Interprocedural Data Flow Analysis

Uday Khedker

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

October 2013
Part 1

About These Slides
Copyright

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


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


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

• Issues in interprocedural analysis
• Functional approach
• The classical call strings approach
• Value based termination of call string construction
Part 2

Issues in Interprocedural Analysis
Interprocedural Analysis: Overview

- Extends the scope of data flow analysis across procedure boundaries
  Incorporates the effects of
  - procedure calls in the caller procedures, and
  - calling contexts in the callee procedures

- Approaches:
  - Generic: Call strings approach, functional approach
  - Problem specific: Alias analysis, Points-to analysis, Partial redundancy elimination, Constant propagation
Inherited and Synthesized Data Flow Information

\[ x' = f_r(x) \]
\[ y' = f_r(y) \]

<table>
<thead>
<tr>
<th>Data Flow Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>( x' )</td>
</tr>
<tr>
<td>( y' )</td>
</tr>
</tbody>
</table>
Inherited and Synthesized Data Flow Information

• Example of uses of inherited data flow information

Answering questions about formal parameters and global variables:
  ▶ Which variables are constant?
  ▶ Which variables aliased with each other?
  ▶ Which locations can a pointer variable point to?

• Examples of uses of synthesized data flow information

Answering questions about side effects of a procedure call:
  ▶ Which variables are defined or used by a called procedure?
    (Could be local/global/formal variables)

• Most of the above questions may have a *May* or *Must* qualifier
Program Representation for Interprocedural Data Flow Analysis: Call Multi-Graph

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

Supergraphs of procedures

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

Supergraphs of procedures

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

Supergraphs of procedures

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

Supergraphs of procedures

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

Supergraphs of procedures

Call multi-graph

Oct 2013
Program Representation for Interprocedural Data Flow Analysis: Supergraph

\[ S_{main} \]
\[ a + b \]
Call \( p \)

\[ S_p \]
Call \( q \)

\[ n_1 \]
\[ d = a + b \]
Call \( p \)

\[ n_2 \]
\[ a = 1 \]
Call \( p \)

\[ n_3 \]
\[ E_p \]

\[ n_4 \]
\[ E_q \]

\[ E_{main} \]
Program Representation for Interprocedural Data Flow Analysis: Supergraph
Program Representation for Interprocedural Data Flow Analysis: Supergraph

\[ S_{main} : a + b \]
\[ C_1 : \text{Call p} \]
\[ R_1 \]
\[ E_{main} \]

\[ S_p \]
\[ C_2 : \text{Call q} \]
\[ R_2 \]
\[ E_p \]

\[ S_q \]
\[ n_1 : d = a + b \]
\[ n_2 : a = 1 \]
\[ n_3 \]
\[ n_4 \]
\[ C_3 : \text{Call p} \]
\[ C_4 : \text{Call p} \]

Oct 2013
Program Representation for Interprocedural Data Flow Analysis: Supergraph

\[ S_{main} \]
\[ a + b \]
\[ C_1 \text{ Call } p \]
\[ S_p \]
\[ C_2 \text{ Call } q \]
\[ R_1 \]
\[ E_{main} \]

\[ S_p \]
\[ C_2 \]
\[ R_2 \]
\[ E_p \]

\[ n_1 \]
\[ d = a + b \]
\[ C_3 \text{ Call } p \]
\[ R_3 \]
\[ n_3 \]
\[ n_4 \]

\[ S_q \]
\[ a = 1 \]
\[ n_2 \]
\[ C_4 \text{ Call } p \]
\[ R_4 \]

Oct 2013 IIT Bombay
Program Representation for Interprocedural Data Flow Analysis: Supergraph
Program Representation for Interprocedural Data Flow Analysis: Supergraph
Validity of Interprocedural Control Flow Paths

Interprocedurally valid control flow path
Validity of Interprocedural Control Flow Paths

Interprocedurally valid control flow path
Validity of Interprocedural Control Flow Paths

Interprocedurally valid control flow path

Oct 2013
Validity of Interprocedural Control Flow Paths

Interprocedurally invalid control flow path
Validity of Interprocedural Control Flow Paths

Interprocedurally invalid control flow path
Validity of Interprocedural Control Flow Paths

Interprocedurally valid control flow path
Safety, Precision, and Efficiency of Data Flow Analysis

- Data flow analysis uses static representation of programs to compute summary information along paths
Safety, Precision, and Efficiency of Data Flow Analysis

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

- *Ensuring Safety*. All valid paths must be covered
Safety, Precision, and Efficiency of Data Flow Analysis

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

- **Ensuring Safety.** All valid paths must be covered
Safety, Precision, and Efficiency of Data Flow Analysis

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

**Ensuring Safety.** All **valid** paths must be covered.

**Ensuring Precision.** Only valid paths should be covered.
Safety, Precision, and Efficiency of Data Flow Analysis

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

- **Ensuring Safety.** All **valid** paths must be covered

- **Ensuring Precision.** Only valid paths should be covered

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

A path which represents legal control flow
Safety, Precision, and Efficiency of Data Flow Analysis

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

  - **Ensuring Safety.** All valid paths must be covered.

  - **Ensuring Precision.** Only valid paths should be covered.

  - **Ensuring Efficiency.** Only relevant valid paths should be covered.

Subject to merging data flow values at shared program points without creating invalid paths.
Safety, Precision, and Efficiency of Data Flow Analysis

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

- **Ensuring Safety.** All valid paths must be covered.

- **Ensuring Precision.** Only valid paths should be covered.

- **Ensuring Efficiency.** Only relevant valid paths should be covered.

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

A path which represents legal control flow.

A path which yields information that affects the summary information.
Flow and Context Sensitivity

- Flow sensitive analysis:
  Considers *intraprocedurally* valid paths
Flow and Context Sensitivity

- Flow sensitive analysis:
  Considers intraprocedurally valid paths

- Context sensitive analysis:
  Considers interprocedurally valid paths
Flow and Context Sensitivity

- Flow sensitive analysis:
  Considers *intraprocedurally* valid paths

- Context sensitive analysis:
  Considers *interprocedurally* valid paths

- For **maximum statically attainable precision**, analysis must be both flow and context sensitive
Flow and Context Sensitivity

- Flow sensitive analysis:
  Considers intraprocedurally valid paths

- Context sensitive analysis:
  Considers interprocedurally valid paths

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

MFP computation restricted to valid paths only
Context Sensitivity in Interprocedural Analysis

\[ x' = f_r(x) \]
\[ y' = f_r(y) \]
Context Sensitivity in Interprocedural Analysis

\[ S_s \xrightarrow{x} C_i \xrightarrow{x'} R_i \xrightarrow{x''} E_s \]

\[ S_r \xrightarrow{y} C_j \]

\[ S_t \xrightarrow{y'} R_j \xrightarrow{y''} E_t \]
Context Sensitivity in Interprocedural Analysis
Context Sensitivity in Interprocedural Analysis

\[ S_s \xrightarrow{x} C_i \xrightarrow{c_i} R_i \xrightarrow{x'} E_s \]
\[ S_r \xrightarrow{f_r} C_j \xrightarrow{y} R_j \xrightarrow{y'} E_t \]

Oct 2013
Context Sensitivity in Interprocedural Analysis

The diagram illustrates a flow of data and control between different components of a program, highlighting the context sensitivity in interprocedural analysis. The nodes represent contexts, and the edges represent transitions or interactions. The symbols $S_s$, $C_i$, $R_i$, $E_s$, $S_r$, $f_r$, $E_r$, $S_t$, $C_j$, $R_j$, and $E_t$ denote specific contexts or states. The arrows show the flow of data $x$, $y$, and $x'$, illustrating how information is propagated or transformed as the program executes.

