseteq \hat{L}_k \) (By Definition)
2. ∀ k ≥ 0 : \( \hat{P}_0 \supseteq \hat{P}_k \) (Lemma 1)
3. ∀ k ≥ 0 : \( \hat{L}_1 \supseteq \hat{L}_{k+1} \) (Lemma 2)
4. ∀ k ≥ 0 : \( \hat{P}_1 \subseteq \hat{P}_{k+1} \) (Lemma 1)
5. ∀ k ≥ 0 : \( \hat{L}_2 \subseteq \hat{L}_{k+2} \) (Lemma 2)
6. ∀ k > 0 : \( \hat{L}_2 \subseteq \hat{L}_k \) (Step 3 and 5)
The results of the second round of heap liveness analysis is the most precise result which is also sound. That is, $\forall k > 0 : \hat{L}_2 \subseteq \hat{L}_k$. 

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)
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)**

\[ \forall i, j : \hat{P}_i \subseteq \hat{P}_j \Rightarrow \hat{L}_{i+1} \subseteq \hat{L}_{j+1} \]
The results of the second round of heap liveness analysis is the most precise result which is also sound. That is, \( \forall k > 0 : \hat{L}_2 \subseteq \hat{L}_k \).

**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

1. Background and Motivation
   - Heap Reference Analysis
   - Key Issues

2. Heap Alias Analysis
   - Need for Alias Analysis
   - Existing Abstractions
   - Proposed Abstraction: Accessor Relationship Graph

3. Interprocedural Analysis
   - Existing Frameworks
   - Our Framework: Value Contexts
   - The Role of Call Graphs

4. Access Graphs for Garbage Collection
   - Existing Ideas
   - Novel Technique: Dynamic Heap Pruning

5. Summary & Future Work
No existing implementation of inter-procedural heap reference 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]
  \[ f(\{x, y, z\}) = f(\{x\}) \sqcap f(\{y\}) \sqcap f(\{z\}) \]
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]

\[ f(\{x, y, z\}) = f(\{x\}) \cap f(\{y\}) \cap f(\{z\}) \]

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]
  - \( f(\{x, y, z\}) = f(\{x\}) \cap f(\{y\}) \cap f(\{z\}) \)
  - Functions on \( 2^D \) reduced to functions on \( D \)
  - Modelled as a graph reachability problem
  - Main limitation: Requires distributive flow functions
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]
\[ f(\{x, y, z\}) = f(\{x\}) \cap f(\{y\}) \cap f(\{z\}) \]
- 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]
  - \( f(\{x, y, z\}) = f(\{x\}) \cap f(\{y\}) \cap f(\{z\}) \)
  - 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

\[
y \rightarrow o_1 \xrightarrow{n} o_2
\]

\[
x = y.n
\]

\[
y \rightarrow o_1 \xrightarrow{n} o_2
\]

\[
x
\]
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)
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)
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

- 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)
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]
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]
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
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)
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:

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

- \( X = \langle \textit{method}, \textit{entryValue} \rangle \)

Data Flow Analysis is performed using traditional work-list method. A work-list contains \( \langle \textit{context}, \textit{node} \rangle \) pairs. Call-sites: Find value context \( X = \langle \textit{method}, \textit{entryValue} \rangle \)

- Found: Re-use \( \textit{exitValue}(X) \)
- Not found: Create new \( X \) and add all nodes to work-list.

Exit-sites: Set \( \textit{exitValue}(X) \) and add callers to work-list.
Value contexts:

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

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

- \( X = \langle \text{method}, entryValue \rangle \)
- \( exitValue(X) \)
- Data Flow Analysis is performed using traditional work-list method
Value contexts:

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

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

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

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

```
main()
  p = 5
q = f(p, -3)
r = g(-q)
exit
```

```
f(a, b)
  if (...)
c = a * b
c = g(10)
return c
```

```
g(u)
v = f(-u, u)
return v
```

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
</table>

Value Contexts

Component Lattice
Example - Sign Analysis

main()

n1 p = 5

c1 q = f(p, -3)

c4 r = g(-q)

n6 exit

f(a, b)

n2 if (...)

c2 c = g(10)

n5 return c

g(u)

n3 c = a * b

n3 c = g(10)

n6 return v

Context
Proc. Entry Exit

\( X_0 \) main \( \top \) \( \top \)

Value Contexts

Component Lattice
Example - Sign Analysis

```
main()

p = 5

q = f(p, -3)

c1

r = g(-q)

c4

exit

f(a, b)

n2

if (...)

c2

v = f(-u, u)

c3

n6

return v

c5

return c

n5

c = a * b

n3

c = g(10)

Context

Proc. | Entry | Exit

| X0 | main | T | T |

Value Contexts

Component Lattice

Rohan Padhye (IIT Bombay)
Interprocedural Heap Analysis
MTP 23 / 32
```
Example - Sign Analysis

```
main()
p = 5
q = f(p, -3)
r = g(-q)
exit
```

```
f(a, b)
if (...)
c = a * b
c = g(10)
return c
```

```
g(u)
v = f(-u, u)
return v
```

Value Contexts:

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₀</td>
<td>main</td>
<td>⊤</td>
<td>⊤</td>
</tr>
</tbody>
</table>

Component Lattice:
Example - Sign Analysis

main()

p = 5

q = f(p, -3)

c1

r = g(-q)

c4

n1

n2

if (…)

c2

c = a * b

n3

n4

n5

return c

n6

exit

f(a, b)

c = g(10)

n7

n8

return v

g(u)

⟨X₀, ⊤⟩

⟨X₀, p⁺⟩

c1

⟨X₀, T⟩

⟨X₀, p⁺⟩

c1

⟨X₀, T⟩

n1

n2

n3

n4

n5

n6

Context

Proc.