The context sensitivity aspect is emphasized by the different paths and labels, indicating how context can influence the analysis of program behavior across procedures or functions.
Staircase Diagrams of Interprocedurally Valid Paths
Staircase Diagrams of Interprocedurally Valid Paths
Staircase Diagrams of Interprocedurally Valid Paths

- "You can descend only as much as you have ascended!"
Staircase Diagrams of Interprocedurally Valid Paths

- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step
Context Sensitivity in Presence of Recursion
Context Sensitivity in Presence of Recursion
Context Sensitivity in Presence of Recursion
Context Sensitivity in Presence of Recursion
Context Sensitivity in Presence of Recursion
Context Sensitivity in Presence of Recursion

\[ u \rightarrow S_p \rightarrow S_i \rightarrow S_q \rightarrow S_j \rightarrow S_k \rightarrow S_r \rightarrow f \rightarrow f' \rightarrow g' \rightarrow E_p \rightarrow E_i \rightarrow E_q \rightarrow E_j \rightarrow E_k \rightarrow E_r \rightarrow v \]
Context Sensitivity in Presence of Recursion

Oct 2013 IIT Bombay
Context Sensitivity in Presence of Recursion

Oct 2013
Context Sensitivity in Presence of Recursion

\[ u \xrightarrow{S_k} f \xrightarrow{S_r} S_p \xrightarrow{S_i} S_q \xrightarrow{S_j} f' \xrightarrow{h} f \xrightarrow{g} g' \xrightarrow{v} E_i \xrightarrow{E_q} E_j \xrightarrow{v} g' \xrightarrow{g} g' \xrightarrow{v} E_p \xrightarrow{E_k} E_r \xrightarrow{v} f' \xrightarrow{f} u \xrightarrow{E_q} E_r \xrightarrow{v} f' \xrightarrow{f} u \xrightarrow{E_q} E_r \xrightarrow{v} f' \xrightarrow{f} u \]
Context Sensitivity in Presence of Recursion
Context Sensitivity in Presence of Recursion
Context Sensitivity in Presence of Recursion

For a path from $u$ to $v$, $g$ must be applied exactly the same number of times as $f$.

For a prefix of the above path, $g$ can be applied only at most as many times as $f$. 

Oct 2013 IIT Bombay
Staircase Diagrams of Interprocedurally Valid Paths
Staircase Diagrams of Interprocedurally Valid Paths
Staircase Diagrams of Interprocedurally Valid Paths
Staircase Diagrams of Interprocedurally Valid Paths

\[
\begin{align*}
C_p & \xrightarrow{C_k} C_r \\
C_i & \xrightarrow{C_j} C_q \\
R_p & \xrightarrow{R_i} R_q \\
R_k & \xrightarrow{R_r} R_j
\end{align*}
\]
Staircase Diagrams of Interprocedurally Valid Paths
Staircase Diagrams of Interprocedurally Valid Paths
Staircase Diagrams of Interprocedurally Valid Paths
Increasing Precision in Data Flow Analysis

Flow insensitive
intraprocedural

Flow sensitive
intraprocedural

Context insensitive
flow insensitive

Context insensitive
flow sensitive

Context sensitive
flow insensitive

Context sensitive
flow sensitive
Increasing Precision in Data Flow Analysis

- Flow insensitive
  - Flow insensitive
  - Context insensitive
    - Context insensitive
      - Context sensitive
      - Context sensitive
      - Context sensitive
  - Context sensitive
    - Context sensitive
    - Context sensitive
    - Context sensitive
- Flow sensitive
  - Flow sensitive
  - Context insensitive
  - Context sensitive
    - Context sensitive
    - Context sensitive
    - Context sensitive
- Context insensitive
  - Context insensitive
  - Context insensitive
  - Context insensitive

actually, only caller sensitive
Part 3

Classical Functional Approach
Functional Approach

\[ x' = f_r(x) \]
Functional Approach

- Compute summary flow functions for each procedure
- Use summary flow functions as the flow function for a call block

\[ x' = f_r(x) \]

Diagram:
- \( S_s \rightarrow C_i \rightarrow R_i \rightarrow E_s \)
- \( S_r \rightarrow f_r \rightarrow E_r \)
Notation for Summary Flow Function

For simplicity forward flow is assumed
Notation for Summary Flow Function

For simplicity forward flow is assumed

- $u_i$: Program points
- $f_i$: Node flow functions
- $\Phi_r(u_i)$: Summary flow functions mapping data flow value from $S_r$ to $u_i$
Notation for Summary Flow Function

For simplicity forward flow is assumed

- $u_i$: Program points
- $f_i$: Node flow functions
- $\Phi_r(u_i)$: Summary flow functions mapping data flow value from $S_r$ to $u_i$
Notation for Summary Flow Function

For simplicity forward flow is assumed

- $u_i$: Program points
- $f_i$: Node flow functions
- $\Phi_r(u_i)$: Summary flow functions mapping data flow value from $S_r$ to $u_i$
Notation for Summary Flow Function

For simplicity forward flow is assumed

- $u_i$: Program points
- $f_i$: Node flow functions
- $\Phi_r(u_i)$: Summary flow functions mapping data flow value from $S_r$ to $u_i$

\[
\Phi_r(u_1) \equiv \phi_{id} \\
\Phi_r(u_2) \equiv f_1 \\
\Phi_r(u_3) \equiv f_1 \\
\Phi_r(u_4) \equiv f_1
\]
Notation for Summary Flow Function

For simplicity forward flow is assumed

- $u_i$: Program points
- $f_i$: Node flow functions
- $\Phi_r(u_i)$: Summary flow functions mapping data flow value from $S_r$ to $u_i$

$$\Phi_r(u_1) \equiv \phi_{id}$$
$$\Phi_r(u_2) \equiv f_1$$
$$\Phi_r(u_3) \equiv f_1$$
$$\Phi_r(u_4) \equiv f_1$$
$$\Phi_r(u_5) \equiv f_2 \circ f_1$$
Notation for Summary Flow Function

For simplicity forward flow is assumed

- $u_i$: Program points
- $f_i$: Node flow functions
- $\Phi_r(u_i)$: Summary flow functions mapping data flow value from $S_r$ to $u_i$

- $\Phi_r(u_1) \equiv \phi_{id}$
- $\Phi_r(u_2) \equiv f_1$
- $\Phi_r(u_3) \equiv f_1$
- $\Phi_r(u_5) \equiv f_2 \circ f_1$
- $\Phi_r(u_6) \equiv f_3 \circ f_1$
- $\Phi_r(u_8)$

Oct 2013
Notation for Summary Flow Function

For simplicity forward flow is assumed

- $u_i$: Program points
- $f_i$: Node flow functions
- $\Phi_r(u_i)$: Summary flow functions mapping data flow value from $S_r$ to $u_i$

$$
\begin{align*}
\Phi_r(u_1) & \equiv \phi_{id} \\
\Phi_r(u_2) & \equiv f_1 \\
\Phi_r(u_3) & \equiv f_1 \\
\Phi_r(u_5) & \equiv f_2 \circ f_1 \\
\Phi_r(u_7) & \equiv f_2 \circ f_1 \sqcap f_3 \circ f_1 \\
\Phi_r(u_6) & \equiv f_3 \circ f_1 \\
\end{align*}
$$
Notation for Summary Flow Function

For simplicity forward flow is assumed

- $u_i$: Program points
- $f_i$: Node flow functions
- $\Phi_r(u_i)$: Summary flow functions mapping data flow value from $S_r$ to $u_i$

![Diagram](attachment:image.png)

- $\Phi_r(u_1) \equiv \phi_{id}$
- $\Phi_r(u_2) \equiv f_1$
- $\Phi_r(u_3) \equiv f_1$
- $\Phi_r(u_5) \equiv f_2 \circ f_1$
- $\Phi_r(u_7) \equiv f_2 \circ f_1 \sqcap f_3 \circ f_1$
- $\Phi_r(u_8) \equiv f_4 \circ (f_2 \circ f_1 \sqcap f_3 \circ f_1)$
Equations for Constructing Summary Flow Functions

For simplicity forward flow is assumed

\[
\Phi_r(Entry(n)) = \begin{cases} 
\phi_{id} & \text{if } n \text{ is } S_r \\
\bigwedge_{p \in \text{pred}(n)} \left( \Phi_r(Exit(p)) \right) & \text{otherwise}
\end{cases}
\]

\[
\Phi_r(Exit(n)) = \begin{cases} 
\Phi_s(u) \circ \Phi_r(Entry(n)) & \text{if } n \text{ calls procedure } s \\
f_n \circ \Phi_r(Entry(n)) & \text{otherwise}
\end{cases}
\]

The summary flow function of a given procedure \( r \)

- is influenced by summary flow functions of the callees of \( r \)
- is not influenced by summary flow functions of the callers of \( r \)

Fixed point computation may be required in the presence of loops or recursion
Constructing Summary Flow Functions Iteratively

\[ r \]

\[ f_1 \]

\[ f_2 \]
Constructing Summary Flow Functions Iteratively

\[ \Phi_r(u_1) = \phi_{id} \]
\[ \Phi_r(u_2) = f_1 \]
\[ \Phi_r(u_3) = f_1 \]
\[ \Phi_r(u_4) = f_2 \circ f_1 \]
Constructing Summary Flow Functions Iteratively

Iteration #2

\[ \Phi_r(u_1) = \psi_{id} \]

\[ \Phi_r(u_2) = f_1 \]

\[ \Phi_r(u_3) = f_1 \cap f_2 \circ f_1 \]

\[ \Phi_r(u_4) = f_2 \circ (f_1 \cap f_2 \circ f_1) \]
Constructing Summary Flow Functions Iteratively

Iteration #3

$r$

\[\Phi_r(u_1) = \phi_{id}\]

\[\Phi_r(u_2) = f_1\]

\[\Phi_r(u_3) = f_1 \sqcap f_2 \circ (f_1 \sqcap f_2 \circ f_1)\]

\[\Phi_r(u_4) = f_2 \circ (f_1 \sqcap f_2 \circ (f_1 \sqcap f_2 \circ f_1))\]

Termination is possible only if all function compositions and confluences can be reduced to a finite set of functions.
Lattice of Flow Functions for Live Variables Analysis

Component functions (i.e. for a single variable)

<table>
<thead>
<tr>
<th>Lattice of data flow values</th>
<th>All possible flow functions</th>
<th>Lattice of flow functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{T} = \emptyset$</td>
<td>$\hat{f}_n(x), \forall x \in {\hat{\top}, \hat{\bot}}$</td>
<td></td>
</tr>
<tr>
<td>$\hat{\bot} = {a}$</td>
<td>$\emptyset$</td>
<td>$\hat{\phi}_\bot$</td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\hat{\phi}_id$</td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>${a}$</td>
<td>$\hat{\top}$</td>
</tr>
<tr>
<td>${a}$</td>
<td>$\emptyset$</td>
<td>$\hat{\bot}$</td>
</tr>
<tr>
<td>${a}$</td>
<td>${a}$</td>
<td>$\hat{\phi}_\bot$</td>
</tr>
</tbody>
</table>

$\hat{\phi}_id$ and $\hat{\phi}_\bot$ are the identity and bottom elements, respectively.
Reducing Component Flow Functions for Live Variables
Analysis

Let $\hat{\phi} \in \{\hat{\phi}_T, \hat{\phi}_{id}, \hat{\phi}_\bot\}$ and $x \in \{1, 0\}$. Then,

- $\hat{\phi}_T \sqcap \hat{\phi} = \hat{\phi}$ (because $0 + x = x$)
- $\hat{\phi}_\bot \sqcap \hat{\phi} = \hat{\phi}_\bot$ (because $1 + x = 1$)
- $\hat{\phi}_T \circ \hat{\phi} = \hat{\phi}_T$ (because $\hat{\phi}_T$ is a constant function)
- $\hat{\phi}_\bot \circ \hat{\phi} = \hat{\phi}_\bot$ (because $\hat{\phi}_\bot$ is a constant function)
- $\hat{\phi}_{id} \circ \hat{\phi} = \hat{\phi}$ (because $\hat{\phi}_{id}$ is the identity function)
Reducing Function Compositions in Bit Vector Frameworks

Kill$_n$ is $UCKill_n$ (denoted $K_n$) and Gen$_n$ is $UCGen_n$ (denoted $G_n$)

$$f_3(x) = f_2(f_1(x))$$
Reducing Function Compositions in Bit Vector Frameworks

Kill$_n$ is \textit{U}CKill$_n$ (denoted $K_n$) and Gen$_n$ is \textit{U}CGen$_n$ (denoted $G_n$)

\[ f_3(x) = f_2(f_1(x)) = f_2((x - K_1) \cup G_1) \]
Reducing Function Compositions in Bit Vector Frameworks

Killₙ is $UC\text{Kill}_n$ (denoted $K_n$) and Genₙ is $UC\text{Gen}_n$ (denoted $G_n$)

\[
\begin{align*}
f_3(x) &= f_2(f_1(x)) \\
       &= f_2((x - K_1) \cup G_1) \\
       &= ((x - K_1) \cup G_1) - K_2 \cup G_2
\end{align*}
\]
Reducing Function Compositions in Bit Vector Frameworks

\( \text{Kill}_n \) is \( \text{UCKill}_n \) (denoted \( K_n \)) and \( \text{Gen}_n \) is \( \text{UCGen}_n \) (denoted \( G_n \))

\[
f_3(x) = f_2(f_1(x)) = f_2((x - K_1) \cup G_1) = ((x - K_1) \cup G_1) - K_2 \cup G_2 = (x - (K_1 \cup K_2)) \cup (G_1 - K_2) \cup G_2
\]
Reducing Function Compositions in Bit Vector Frameworks

Kill\(_n\) is \(UCKill\_n\) (denoted \(K\_n\)) and Gen\(_n\) is \(UCGen\_n\) (denoted \(G\_n\))

\[
f_3(x) = f_2(f_1(x))
= f_2((x - K_1) \cup G_1)
= (((x - K_1) \cup G_1) - K_2) \cup G_2
= (x - (K_1 \cup K_2)) \cup (G_1 - K_2) \cup G_2
\]

Hence,

\[
K_3 = K_1 \cup K_2
G_3 = (G_1 - K_2) \cup G_2
\]
Reducing Bit Vector Flow Function Confluences (1)

Kill\textsubscript{n} is UCKill\textsubscript{n} (denoted \(K_\text{n}\)) and Gen\textsubscript{n} is UCGen\textsubscript{n} (denoted \(G_\text{n}\))

- When \(\cap\) is \(\cup\),

\[
f_3(x) = f_2(x) \cup f_1(x)
\]

\[
= \left((x - K_2) \cup G_2\right) \cup \left((x - K_1) \cup G_1\right)
\]

\[
= \left(x - (K_1 \cap K_2)\right) \cup \left(G_1 \cup G_2\right)
\]

Hence,

\[
K_3 = K_1 \cap K_2
\]

\[
G_3 = G_1 \cup G_2
\]
Reducing Bit Vector Flow Function Confluences (1)

Kill\(_n\) is \textit{UCKill}\(_n\) (denoted \(K_n\)) and Gen\(_n\) is \textit{UCGen}\(_n\) (denoted \(G_n\))

- When \(\sqcap\) is \(\cup\),

\[
\begin{align*}
f_3(x) &= f_2(x) \cup f_1(x) \\
&= ((x - K_2) \cup G_2) \cup ((x - K_1) \cup G_1) \\
&= (x - (K_1 \cap K_2)) \cup (G_1 \cup G_2)
\end{align*}
\]

Hence,

\[
\begin{align*}
K_3 &= K_1 \cap K_2 \\
G_3 &= G_1 \cup G_2
\end{align*}
\]
Reducing Bit Vector Flow Function Confluences (2)