Entry

Exit

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₀</td>
<td>main</td>
<td>⊤</td>
<td>⊤</td>
</tr>
<tr>
<td>X₁</td>
<td>f</td>
<td>a⁺ b⁻</td>
<td>⊤</td>
</tr>
</tbody>
</table>

Value Contexts

Component Lattice
Example - Sign Analysis

```
main()

p = 5

q = f(p, -3)

c1

r = g(-q)

c2

exit

f(a, b)

if (...) n2

c = a * b

c = g(10)

return c

g(u)

v = f(-u, u)

return v
```

Value Contexts

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X_0)</td>
<td>main</td>
<td>(\top)</td>
<td>(\top)</td>
</tr>
<tr>
<td>(X_1)</td>
<td>f</td>
<td>(a^+ b^-)</td>
<td>(\top)</td>
</tr>
</tbody>
</table>

Context Transitions

\[ X_0 \xrightarrow{c_1} X_1 \]

Component Lattice
Example - Sign Analysis

```
Example - Sign Analysis

main()
p = 5
q = f(p, -3)
r = g(-q)
exit

f(a, b)
if (...) c = a * b c = g(10)
return c

g(u)
v = f(-u, u)
return v

⟨X₀, T⟩
⟨X₀, p⁺⟩
c₁ q = f(p, -3)
c₄ r = g(-q)
⟩
⟨X₁, a⁺b⁻⟩

⟨X₂, u⁺⟩

Context Proc. Entry Exit
X₀ main ⊤ ⊤
X₁ f a⁺b⁻ ⊤

Value Contexts

Context Transitions

Component Lattice

Rohan Padhye (IIT Bombay)
Interprocedural Heap Analysis
MTP 23 / 32
```
Example - Sign Analysis

```
main()
    p = 5
    q = f(p, -3)
    r = g(-q)
    exit
```

```
f(a, b)
    if (...)
        c = a * b
        c = g(10)
    return c
```

```
g(u)
    v = f(-u, u)
    return v
```

**Context**

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₀</td>
<td>main</td>
<td>⊤</td>
<td>⊤</td>
</tr>
<tr>
<td>X₁</td>
<td>f</td>
<td>a⁺b⁻</td>
<td>⊤</td>
</tr>
</tbody>
</table>

**Value Contexts**

```
⟨X₀, T⟩
⟨X₀, p⁺⟩
c₁  q = f(p, -3)
c₄  r = g(-q)
```

```
⟨X₁, a⁺b⁻⟩
⟨X₁, a⁺b⁻⟩
c₂  c = a * b
c₃  c = g(10)
```

```
⟨X₂, u⁺⟩
⟨X₂, u⁺⟩
c₅  return c
c₆  return v
```

**Context Transitions**

```
X₀  c₁  X₁
```

```
X₀  c₁  X₁
```

**Component Lattice**
Example - Sign Analysis

```
main()
```

```
f(a, b)
```

```
g(u)
```

```
exit
```

```
f(a, b)
```

```
if (...)
```

```
c = a * b
```

```
c = g(10)
```

```
r = g(-q)
```

```
p = 5
```

```
q = f(p, -3)
```

```
return v
```

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_0$</td>
<td>main</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
<tr>
<td>$X_1$</td>
<td>f</td>
<td>$a^+b^-$</td>
<td>$\top$</td>
</tr>
<tr>
<td>$X_2$</td>
<td>g</td>
<td>$u^+$</td>
<td>$\top$</td>
</tr>
</tbody>
</table>

Value Contexts

Context Transitions

Component Lattice
Example - Sign Analysis

main()

\( p = 5 \)

\( n_1 \)

\( \langle X_0, \top \rangle \)

\( \langle X_0, p^+ \rangle \)

\( c_1 \)

\( q = f(p, -3) \)

\( c_4 \)

\( r = g(-q) \)

\( n_6 \)

\( \text{exit} \)

\( \langle X_0, T \rangle \)

\( f(a, b) \)

\( \langle X_1, a^+ b^- \rangle \)

\( n_2 \)

\( \text{if (...)} \)

\( c_1 \)

\( c = a * b \)

\( c_2 \)

\( c = g(10) \)

\( n_3 \)

\( n_5 \)

\( \text{return c} \)

\( c_3 \)

\( v = f(-u, u) \)

\( n_6 \)

\( \text{return v} \)

\( \langle X_1, a^+ b^- \rangle \)

\( \langle X_1, a^+ b^- \rangle \)