Kill$_n$ is $\text{UCKill}_n$ (denoted $K_n$) and Gen$_n$ is $\text{UCGen}_n$ (denoted $G_n$)

- When $\cap$ is $\cap$,

$$f_3(x) = f_2(x) \cap f_1(x) = ((x - K_2) \cup G_2) \cap ((x - K_1) \cup G_1) = (x - (K_1 \cup K_2)) \cup (G_1 \cap G_2)$$

Hence,

$$K_3 = K_1 \cup K_2$$
$$G_3 = G_1 \cap G_2$$
Kill\textsubscript{n} is \textit{UCKill}_{n} (denoted \textit{K}_{n}) and Gen\textsubscript{n} is \textit{UCGen}_{n} (denoted \textit{G}_{n})

- When \(\sqcap\) is \(\cap\),

\[
\begin{align*}
    f_3(x) &= f_2(x) \cap f_1(x) \\
    &= \left( (x - K_2) \cup G_2 \right) \cap \left( (x - K_1) \cup G_1 \right) \\
    &= (x - (K_1 \cup K_2)) \cup (G_1 \cap G_2)
\end{align*}
\]

Hence,

\[
\begin{align*}
    K_3 &= K_1 \cup K_2 \\
    G_3 &= G_1 \cap G_2
\end{align*}
\]
An Example of Interprocedural Liveness Analysis

\[ S_{main} \]

\[ a = 5; \quad b = 3 \]
\[ c = 7; \quad \text{read } d \]

\[ c_1 \]

\[ \text{Call p} \]

\[ n_1 \]

\[ a = a + 2 \]
\[ e = c + d \]

\[ n_2 \]

\[ d = a \times b \]

\[ c_2 \]

\[ \text{Call q} \]

\[ \text{print } a + c + e \]

\[ S_p \]

\[ b = 2 \]
\[ \text{if } (b < d) \]

\[ n_3 \]

\[ c = a + b \]

\[ c_3 \]

\[ \text{Call p} \]

\[ \text{T} \]

\[ \text{F} \]

\[ \text{print } c + d \]

\[ c_4 \]

\[ \text{Call q} \]

\[ \text{Eq} \]

\[ a = a \times b \]

\[ \text{Eq} \]

\[ a = a \times b \]
Summary Flow Functions for Interprocedural Liveness Analysis

<table>
<thead>
<tr>
<th>Proc</th>
<th>Flow Function</th>
<th>Defining Expression</th>
<th>Iteration #1</th>
<th>Changes in iteration #2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Gen</td>
<td>Kill</td>
</tr>
<tr>
<td>$p$</td>
<td>$\Phi_p(E_p)$</td>
<td>$f_{E_p}$</td>
<td>${c, d}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td></td>
<td>$\Phi_p(n_3)$</td>
<td>$f_{n_3} \circ \Phi_p(E_p)$</td>
<td>${a, b, d}$</td>
<td>${c}$</td>
</tr>
<tr>
<td></td>
<td>$\Phi_p(c_4)$</td>
<td>$f_q \circ \Phi_p(E_p) = \phi_T$</td>
<td>$\emptyset$</td>
<td>${a, b, c, d, e}$</td>
</tr>
<tr>
<td></td>
<td>$\Phi_p(S_p)$</td>
<td>$f_{S_p} \circ (\Phi_p(n_3) \sqcap \Phi_p(c_4))$</td>
<td>${a, d}$</td>
<td>${b, c}$</td>
</tr>
<tr>
<td></td>
<td>$f_p$</td>
<td>$\Phi_p(S_p)$</td>
<td>${a, d}$</td>
<td>${b, c}$</td>
</tr>
<tr>
<td>$q$</td>
<td>$\Phi_q(E_q)$</td>
<td>$f_{E_q}$</td>
<td>${a, b}$</td>
<td>${a}$</td>
</tr>
<tr>
<td></td>
<td>$\Phi_q(c_3)$</td>
<td>$f_p \circ \Phi_q(E_q)$</td>
<td>${a, d}$</td>
<td>${a, b, c}$</td>
</tr>
<tr>
<td></td>
<td>$\Phi_q(S_q)$</td>
<td>$f_{S_q} \circ \Phi_q(c_3)$</td>
<td>${d}$</td>
<td>${a, b, c}$</td>
</tr>
<tr>
<td></td>
<td>$f_q$</td>
<td>$\Phi_q(S_q)$</td>
<td>${d}$</td>
<td>${a, b, c}$</td>
</tr>
</tbody>
</table>
Computed Summary Flow Functions

\[
\begin{align*}
S_p & \quad b = 2 \\
& \quad \text{if } (b < d) \\
& \quad c = a + b \\
E_p & \quad \text{print } c + d
\end{align*}
\]

\[
\begin{align*}
n_3 & \quad c = a + b \\
c_4 & \quad \text{Call } q
\end{align*}
\]

\[
\begin{align*}
S_q & \quad a = 1 \\
c_3 & \quad \text{Call } p
\end{align*}
\]

\[
\begin{align*}
E_q & \quad a = a \times b
\end{align*}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Summary Flow Function} & \\
\Phi_p(E_p) & BI_p \cup \{c, d\} \\
\Phi_p(n_3) & (BI_p - \{c\}) \cup \{a, b, d\} \\
\Phi_p(c_4) & (BI_p - \{a, b, c\}) \cup \{d\} \\
\Phi_p(S_p) & (BI_p - \{b, c\}) \cup \{a, d\} \\
\Phi_q(E_q) & (BI_q - \{a\}) \cup \{a, b\} \\
\Phi_q(c_3) & (BI_q - \{a, b, c\}) \cup \{a, d\} \\
\Phi_q(S_q) & (BI_q - \{a, b, c\}) \cup \{d\} \\
\hline
\end{array}
\]
Result of Interprocedural Liveness Analysis

<table>
<thead>
<tr>
<th>Data flow variable</th>
<th>Summary flow function</th>
<th>Data flow value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In</strong> (E_m)</td>
<td>( \Phi_m(E_m) )</td>
<td>( BL_m \cup {a, c, e} )</td>
</tr>
<tr>
<td><strong>In</strong> (c_2)</td>
<td>( \Phi_m(c_2) )</td>
<td>( (BL_m - {a, b, c}) \cup {d, e} )</td>
</tr>
<tr>
<td><strong>In</strong> (n_2)</td>
<td>( \Phi_m(n_2) )</td>
<td>( (BL_m - {a, b, c, d}) \cup {a, b, e} )</td>
</tr>
<tr>
<td><strong>In</strong> (n_1)</td>
<td>( \Phi_m(n_1) )</td>
<td>( (BL_m - {a, b, c, d, e}) \cup {a, b, c, d} )</td>
</tr>
<tr>
<td><strong>In</strong> (c_1)</td>
<td>( \Phi_m(c_1) )</td>
<td>( (BL_m - {a, b, c, d, e}) \cup {a, d} )</td>
</tr>
<tr>
<td><strong>In</strong> (S_m)</td>
<td>( \Phi_m(S_m) )</td>
<td>( BL_m - {a, b, c, d, e} )</td>
</tr>
</tbody>
</table>

Procedure main, \( BL = \emptyset \)
### Result of Interprocedural Liveness Analysis