Context

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_0 )</td>
<td>main</td>
<td>( \top )</td>
<td>( \top )</td>
</tr>
<tr>
<td>( X_1 )</td>
<td>f</td>
<td>( a^+ b^- )</td>
<td>( \top )</td>
</tr>
<tr>
<td>( X_2 )</td>
<td>g</td>
<td>( u^+ )</td>
<td>( \top )</td>
</tr>
</tbody>
</table>

Value Contexts

Context Transitions

Component Lattice

Rohan Padhye (IIT Bombay) Interprocedural Heap Analysis
Example - Sign Analysis

```
main()
p = 5
q = f(p, -3)
r = g(-q)
exit
```

```
f(a, b)
if (...)
c = a * b
c = g(10)
```

```
g(u)
v = f(-u, u)
return v
```

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_0$</td>
<td>main</td>
<td>⊤</td>
<td>⊤</td>
</tr>
<tr>
<td>$X_1$</td>
<td>f</td>
<td>$a^+b^-$</td>
<td>⊤</td>
</tr>
<tr>
<td>$X_2$</td>
<td>g</td>
<td>$u^+$</td>
<td>⊤</td>
</tr>
</tbody>
</table>

Value Contexts

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_0$</td>
<td>$X_1$</td>
<td>$X_2$</td>
<td></td>
</tr>
</tbody>
</table>

Context Transitions

Component Lattice

Rohan Padhye (IIT Bombay)
Interprocedural Heap Analysis

MTP 23 / 32
Example - Sign Analysis

```
main()

p = 5

q = f(p, -3)

c1

r = g(-q)

c4

exit

f(a, b)

n2 if (...)

n3 c = a * b
c2 c = g(10)

n5 return c

n6 return v

g(u)

n

⟨X₀, ⊤⟩

⟨X₀, p⁺⟩

c1

⟨X₁, a⁺b⁻⟩

⟨X₀, T⟩

n1 p = 5

 ⟨X₁, a⁺b⁻⟩

n2 if (...)

n3 c = a * b
c2 c = g(10)

n5 return c

n6 return v

g(u)

n

⟨X₂, u⁺⟩

c3

⟨X₁, a⁺b⁻⟩

⟨X₁, a⁺b⁻⟩

⟨X₂, a⁻b⁺⟩

⟨X₃, a⁻b⁺⟩

⟨X₃, a⁻b⁺⟩

Context
Proc. Entry Exit

X₀ main ⊤ ⊤
X₁ f a⁺b⁻ ⊤ ⊤
X₂ g u⁺ ⊤ ⊤
X₃ f a⁻b⁺ ⊤ ⊤

Value Contexts

Context Transitions

Component Lattice

Rohan Padhye (IIT Bombay) Interprocedural Heap Analysis MTP 23 / 32
Example - Sign Analysis

```
main()

p = 5

q = f(p, -3)

c = a * b

r = g(-q)

exit
```

```
f(a, b)

if (...)

c = a * b

c = g(10)

return c
```

g(u)

```
v = f(-u, u)

return v
```
Example - Sign Analysis

```
main()
    p = 5
    q = f(p, -3)
    r = g(-q)
    exit
```

```
f(a, b)
    if (...)
        c = a * b
    return c
```

```
g(u)
    v = f(-u, u)
    return v
```

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₀</td>
<td>main</td>
<td>⊤</td>
<td>⊤</td>
</tr>
<tr>
<td>X₁</td>
<td>f</td>
<td>a⁺b⁻</td>
<td>⊤</td>
</tr>
<tr>
<td>X₂</td>
<td>g</td>
<td>u⁺</td>
<td>⊤</td>
</tr>
<tr>
<td>X₃</td>
<td>f</td>
<td>a⁻b⁺</td>
<td>⊤</td>
</tr>
</tbody>
</table>

Value Contexts

Context Transitions

Component Lattice

Rohan Padhye (IIT Bombay)  Interprocedural Heap Analysis  MTP 23 / 32
Example - Sign Analysis

```
main()

p = 5

q = f(p, -3)

r = g(-q)

exit
```

```
f(a, b)

if (...)

c = a * b

c = g(10)

return c
```

```
g(u)

v = f(-u, u)

return v
```

Context

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₀</td>
<td>main</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>X₁</td>
<td>f</td>
<td>aᵇ⁻</td>
<td>T</td>
</tr>
<tr>
<td>X₂</td>
<td>g</td>
<td>u⁺</td>
<td>T</td>
</tr>
<tr>
<td>X₃</td>
<td>f</td>
<td>a⁻ᵇ⁺</td>
<td>T</td>
</tr>
</tbody>
</table>

Value Contexts

Component Lattice

Context Transitions

Rohan Padhye (IIT Bombay)
**Example - Sign Analysis**

```
main()

p = 5
q = f(p, -3)
r = g(-q)

exit
```

```
f(a, b)

if (...)
c = a * b
c = g(10)

return c
```

```
g(u)
v = f(-u, u)

return v
```

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₀</td>
<td>main</td>
<td>⊤</td>
<td>⊤</td>
</tr>
<tr>
<td>X₁</td>
<td>f</td>
<td>a⁺b⁻</td>
<td>⊤</td>
</tr>
<tr>
<td>X₂</td>
<td>g</td>
<td>u⁺</td>
<td>⊤</td>
</tr>
<tr>
<td>X₃</td>
<td>f</td>
<td>a⁻b⁺</td>
<td>⊤</td>
</tr>
</tbody>
</table>

**Value Contexts**

**Context Transitions**

**Component Lattice**
Example - Sign Analysis

```
main()
p = 5
q = f(p, -3)
r = g(-q)
exit
```

```
f(a, b)
if (...) c = a * b
c = g(10)
return c
```

```
g(u)
v = f(-u, u)
return v
```

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₀</td>
<td>main</td>
<td>⊤</td>
<td>⊤</td>
</tr>
<tr>
<td>X₁</td>
<td>f</td>
<td>a⁺ b⁻</td>
<td>⊤</td>
</tr>
<tr>
<td>X₂</td>
<td>g</td>
<td>u⁺</td>
<td>⊤</td>
</tr>
<tr>
<td>X₃</td>
<td>f</td>
<td>a⁻ b⁺</td>
<td>⊤</td>
</tr>
</tbody>
</table>

Value Contexts

Context Transitions

Component Lattice
Example - Sign Analysis

```
main()
p = 5
q = f(p, -3)
r = g(-q)
exit
```

```
f(a, b)
if (...)
c = a * b
c = g(10)
return c
```

```
g(u)
v = f(-u, u)
return v
```

Context Transitions:

Value Contexts:

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₀</td>
<td>main</td>
<td>⊤</td>
<td>⊤</td>
</tr>
<tr>
<td>X₁</td>
<td>f</td>
<td>a⁺b⁻</td>
<td>⊤</td>
</tr>
<tr>
<td>X₂</td>
<td>g</td>
<td>u⁺</td>
<td>⊤</td>
</tr>
<tr>
<td>X₃</td>
<td>f</td>
<td>a⁻b⁺</td>
<td>⊤</td>
</tr>
</tbody>
</table>

Component Lattice:
Example - Sign Analysis

\[ (X_0, T) \]
\[ p = 5 \]
\[ (X_0, p^+) \]
\[ q = f(p, -3) \]
\[ r = g(-q) \]
\[ \text{exit} \]

\[ f(a, b) \]
\[ if (...) \]
\[ c = a \times b \]
\[ c = g(10) \]
\[ return c \]

\[ g(u) \]
\[ v = f(-u, u) \]
\[ return v \]

\[ ⟨X_0, T⟩ \]
\[ n_1 \]
\[ p = 5 \]
\[ (X_0, p^+) \]
\[ c_1 \]
\[ q = f(p, -3) \]
\[ c_4 \]
\[ r = g(-q) \]
\[ n_6 \]
\[ \text{exit} \]

\[ ⟨X_1, a^+ b^-⟩ \]
\[ ⟨X_3, a^- b^+⟩ \]
\[ n_2 \]
\[ if (...) \]
\[ (X_3, a^- b^+) \]
\[ (X_3, a^- b^+) \]
\[ n_3 \]
\[ c = a \times b \]
\[ c_2 \]
\[ c_3 \]
\[ c_4 \]
\[ n_5 \]
\[ return c \]

\[ ⟨X_2, u^+⟩ \]
\[ n_6 \]
\[ \text{return } v \]

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₀</td>
<td>main</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>X₁</td>
<td>f</td>
<td>a⁺b⁻</td>
<td>T</td>
</tr>
<tr>
<td>X₂</td>
<td>g</td>
<td>u⁺</td>
<td>T</td>
</tr>
<tr>
<td>X₃</td>
<td>f</td>
<td>a⁻b⁺</td>
<td>T</td>
</tr>
</tbody>
</table>

Value Contexts

Context Transitions

Component Lattice
Example - Sign Analysis

```
main()
p = 5
q = f(p, -3)
r = g(-q)
exit
```

```
f(a, b)
if (...)
c = a * b
c = g(10)
return c
```

```
g(u)
v = f(-u, u)
return v
```

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₀</td>
<td>main</td>
<td>⊤</td>
<td>⊤</td>
</tr>
<tr>
<td>X₁</td>
<td>f</td>
<td>a⁺b⁻</td>
<td>⊤</td>
</tr>
<tr>
<td>X₂</td>
<td>g</td>
<td>u⁺</td>
<td>⊤</td>
</tr>
<tr>
<td>X₃</td>
<td>f</td>
<td>a⁻b⁺</td>
<td>⊤</td>
</tr>
</tbody>
</table>

Value Contexts

Context Transitions

Component Lattice
Example - Sign Analysis

```
main()
  p = 5
  q = f(p, -3)
  r = g(-q)
  exit

f(a, b)
  if (...)
    c = a * b
    c = g(10)
    return c

f(a, b)
  v = f(-u, u)
  return v
```

Context

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₀</td>
<td>main</td>
<td>⊤</td>
<td>⊤</td>
</tr>
<tr>
<td>X₁</td>
<td>f</td>
<td>a⁺b⁻</td>
<td>⊤</td>
</tr>
<tr>
<td>X₂</td>
<td>g</td>
<td>u⁺</td>
<td>⊤</td>
</tr>
<tr>
<td>X₃</td>
<td>f</td>
<td>a⁻b⁺</td>
<td>a⁻b⁺c⁻</td>
</tr>
</tbody>
</table>

Value Contexts

Context Transitions

Component Lattice
Example - Sign Analysis