<table>
<thead>
<tr>
<th>Data flow variable</th>
<th>Summary flow function</th>
<th>Data flow value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Procedure $p$, $BI = {a, b, c, d, e}$</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ln_{E_p}$</td>
<td>$\Phi_p(E_p)$</td>
<td>$BI_p \cup {c, d}$</td>
</tr>
<tr>
<td>$ln_{n_3}$</td>
<td>$\Phi_p(n_3)$</td>
<td>$(BI_p - {c}) \cup {a, b, d}$</td>
</tr>
<tr>
<td>$ln_{c_4}$</td>
<td>$\Phi_p(c_4)$</td>
<td>$(BI_p - {a, b, c}) \cup {d}$</td>
</tr>
<tr>
<td>$ln_{S_p}$</td>
<td>$\Phi_p(S_p)$</td>
<td>$(BI_p - {b, c}) \cup {a, d}$</td>
</tr>
<tr>
<td><strong>Procedure $q$, $BI = {a, b, c, d, e}$</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ln_{E_q}$</td>
<td>$\Phi_q(E_q)$</td>
<td>$(BI_q - {a}) \cup {a, b}$</td>
</tr>
<tr>
<td>$ln_{c_3}$</td>
<td>$\Phi_q(c_3)$</td>
<td>$(BI_q - {a, b, c}) \cup {a, d}$</td>
</tr>
<tr>
<td>$ln_{S_q}$</td>
<td>$\Phi_q(S_q)$</td>
<td>$(BI_q - {a, b, c}) \cup {d}$</td>
</tr>
</tbody>
</table>
Context Sensitivity of Interprocedural Liveness Analysis

\[
S_{\text{main}}
\]

\[
\begin{align*}
& a = 5; b = 3 \\
& c = 7; \text{read } d
\end{align*}
\]

\[
\begin{align*}
S_p & : b = 2 \\
& \text{if } (b < d)
\end{align*}
\]

\[
\begin{align*}
S_q & : a = 1
\end{align*}
\]

\[
\begin{align*}
& n_1 \\
& \{a, b, c, d\}
\end{align*}
\]

\[
\begin{align*}
& c_1 \quad \text{Call } p \\
& \{a, d\}
\end{align*}
\]

\[
\begin{align*}
& c_2 \quad \text{Call } q \\
& \{a, c, e\}
\end{align*}
\]

\[
\begin{align*}
& n_2 \quad d = a \times b \\
& \{a, b, e\}
\end{align*}
\]

\[
\begin{align*}
& n_3 \quad c = a + b
\end{align*}
\]

\[
\begin{align*}
& n_4 \quad \text{Call } q
\end{align*}
\]

\[
\begin{align*}
& c_3 \quad \text{Call } p
\end{align*}
\]

\[
\begin{align*}
& E_{\text{main}} \quad \text{print } a + c + e
\end{align*}
\]

\[
\begin{align*}
& E_p \quad \text{print } c + d
\end{align*}
\]

\[
\begin{align*}
& E_q \quad a = a \times b
\end{align*}
\]

\[
\begin{align*}
& E_{\text{main}} \quad \{a, d, e\}
\end{align*}
\]

\[
\begin{align*}
& E_p \quad \{a, b, d, e\}
\end{align*}
\]

\[
\begin{align*}
& E_q \quad \{a, b, c, d, e\}
\end{align*}
\]

\[
\begin{align*}
& E_{\text{main}} \quad \{a, d, e\}
\end{align*}
\]
Context Sensitivity of Interprocedural Liveness Analysis

\[ S_{main} \]
\[
\begin{align*}
    a &= 5; b = 3 \\
    c &= 7; \text{read } d
\end{align*}
\]

\[ S_p \]
\[
\begin{align*}
    b &= 2 \\
    \text{if } (b < d) \\
    c &= a + b
\end{align*}
\]

\[ S_q \]
\[
\begin{align*}
    a &= 1 \\
    \text{Call } p
\end{align*}
\]

\[ E_{main} \]
\[
\text{print } a + c + e
\]

\[ E_q \]
\[
\begin{align*}
    a &= a \ast b
\end{align*}
\]
Tutorial Problem #1

Perform interprocedural live variables analysis for the following program:

```c
main()
{
    p();
}

p()
{
    while (c < 10)
    {
        p();
        a = a*b;
    }
}
```
Tutorial Problem #2: Summary Flow Function for Constant Propagation

\[
b = c + d
\]

\[
a = a + b
\]

\[
C_1 \quad Call \ p
\]

\[
a = a - b
\]

\[
S_p
\]

\[
n_1
\]

\[
n_2
\]

\[
E_p
\]
Tutorial Problem #2: Summary Flow Function for Constant Propagation

\[
\begin{align*}
S_p & \\
n_1 & \quad a = a + b \\
C_1 & \quad Call \ p \\
n_2 & \quad a = a - b \\
E_p &
\end{align*}
\]

<table>
<thead>
<tr>
<th>( \Phi_p(S_p) )</th>
<th>( \Phi_p(n_1) )</th>
<th>( \Phi_p(C_1) )</th>
<th>( \Phi_p(n_2) )</th>
<th>( \Phi_p(E_p) )</th>
<th>( f_p )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle v_a, v_b \rangle )</td>
<td>( \langle v_a + v_b, v_b \rangle )</td>
<td>( \langle \top, \top \rangle )</td>
<td>( \langle \top, \top \rangle )</td>
<td>( \langle v_a, v_b \rangle )</td>
<td>( \langle v_a, v_b \rangle )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iter. #1</th>
<th>Iter. #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle v_a, v_b \rangle )</td>
<td>( \langle v_a, v_b \rangle )</td>
</tr>
<tr>
<td>( \langle v_a + v_b, v_b \rangle )</td>
<td>( \langle v_a + v_b, v_b \rangle )</td>
</tr>
<tr>
<td>( \langle \top, \top \rangle )</td>
<td>( \langle \top, \top \rangle )</td>
</tr>
<tr>
<td>( \langle v_a, v_b \rangle )</td>
<td>( \langle v_a, v_b \rangle )</td>
</tr>
<tr>
<td>( \langle v_a, v_b \rangle )</td>
<td>( \langle v_a, v_b \rangle )</td>
</tr>
</tbody>
</table>
Tutorial Problem #2: Summary Flow Function for Constant Propagation

Will this work always?
Tutorial Problem #3

- Is \(a \times b\) available on line 18? Line 6?
- Perform available expressions analysis by constructing the summary flow function for procedure \(p\)

```
1. main()
2. {
3.     c = a*b;
4.     p();
5.     a = a*b;
6. }
7. p()
8. {   if (...) 
9.     {   a = a*b;
10.     p();
11.   }
12.   else if (...) 
13.     {   c = a * b;
14.     p();
15.     c = a;
16.   }
17.   else 
18.       ; /* ignore */
19.   }
```
Limitations of Functional Approach to Interprocedural Data Flow Analysis

Problems with constructing summary flow functions
Limitations of Functional Approach to Interprocedural Data Flow Analysis

Problems with constructing summary flow functions

- Reducing expressions defining flow functions may not be possible in the presence of dependent parts
- May work for some instances of some problems but not for all
- Hence basic blocks in pointer analysis and constant propagation contain a single statement
Overall Flow Function and Component Function

- Overall flow function $f : L \mapsto L$ is $\langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle$
- Component function: $\hat{h}_i$ which computes the value of $\hat{x}_i$
Overall Flow Function and Component Function

- Overall flow function \( f : L \mapsto L \) is \( \langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle \)

- Component function: \( \hat{h}_i \) which computes the value of \( \hat{x}_i \)

Separable

General Non-Separable
Overall Flow Function and Component Function

- Overall flow function $f : L \mapsto L$ is $\langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle$
- Component function: $\hat{h}_i$ which computes the value of $\hat{x}_i$

**Separable**

$$\langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \xrightarrow{f} \langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle$$

**General Non-Separable**

$$\langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \xrightarrow{f} \langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle$$
Overall Flow Function and Component Function

- Overall flow function \( f : L \mapsto L \) is \( \langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle \)

- Component function: \( \hat{h}_i \) which computes the value of \( \hat{x}_i \)

**Separable**

\[
\langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \\
\hat{h}_2 \\
\langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle
\]

**General Non-Separable**

\[
\langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \\
f \\
\langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle
\]

Oct 2013 IIT Bombay
Overall Flow Function and Component Function

- Overall flow function $f : L \mapsto L$ is $\langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle$
- Component function: $\hat{h}_i$ which computes the value of $\hat{x}_i$