```plaintext
main()

p = 5

q = f(p, -3)

r = g(-q)

exit

f(a, b)

if (...)

c = a * b

c = g(10)

return c

g(u)

v = f(-u, u)

return v
```

**Context Transitions**

- \( X_0 \) \( \xrightarrow{c_1} \) \( X_1 \) \( \xrightarrow{c_2} \) \( X_2 \)
- \( X_0 \) \( \xrightarrow{c_3} \) \( X_3 \)

**Value Contexts**

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_0 )</td>
<td>main</td>
<td>( \top )</td>
<td>( \top )</td>
</tr>
<tr>
<td>( X_1 )</td>
<td>f</td>
<td>( a^+ b^- )</td>
<td>( \top )</td>
</tr>
<tr>
<td>( X_2 )</td>
<td>g</td>
<td>( u^+ )</td>
<td>( \top )</td>
</tr>
<tr>
<td>( X_3 )</td>
<td>f</td>
<td>( a^- b^+ )</td>
<td>( a^- b^+ c^- )</td>
</tr>
</tbody>
</table>

**Component Lattice**
Example - Sign Analysis

main()

\[ p = 5 \]

\[ q = f(p, -3) \]

\[ r = g(-q) \]

\[ c_1 \]

\[ c_4 \]

\[ \langle X_0, \top \rangle \]

\[ \langle X_0, p^+ \rangle \]

\[ n_1 \]

\[ n_6 \] exit

\[ \langle X_3, a^+ b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ c^- \rangle \]

\[ \langle X_1, a^+ b^- \rangle \]

\[ \langle X_1, a^- b^- \rangle \]

\[ n_2 \]

\[ n_3 \]

\[ c_2 \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^+ c^- \rangle \]

\[ c_3 \]

\[ v = f(-u, u) \]

\[ \langle X_2, u^+ \rangle \]

\[ \langle X_2, u^+ v^- \rangle \]

\[ n_5 \]

\[ \text{return } c \]

\[ n_6 \]

\[ \text{return } v \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_2, u^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]

\[ \langle X_3, a^- b^+ \rangle \]

\[ \langle X_3, a^- b^- \rangle \]
Example - Sign Analysis

main()

\( p = 5 \)

\( q = f(p, -3) \)

\( r = g(-q) \)

exit

\( f(a, b) \)

\( c = a \cdot b \)

\( c = g(10) \)

return \( c \)

g(u)

\( v = f(-u, u) \)

return \( v \)

Context

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_0 )</td>
<td>main</td>
<td>( \top )</td>
<td>( \top )</td>
</tr>
<tr>
<td>( X_1 )</td>
<td>f</td>
<td>( a^+ b^- )</td>
<td>( \top )</td>
</tr>
<tr>
<td>( X_2 )</td>
<td>g</td>
<td>( u^+ )</td>
<td>( u^+ v^- )</td>
</tr>
<tr>
<td>( X_3 )</td>
<td>f</td>
<td>( a^- b^+ )</td>
<td>( a^- b^+ c^- )</td>
</tr>
</tbody>
</table>

Value Contexts

Context Transitions

Component Lattice
Example - Sign Analysis

main()

p = 5

q = f(p, -3)

r = g(-q)

exit

f(a, b)

if (...)

c = a * b

c = g(10)

return c

g(u)

v = f(-u, u)

return v

Value Contexts

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>X_0</td>
<td>main</td>
<td>\top</td>
<td>\top</td>
</tr>
<tr>
<td>X_1</td>
<td>f</td>
<td>a^+b^-</td>
<td>\top</td>
</tr>
<tr>
<td>X_2</td>
<td>g</td>
<td>u^+</td>
<td>u^+v^-</td>
</tr>
<tr>
<td>X_3</td>
<td>f</td>
<td>a^-b^+</td>
<td>a^-b^+c^-</td>
</tr>
</tbody>
</table>

Context Transitions

Component Lattice
Example - Sign Analysis

```
main()
p = 5
q = f(p, -3)
r = g(-q)
exit
```

\[
\langle X_0, \top \rangle
\]

\[
\langle X_0, p^+ \rangle
\]

\[
c_1 \quad q = f(p, -3)
\]

\[
c_2 \quad c = a \times b
\]

\[
c_3 \quad c = g(10)
\]

\[
c_4 \quad r = g(-q)
\]

\[
c_5 \quad \text{return } c
\]

\[
c_6 \quad \text{return } v
\]

**Context Transitions**

\[
X_0 \xrightarrow{c_1} X_1 \xrightarrow{c_2} X_2 \xrightarrow{c_3} X_3
\]

**Component Lattice**

\[
\begin{array}{c}
\top \\
- \\
0 \\
+ \\
\bot
\end{array}
\]

**Value Contexts**

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>X_0</td>
<td>main</td>
<td>\top</td>
<td>\top</td>
</tr>
<tr>
<td>X_1</td>
<td>f</td>
<td>(a^+ b^-)</td>
<td>\top</td>
</tr>
<tr>
<td>X_2</td>
<td>g</td>
<td>(u^+)</td>
<td>(u^+ v^-)</td>
</tr>
<tr>
<td>X_3</td>
<td>f</td>
<td>(a^- b^+)</td>
<td>(a^- b^+ c^-)</td>
</tr>
</tbody>
</table>
Example - Sign Analysis

```plaintext
main()

\( p = 5 \)

\( q = f(p, -3) \)

\( r = g(-q) \)

\( \text{exit} \)

\( f(a, b) \)

\( \text{if (...) } \)

\( c = a \times b \)

\( c = g(10) \)

\( v = f(-u, u) \)

\( \text{return } v \)

\( g(u) \)

\( \langle X_0, \top \rangle \)

\( n_1 \)

\( \langle X_0, p^+ \rangle \)

\( c_1 \)

\( q = f(p, -3) \)

\( \langle X_0, X_1, a^+b^- \rangle \)

\( n_2 \)

\( \langle X_1, a^+b^- \rangle \)

\( \langle X_3, a^-b^+ \rangle \)

\( n_3 \)

\( c = a \times b \)

\( \langle X_1, a^+b^- \rangle \)

\( \langle X_3, a^-b^+ \rangle \)

\( n_4 \)

\( \langle X_1, a^+b^-c^- \rangle \)

\( \langle X_3, a^-b^+c^- \rangle \)

\( n_5 \)

\( \langle X_3, a^-b^+c^- \rangle \)

\( \langle X_3, a^-b^+c^- \rangle \)

\( n_6 \)

\( \langle X_2, u^+ \rangle \)

\( \langle X_2, u^+v^- \rangle \)

\( c_3 \)

\( \text{Value Contexts} \)

\( \text{Context} \)

\( \text{Proc.} \)

\( \text{Entry} \)

\( \text{Exit} \)

\( X_0 \)

main

\( \top \)

\( \top \)

\( X_1 \)

f

\( a^+b^- \)

\( \top \)

\( X_2 \)

g

\( u^+ \)

\( u^+v^- \)

\( X_3 \)

f

\( a^-b^+ \)

\( a^-b^+c^- \)

\( X_3 \)

\( a^-b^+ \)

\( a^-b^+c^- \)

\( \text{Context Transitions} \)

\( X_0 \)

\( X_1 \)

\( X_2 \)

\( X_3 \)

\( c_1 \)

\( c_2 \)

\( c_3 \)

\( c_2 \)

\( \text{Component Lattice} \)

\( \top \)

\( - \)

\( 0 \)

\( + \)

\( \bot \)

Rohan Padhye (IIT Bombay) Interprocedural Heap Analysis MTP 23 / 32
```
Example - Sign Analysis

```
main()
p = 5
q = f(p, -3)
r = g(-q)
exit
```

```
f(a, b)
if (...)
c = a * b
return c
g(u)
v = f(-u, u)
return v
```

```
⟨X₀, T⟩
⟨X₀, p⁺⟩
c₁ q = f(p, -3)
c₄ r = g(-q)
n₁

⟨X₁, a⁺b⁻⟩
⟨X₃, a⁻b⁺⟩
c₂ c = g(10)

⟨X₁, a⁺b⁻⟩
⟨X₃, a⁻b⁺⟩
if (...)

⟨X₁, a⁺b⁻⟩
⟨X₃, a⁻b⁺⟩
⟨X₁, a⁺b⁻c⁻⟩
⟨X₃, a⁻b⁺c⁻⟩

⟨X₁, a⁺b⁻⟩
⟨X₃, a⁻b⁺⟩
⟨X₁, a⁺b⁻c⁻⟩
⟨X₃, a⁻b⁺c⁻⟩

⟨X₁, a⁺b⁻⟩
⟨X₃, a⁻b⁺⟩
⟨X₁, a⁺b⁻c⁻⟩
⟨X₃, a⁻b⁺c⁻⟩

⟨X₂, u⁺⟩
⟨X₂, u⁺v⁻⟩
```

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₀</td>
<td>main</td>
<td>T⁺</td>
<td>T⁺</td>
</tr>
<tr>
<td>X₁</td>
<td>f</td>
<td>a⁺b⁻</td>
<td>T⁺</td>
</tr>
<tr>
<td>X₂</td>
<td>g</td>
<td>u⁺</td>
<td>u⁺v⁻</td>
</tr>
<tr>
<td>X₃</td>
<td>f</td>
<td>a⁻b⁺</td>
<td>a⁻b⁺c⁻</td>
</tr>
</tbody>
</table>

Value Contexts

Context Transitions

Component Lattice
Example - Sign Analysis