Separable

\[
\langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle
\]

\[
\hat{h}_2
\]

\[
\langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle
\]

\[
\hat{h} : \hat{L} \mapsto \hat{L}
\]

General Non-Separable

\[
\langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle
\]

\[
f
\]

\[
\langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle
\]
Overall Flow Function and Component Function

- Overall flow function $f : L \mapsto L$ is $\langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle$
- Component function: $\hat{h}_i$ which computes the value of $\hat{x}_i$

Separable

\[
\langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle
\]

\[
\hat{h}_2
\]

\[
\langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle
\]

General Non-Separable

\[
\langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle
\]

\[
\hat{h}_2
\]

\[
\langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle
\]

$\hat{h} : \hat{L} \mapsto \hat{L}$
Overall Flow Function and Component Function

- Overall flow function \( f : L \mapsto \hat{L} \) is \( \langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle \)
- Component function: \( \hat{h}_i \) which computes the value of \( \hat{x}_i \)

Separable

\[
\langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle
\]

\[
\langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle
\]

\[
\hat{h} : \hat{L} \mapsto \hat{L}
\]

Example: All bit vector frameworks

General Non-Separable

\[
\langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle
\]

\[
\langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle
\]

\[
\hat{h} : L \mapsto \hat{L}
\]

Example: Points-To Analysis
Component Functions and Primitive Entity Functions

Component functions consist of primitive entity functions

\[
\langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle
\]

\[
\langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle
\]

\[
\hat{h}_2
\]

\[
\langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \
\theta^1_2
\]

\[
\langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle
\]
Component Functions and Primitive Entity Functions

Component functions consist of primitive entity functions

\[ \langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \]
\[ \langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle \]

\[ \hat{h}_2 \]

\[ \langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \]
\[ \langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle \]

\[ \hat{h}_i(x) = \prod_{j=1}^{j \leq m} \theta^j_i \]
### Entity Functions in Points-to Analysis

<table>
<thead>
<tr>
<th>Statement with $a \in L_locations$</th>
<th>Entity functions</th>
<th>Closed under composition?</th>
</tr>
</thead>
<tbody>
<tr>
<td>... = null</td>
<td>Constant</td>
<td>$\hat{L} \mapsto \hat{L}$</td>
</tr>
<tr>
<td>... = &amp;b</td>
<td>Constant</td>
<td>$\hat{L} \mapsto \hat{L}$</td>
</tr>
<tr>
<td>... = b</td>
<td>Identity</td>
<td>$\hat{L} \mapsto \hat{L}$</td>
</tr>
<tr>
<td>... = *b</td>
<td>?</td>
<td>$L \mapsto \hat{L}$</td>
</tr>
</tbody>
</table>
## Entity Functions in Constant Propagation

<table>
<thead>
<tr>
<th>Statement</th>
<th>Entity functions</th>
<th>Closed under composition?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = 5$</td>
<td>Constant</td>
<td>$\hat{L} \mapsto \hat{L}$</td>
</tr>
<tr>
<td>$a = b$</td>
<td>Constant</td>
<td>$\hat{L} \mapsto \hat{L}$</td>
</tr>
<tr>
<td>$a = b + 5$</td>
<td>Linear</td>
<td>$\hat{L} \mapsto \hat{L}$</td>
</tr>
<tr>
<td>$a = b + c$</td>
<td>?</td>
<td>$L \mapsto \hat{L}$</td>
</tr>
</tbody>
</table>
Enumeration Based Functional Approach

- Instead of constructing flow functions, remember the mapping $x \mapsto y$ as input output values.
- Reuse output value of a flow function when the same input value is encountered again.
Enumeration Based Functional Approach

- Instead of constructing flow functions, remember the mapping $x \mapsto y$ as input output values
- Reuse output value of a flow function when the same input value is encountered again

Requires the number of values to be finite
Part 4

Classical Call Strings Approach
Classical Full Call Strings Approach

Most general, flow and context sensitive method

- Remember call history
  Information should be propagated back to the correct point

- Call string at a program point:
  - Sequence of unfinished calls reaching that point
  - Starting from the $S_{\text{main}}$

A snap-shot of call stack in terms of call sites
Interprocedural Validity and Calling Contexts
Interprocedural Validity and Calling Contexts

Oct 2013
Interprocedural Validity and Calling Contexts

• “You can descend only as much as you have ascended!”
“You can descend only as much as you have ascended!”

Every descending step must match a corresponding ascending step
Interprocedural Validity and Calling Contexts

- “You can descend only as much as you have ascended!”
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps
Interprocedural Validity and Calling Contexts

• “You can descend only as much as you have ascended!”
• Every descending step must match a corresponding ascending step
• Calling context is represented by the remaining descending steps
Interprocedural Validity and Calling Contexts

• “You can descend only as much as you have ascended!”
• Every descending step must match a corresponding ascending step
• Calling context is represented by the remaining descending steps
• “You can descend only as much as you have ascended!”
• Every descending step must match a corresponding ascending step
• Calling context is represented by the remaining descending steps
Interprocedural Validity and Calling Contexts

- “You can descend only as much as you have ascended!”
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps
Interprocedural Validity and Calling Contexts

- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps
Interprocedural Validity and Calling Contexts

- “You can descend only as much as you have ascended!”
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps
Interprocedural Validity and Calling Contexts

- "You can descend only as much as you have ascended!"
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps
Interprocedural Validity and Calling Contexts

- “You can descend only as much as you have ascended!”
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps
“You can descend only as much as you have ascended!”

• Every descending step must match a corresponding ascending step

• Calling context is represented by the remaining descending steps
“You can descend only as much as you have ascended!”

Every descending step must match a corresponding ascending step

Calling context is represented by the remaining descending steps
Interprocedural Validity and Calling Contexts

- “You can descend only as much as you have ascended!”
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps
Interprocedural Validity and Calling Contexts

- “You can descend only as much as you have ascended!”
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps
Interprocedural Validity and Calling Contexts

• “You can descend only as much as you have ascended!”
• Every descending step must match a corresponding ascending step
• Calling context is represented by the remaining descending steps

Oct 2013 IIT Bombay
Interprocedural Validity and Calling Contexts

• “You can descend only as much as you have ascended!”
• Every descending step must match a corresponding ascending step
• Calling context is represented by the remaining descending steps
Interprocedural Validity and Calling Contexts

- “You can descend only as much as you have ascended!”
- Every descending step must match a corresponding ascending step
- Calling context is represented by the remaining descending steps
Interprocedural Data Flow Analysis Using Call Strings

- Augmented data flow information
  - $IN_n$ and $OUT_n$ are partial maps from call strings to $L$
  - The final data flow information at a program point is
    \[
    \text{In}_n = \bigwedge_{(\sigma, x) \in IN_n} x
    \]
    \[
    \text{Out}_n = \bigwedge_{(\sigma, x) \in OUT_n} x
    \]
    (glb of data flow values for all call strings)

- Flow functions to manipulate tagged data flow information
  - Intraprocedural edges manipulate data flow value $x$
  - Interprocedural edges manipulate call string $\sigma$
Augmented Data Flow Equations: Computing $\text{IN}_n$

$$
\text{IN}_n = \begin{cases} 
\langle \lambda, BI \rangle & \text{if } n \text{ is a } S_{\text{main}} \\
\biguplus_{p \in \text{pred}(n)} \text{OUT}_p & \text{otherwise}
\end{cases}
$$

where we merge underlying data flow values only if the contexts are same
Augmented Data Flow Equations: Computing $\text{IN}_n$

\[
\text{IN}_n = \begin{cases} 
\langle \lambda, BI \rangle & \text{if } n \text{ is a } S_{main} \\
\biguplus_{p \in \text{pred}(n)} \text{OUT}_p & \text{otherwise}
\end{cases}
\]

where we merge underlying data flow values only if the contexts are same

\[
\text{OUT} \cup \text{IN} = \left\{ \langle \sigma, z \rangle \mid \langle \sigma, x \rangle \in \text{OUT} \land \langle \sigma, y \rangle \in \text{IN} \Rightarrow z = x \sqcap y, \langle \sigma, x \rangle \in \text{OUT} \land \langle \sigma, y \rangle \notin \text{IN} \Rightarrow z = x, \langle \sigma, x \rangle \notin \text{OUT} \land \langle \sigma, y \rangle \in \text{IN} \Rightarrow z = y \right\}
\]
Augmented Data Flow Equations: Computing $\text{OUT}_n$

- Call node $C_i$
  - Append $c_i$ to every $\sigma$
  - Propagate the data flow values unchanged
Augmented Data Flow Equations: Computing $\text{OUT}_n$

- Call node $C_i$
  - Append $c_i$ to every $\sigma$
  - Propagate the data flow values unchanged

- Return node $R_i$
  - If the last call site is $c_i$, remove it and propagate the data flow value unchanged
  - Block other data flow values
Augmented Data Flow Equations: Computing $\text{OUT}_n$

- Call node $C_i$
  - Append $c_i$ to every $\sigma$
  - Propagate the data flow values unchanged

- Return node $R_i$
  - If the last call site is $c_i$, remove it and propagate the data flow value unchanged
  - Block other data flow values
Augmented Data Flow Equations: Computing $\text{OUT}_n$

- Call node $C_i$
  - Append $c_i$ to every $\sigma$
  - Propagate the data flow values unchanged

- Return node $R_i$
  - If the last call site is $c_i$, remove it and propagate the data flow value unchanged
  - Block other data flow values

\[
\text{OUT}_n(X) = \begin{cases} 
\{ \langle \sigma \cdot c_i, x \rangle \mid \langle \sigma, x \rangle \in \text{IN}_n \} & \text{if } n \text{ is } C_i \\
\{ \langle \sigma, x \rangle \mid \langle \sigma \cdot c_i, x \rangle \in \text{IN}_n \} & \text{if } n \text{ is } R_i \\
\{ \langle \sigma, f_n(x) \rangle \mid \langle \sigma, x \rangle \in \text{IN}_n \} & \text{otherwise}
\end{cases}
\]
Available Expressions Analysis Using Call Strings Approach

\[ S_{\text{main}} \]
- read \( a, b \)
- \( t := a \times b \)

\[ C_1 \]
call \( p \)

\[ R_1 \]

\[ n_1 \]
print \( a \times b \)

\[ E_{\text{main}} \]

\[ S_p \]
if \( a == 0 \)

\[ n_2 \]
a = a - 1

\[ C_2 \]
call \( p \)

\[ R_2 \]

\[ n_3 \]
t = a \times b

\[ E_p \]
Available Expressions Analysis Using Call Strings Approach

\[ S_{main} \]
- read \( a, b \)
- \( t := a \times b \)

\[ C_1 \]
- call \( p \)

\[ R_1 \]
- \( n_1 \) print \( a \times b \)

\[ n_1 \]
- Is \( a \times b \) available?

\[ E_{main} \]

\[ S_p \]
- if \( a == 0 \)

\[ n_2 \]
- \( a = a - 1 \)

\[ C_2 \]
- call \( p \)

\[ R_2 \]
- \( n_3 \) \( t = a \times b \)

\[ E_p \]
Available Expressions Analysis Using Call Strings Approach

\[
\begin{align*}
S_{main} & \quad \text{read } a, b \\
& \quad t := a \times b \\
C_1 & \quad \text{call } p \\
R_1 & \quad \text{Is } a \times b \text{ available?} \\
E_{main} & \\
S_p & \quad \text{if } a == 0 \\
& \quad a = a - 1 \\
C_2 & \quad \text{call } p \\
R_2 & \\
E_p & \\
n_1 & \quad \text{print } a \times b \\
n_2 & \quad a = a - 1 \\
n_3 & \quad t = a \times b
\end{align*}
\]

```c
int a, b, t;
void p()
{
    if (a == 0)
    {
        a = a - 1;
        p();
        t = a * b;
    }
}
```
Available Expressions Analysis Using Call Strings Approach

```
int a, b, t;
void p()
{
    if (a == 0)
    {
        a = a - 1;
        p();
        t = a * b;
    }
}

Is a * b available?
Yes!
```
Available Expressions Analysis Using Call Strings Approach

\[ S_{main} \]  
read \( a, b \)  
t := \( a \times b \)  

\[ C_1 \]  
call \( p \)  

\[ R_1 \]  

\[ n_1 \]  
print \( a \times b \)  

\[ E_{main} \]  

\[ S_p \]  
if \( a == 0 \)  

\[ n_2 \]  
a = a - 1  
Kill  

\[ C_2 \]  
call \( p \)  

\[ R_2 \]  

\[ n_3 \]  
t = \( a \times b \)  

\[ E_p \]
Available Expressions Analysis Using Call Strings Approach

\[ S_{main} \]
- read \( a, b \)
- \( t := a \ast b \)

\[ C_1 \]
call \( p \)

\[ R_1 \]

\[ n_1 \]
print \( a \ast b \)

\[ E_{main} \]

\[ E_p \]

\[ S_p \]
if \( a == 0 \)

\[ n_2 \]
a = a - 1

\[ C_2 \]
call \( p \)

\[ R_2 \]

\[ n_3 \]
t = a \ast b

Kill

Oct 2013 IIT Bombay
Available Expressions Analysis Using Call Strings Approach

\[ S_{\text{main}} \]

\[ \text{read } a, b \]
\[ t := a \times b \]

\[ C_1 \]
\[ \text{call } p \]

\[ R_1 \]
\[ n_1 \]
\[ \text{print } a \times b \]

\[ E_{\text{main}} \]

\[ S_p \]
\[ \text{if } a == 0 \]

\[ n_2 \]
\[ a = a - 1 \]

\[ C_2 \]
\[ \text{call } p \]

\[ R_2 \]
\[ n_3 \]
\[ t = a \times b \]

\[ E_p \]

Kill

Oct 2013
Available Expressions Analysis Using Call Strings Approach

\[ S_{main} \]
- read \( a, b \)
- \( t := a \times b \)

\[ C_1 \]
- call \( p \)

\[ R_1 \]

\[ n_1 \]
- print \( a \times b \)

\[ E_{main} \]

\[ S_p \]
- if \( a == 0 \)

\[ n_2 \]
- \( a = a - 1 \)

\[ C_2 \]
- call \( p \)

\[ R_2 \]

\[ n_3 \]
- \( t = a \times b \)

\[ E_p \]

\[ S_p \]
- \( a == 0 \)

\[ n_2 \]
- \( a = a - 1 \)

\[ C_2 \]
- call \( p \)

\[ R_2 \]

\[ n_3 \]
- \( t = a \times b \)

\[ E_p \]
Available Expressions Analysis Using Call Strings Approach

\[ S_{main} \]
- read \( a, b \)
- \( t := a \times b \)

\[ C_1 \]
- call \( p \)

\[ R_1 \]

\[ n_1 \]
- print \( a \times b \)

\[ E_{main} \]

\[ S_p \]
- if \( a == 0 \)

\[ n_2 \]
- \( a = a - 1 \)

\[ C_2 \]
- call \( p \)

\[ R_2 \]

\[ n_3 \]
- \( t = a \times b \)

\[ E_p \]