```
main()
p = 5
q = f(p, -3)
r = g(-q)
exit

f(a, b)
if (...)
c = a * b
c = g(10)
return c

g(u)
v = f(-u, u)
return v
```

Context Transitions

```
X0 → X1 → X2 → X3
```

Component Lattice

```
- 0 +
```

Value Contexts

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>X0</td>
<td>main</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>X1</td>
<td>f</td>
<td>a+b^-</td>
<td>a+b^- c^-</td>
</tr>
<tr>
<td>X2</td>
<td>g</td>
<td>u^+</td>
<td>u^+ v^-</td>
</tr>
<tr>
<td>X3</td>
<td>f</td>
<td>a-b^+</td>
<td>a-b^+ c^-</td>
</tr>
</tbody>
</table>
### Context Transitions

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_0$</td>
<td>main</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
<tr>
<td>$X_1$</td>
<td>$f$</td>
<td>$a^+ b^-$</td>
<td>$a^+ b^- c^-$</td>
</tr>
<tr>
<td>$X_2$</td>
<td>$g$</td>
<td>$u^+$</td>
<td>$u^+ v^-$</td>
</tr>
<tr>
<td>$X_3$</td>
<td>$f$</td>
<td>$a^- b^+$</td>
<td>$a^- b^+ c^-$</td>
</tr>
</tbody>
</table>

Value Contexts

\[ \langle X_0, \top \rangle \]
\[ \langle X_0, p^+ \rangle \]
\[ \langle X_0, p^+ q^- \rangle \]
\[ \langle X_1, a^+ b^- \rangle \]
\[ \langle X_3, a^- b^- c^- \rangle \]
\[ \langle X_3, a^- b^+ c^- \rangle \]
\[ \langle X_1, a^+ b^- c^- \rangle \]
\[ \langle X_3, a^- b^+ c^- \rangle \]

Component Lattice

\[ \downarrow \]
\[ - \]  \[ 0 \]  \[ + \]  \[ \downarrow \]
Example - Sign Analysis

main()

\( p = 5 \)

\( q = f(p, -3) \)

\( r = g(-q) \)

\( \text{exit} \)

Context

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_0 )</td>
<td>main</td>
<td>( \top )</td>
<td>( \top )</td>
</tr>
<tr>
<td>( X_1 )</td>
<td>( f )</td>
<td>( a^+ b^- )</td>
<td>( a^+ b^- c^- )</td>
</tr>
<tr>
<td>( X_2 )</td>
<td>( g )</td>
<td>( u^+ )</td>
<td>( u^+ v^- )</td>
</tr>
<tr>
<td>( X_3 )</td>
<td>( f )</td>
<td>( a^- b^+ )</td>
<td>( a^- b^+ c^- )</td>
</tr>
</tbody>
</table>

Value Contexts

f(a, b)

if (...)

\( c = a \times b \)

\( c = g(10) \)

\( \text{return } c \)

\( \text{return } v \)

g(u)

\( v = f(-u, u) \)

\( \text{return } v \)

Context Transitions

Component Lattice
Example - Sign Analysis

```
main()
p = 5
q = f(p, -3)
r = g(-q)
exit
```

```
f(a, b)
if (...)
c = a * b
c = g(10)
return c
```

```
g(u)
v = f(-u, u)
return v
```

<table>
<thead>
<tr>
<th>Context</th>
<th>Proc.</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₀</td>
<td>main</td>
<td>⊤</td>
<td>⊤</td>
</tr>
<tr>
<td>X₁</td>
<td>f</td>
<td>a⁺b⁻</td>
<td>a⁺b⁻c⁻</td>
</tr>
<tr>
<td>X₂</td>
<td>g</td>
<td>u⁺</td>
<td>u⁺v⁻</td>
</tr>
<tr>
<td>X₃</td>
<td>f</td>
<td>a⁻b⁺</td>
<td>a⁻b⁺c⁻</td>
</tr>
</tbody>
</table>

Value Contexts

Context Transitions

Component Lattice

Rohan Padhye (IIT Bombay)
Example - Sign Analysis

main()

\( n_1 \)

\( p = 5 \)

\( n_1 \)

\( \langle X_0, p^+ \rangle \)

\( c_1 \)

\( q = f(p, -3) \)

\( \langle X_0, p^+ q^- \rangle \)

\( c_4 \)

\( r = g(-q) \)

\( \langle X_0, p^+ q^- r^- \rangle \)

\( n_6 \)

exit

\( f(a, b) \)

\( n_2 \)

if (…)

\( n_3 \)

c = a \times b

\( n_3 \)

\( \langle X_1, a^+ b^- \rangle \)

\( \langle X_3, a^- b^+ \rangle \)

\( c_2 \)

c = g(10)

\( \langle X_1, a^+ b^- \rangle \)

\( \langle X_3, a^- b^+ \rangle \)

\( c_3 \)

\( v = f(-u, u) \)

\( \langle X_2, u^+ \rangle \)

\( n_6 \)

return v

\( \langle X_2, u^+ v^- \rangle \)

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

Rohan Padhye (IIT Bombay)
Interprocedural Heap Analysis
MTP 23 / 32
Implementation Framework

InterProceduralAnalysis\(<M,N,A>\>

- \+ topValue() : A
- \+ boundaryValue(M) : A
- \+ copy(A) : A
- \+ meet(A,A) : A
- \+ normalFlowFunction(Context\(<M,N,A>\), N, A) : A
- \+ callEntryFlowFunction(Context\(<M,N,A>\), M, N, A) : A
- \+ callExitFlowFunction(Context\(<M,N,A>\), M, N, A) : A
- \+ callLocalFlowFunction(Context\(<M,N,A>\), N, A) : A
- \+ programRepresentation() : ProgramRepresentation\(<M,N>\>
- \+ doAnalysis() : void
- \+ getContexts() : Map\(<M,List<Context<M,N,A>>\>
- \+ getMeetOverPathsSolution() : DataFlowSolution\(<M,N,A>\>

Context\(<M,N,A>\>

- \+ getMethod(): M
- \+ getEntryValue() : A
- \+ getExitValue() : A
- \+ getValueBefore(N) : A
- \+ getValueAfter(N) : A

ProgramRepresentation\(<M,N>\>

- \+ getEntryPoints() : List\(<M>\>
- \+ getControlFlowGraph(M) : DirectedGraph\(<N>\>
- \+ isCall(N) : boolean
- \+ resolveTargets(M, N) : List\(<M>\>

ForwardInterProceduralAnalysis\(<M,N,A>\>

- \+ doAnalysis() : void

BackwardInterProceduralAnalysis\(<M,N,A>\>

- \+ doAnalysis() : void

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
Context-sensitivity only useful if call graph is precise

OOP: Use points-to analysis to resolve virtual calls

Imprecise points-to analysis ⇒ “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 ⇒ “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)
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

1 Background and Motivation
   - Heap Reference Analysis
   - Key Issues

2 Heap Alias Analysis
   - Need for Alias Analysis
   - Existing Abstractions
   - Proposed Abstraction: Accessor Relationship Graph

3 Interprocedural Analysis
   - Existing Frameworks
   - Our Framework: Value Contexts
   - The Role of Call Graphs

4 Access Graphs for Garbage Collection
   - Existing Ideas
   - Novel Technique: Dynamic Heap Pruning

5 Summary & Future Work
Access Graphs for Garbage Collection

How to use access graphs for improving garbage collection?
How to use access graphs for improving garbage collection?

1. Assign null to dead access paths.
How to use access graphs for improving garbage collection?

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

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.
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 nullify access paths that are not provably safe to dereference.
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 nullify access paths that are not provably safe to dereference.
   - The safety analyses themselves depend on alias information.

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.
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 nullify 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.

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.
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 nullify 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 nullification 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.

Rohan Padhye (IIT Bombay)
Interprocedural Heap Analysis
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 nullify 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 nullification of same reference from aliased access paths.

2. Augment garbage collector to traverse access graphs.

Rohan Padhye (IIT Bombay)
Interprocedural Heap 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 nullify 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 nullification 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 nullify 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 nullification 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.
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 nullify 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 nullification 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.
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 `nullify` 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 `nullification` 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 `nullify` 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 `nullification` 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!

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 \( \sigma \) using the sequence of return addresses from the bottom-of-stack.
   3. Determine the value context \( X \) by traversing \( \sigma \) 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.
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 $\sigma$ using the sequence of return addresses from the bottom-of-stack.
   3. Determine the value context $X$ by traversing $\sigma$ 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.
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 $\sigma$ using the sequence of return addresses from the bottom-of-stack.
   3. Determine the value context $X$ by traversing $\sigma$ in the context transition graph.
   4. Retrieve the access graphs for point $P$ in context $X$.
      1. 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.

Rohan Padhye (IIT Bombay)
Interprocedural Heap Analysis
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).

4. 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.
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 \( \sigma \) using the sequence of return addresses from the bottom-of-stack.

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.
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 $\sigma$ using the sequence of return addresses from the bottom-of-stack.
   3. Determine the value context $X$ by traversing $\sigma$ 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.

6. 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.
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 $\sigma$ using the sequence of return addresses from the bottom-of-stack.
   3. Determine the value context $X$ by traversing $\sigma$ in the context transition graph.
   4. Retrieve the access graphs for point $P$ in context $X$.

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.

Resume the program. Let garbage collection run as normal.

Rohan Padhye (IIT Bombay) Interprocedural Heap Analysis MTP 29 / 32
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 $\sigma$ using the sequence of return addresses from the bottom-of-stack.
   3. Determine the value context $X$ by traversing $\sigma$ 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.

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.

Resume the program. Let garbage collection run as normal.
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 \( \sigma \) using the sequence of return addresses from the bottom-of-stack.
   3. Determine the value context \( X \) by traversing \( \sigma \) 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 $\sigma$ using the sequence of return addresses from the bottom-of-stack.
   3. Determine the value context $X$ by traversing $\sigma$ 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 $\sigma$ using the sequence of return addresses from the bottom-of-stack.
   3. Determine the value context $X$ by traversing $\sigma$ 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 $\sigma$ using the sequence of return addresses from the bottom-of-stack.
   3. Determine the value context $X$ by traversing $\sigma$ 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.

Rohan Padhye (IIT Bombay) Interprocedural Heap Analysis MTP 29 / 32
Outline

1 Background and Motivation
   • Heap Reference Analysis
   • Key Issues

2 Heap Alias Analysis
   • Need for Alias Analysis
   • Existing Abstractions
   • Proposed Abstraction: Accessor Relationship Graph

3 Interprocedural Analysis
   • Existing Frameworks
   • Our Framework: Value Contexts
   • The Role of Call Graphs

4 Access Graphs for Garbage Collection
   • Existing Ideas
   • Novel Technique: Dynamic Heap Pruning

5 Summary & Future Work
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.