Kill

Gen
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

\[ S_{main} \]

read \( a, b \)
\[ t := a \times b \]

\[ C_1 \]
call \( p \)

\[ R_1 \]

\[ n_1 \]
print \( a \times b \)

\[ E_{main} \]

\[ S_p \]
if \( a == 0 \)

\[ n_2 \]
\[ a = a - 1 \]

\[ C_2 \]
call \( p \)

\[ R_2 \]

\[ n_3 \]
\[ t = a \times b \]

\[ E_p \]
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

\[ S_{main} \]
- **read** a, b
- **t := a * b**

\[ \langle \lambda, 1 \rangle \]
- **call p**

\[ C_1 \]

\[ R_1 \]
- **print a * b**

\[ n_1 \]
- **print a * b**

\[ E_{main} \]

\[ S_p \]
- **if a == 0**

\[ n_2 \]
- **a = a - 1**

\[ C_2 \]
- **call p**

\[ R_2 \]
- **print a * b**

\[ n_3 \]
- **t = a * b**

\[ E_p \]

Oct 2013
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

\( S_{main} \)
- \( \text{read } a, b \)
- \( t := a \ast b \)

\( C_1 \)
- \( \text{call } p \)

\( n_1 \)
- \( \text{print } a \ast b \)

\( E_{main} \)

\( S_p \)
- if \( a == 0 \)

\( n_2 \)
- \( a = a - 1 \)

\( C_2 \)
- \( \text{call } p \)

\( n_3 \)
- \( t = a \ast b \)

\( E_p \)

\( \langle c_1, 1 \rangle \)
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

\[ S_{main} \]
\[ \text{read } a, b \]
\[ t := a \ast b \]
\[ \langle \lambda, 1 \rangle \]
\[ C_1 \]
\[ \text{call } p \]

\[ R_1 \]
\[ n_1 \]
\[ \text{print } a \ast b \]

\[ E_{main} \]

\[ S_p \]
\[ \text{if } a == 0 \]

\[ n_2 \]
\[ a = a - 1 \]

\[ C_2 \]
\[ \text{call } p \]

\[ R_2 \]
\[ n_3 \]
\[ t = a \ast b \]

\[ E_p \]
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

\[ S_{\text{main}} \]

- \( \text{read } a, b \)
- \( t := a \times b \)

\[ C_1 \]

- call \( p \)

\[ R_1 \]

\[ n_1 \]

- print \( a \times b \)

\[ E_{\text{main}} \]

\[ \langle c_1, 1 \rangle \]

\[ S_p \]

if \( a == 0 \)

\[ n_2 \]

- \( a = a - 1 \)

\[ C_2 \]

- call \( p \)

\[ R_2 \]

\[ n_3 \]

- \( t = a \times b \)

\[ E_p \]
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

\[ S_{main}: \text{read } a, b \]
\[ t := a \times b \]

\[ C_1: \text{call } p \]

\[ R_1 \]

\[ n_1: \text{print } a \times b \]

\[ E_{main} \]

\[ S_p \]

\[ \langle c_1, 1 \rangle \]

\[ \langle c_1 c_2, 0 \rangle \]

\[ \text{if } a == 0 \]

\[ n_2: a = a - 1 \]

\[ C_2: \text{call } p \]

\[ R_2 \]

\[ n_3: t = a \times b \]

\[ E_p \]
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

\( S_{\text{main}} \)  
\( \text{read } a, b \)  
\( t := a \times b \)

\( C_1 \)  
\( \text{call } p \)

\( R_1 \)

\( n_1 \)  
\( \text{print } a \times b \)

\( E_{\text{main}} \)

\( \langle c_1, 1 \rangle \)

\( S_p \)  
\( \text{if } a == 0 \)

\( n_2 \)  
\( a = a - 1 \)

\( \langle c_1, 0 \rangle \)

\( \langle c_1 c_2, 0 \rangle \)

\( C_2 \)  
\( \text{call } p \)

\( R_2 \)

\( n_3 \)  
\( t = a \times b \)

\( \langle c_1 c_2, 0 \rangle \)

\( E_p \)  
\( a \times b \)
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

\[ S_{main} \]
- read \( a, b \)
- \( t := a \times b \)

\[ C_1 \]
- call \( p \)

\[ R_1 \]
- \( n_1 \) print \( a \times b \)

\[ E_{main} \]

\[ S_p \]
- if \( a == 0 \)

\[ n_2 \]
- \( a = a - 1 \)

\[ C_2 \]
- call \( p \)

\[ R_2 \]
- \( n_3 \) \( t = a \times b \)

\[ E_p \]
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

\[ S_{main} \]
- read \( a, b \)
- \( t := a \times b \)

\[ C_1 \]
call \( p \)

\[ R_1 \]

\[ n_1 \]
print \( a \times b \)

\[ E_{main} \]

\[ S_p \] if \( a == 0 \)

\[ n_2 \]
- \( a = a - 1 \)

\[ C_2 \]
call \( p \)

\[ R_2 \]
- \( \langle c_1, 1 \rangle \)
- \( \langle c_1 c_2, 0 \rangle \)
- \( \langle c_1 c_2 c_2, 0 \rangle \)

\[ n_3 \]
- \( t = a \times b \)
- \( \langle c_1, 1 \rangle \)
- \( \langle c_1 c_2, 0 \rangle \)
- \( \langle c_1 c_2 c_2, 0 \rangle \)

\[ E_p \]
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

\[ S_{main} \]
- \( \text{read } a, b \)
- \( t := a \times b \)
- \( \langle \lambda, 1 \rangle \)

\[ C_1 \]
- \( \text{call } p \)

\[ R_1 \]

\[ n_1 \]
- \( \text{print } a \times b \)

\[ E_{main} \]

\[ S_p \]
- if \( a == 0 \)
- \( \langle c_1, 1 \rangle \)
- \( \langle c_1 c_2, 0 \rangle, \langle c_1 c_2 c_2, 0 \rangle, \ldots \)

\[ n_2 \]
- \( a = a - 1 \)
- \( \langle c_1, 0 \rangle, \langle c_1 c_2, 0 \rangle, \ldots \)

\[ C_2 \]
- \( \text{call } p \)

\[ R_2 \]

\[ n_3 \]
- \( t = a \times b \)
- \( \langle c_1, 0 \rangle \)
- \( \langle c_1 c_2, 0 \rangle \)
- \( \langle c_1 c_2 c_2, 0 \rangle \)
- \( \ldots \)

\[ E_p \]
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

$S_{main}$

- `read a, b`
- `t := a * b`

$C_1$
- `call p`

$E_{main}$

$R_1$

- `$\langle \lambda, 1 \rangle$`

$n_1$
- `print a * b`

$S_p$

- `if a == 0`
- `n_2$
  - `a = a - 1`

$C_2$
- `call p`

$R_2$

- `$\langle c_1, 0 \rangle$, $\langle c_1 c_2, 0 \rangle$, . . .`

$n_3$
- `t = a * b`

$E_p$
- `$\langle c_1, 1 \rangle$
- `$\langle c_1 c_2, 1 \rangle$`
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

\[
\begin{align*}
S_{main} & : \text{read } a, b \\
 & : t := a \ast b \\
C_1 & : \text{call } p \\
R_1 & : \text{print } a \ast b
\end{align*}
\]

\[
\begin{align*}
\langle \lambda, 1 \rangle & \rightarrow \langle c_1, 1 \rangle \\
\langle c_1 c_2, 0 \rangle & \rightarrow \langle c_1 c_2, 0 \rangle
\end{align*}
\]

\[
\begin{align*}
S_p & : \text{if } a == 0 \\
C_2 & : \text{call } p
\end{align*}
\]

\[
\begin{align*}
\langle c_1, 1 \rangle & \rightarrow \langle c_1 c_2, 0 \rangle \\
\langle c_1 c_2, 0 \rangle & \rightarrow \langle c_1 c_2 c_2, 0 \rangle
\end{align*}
\]

\[
\begin{align*}
n_2 & : a = a - 1 \\
R_2 & : \langle c_1, 0 \rangle \\
t & : t = a \ast b
\end{align*}
\]

\[
\begin{align*}
\langle c_1, 1 \rangle & \rightarrow \langle c_1 c_2, 0 \rangle \\
\langle c_1 c_2, 0 \rangle & \rightarrow \langle c_1 c_2 c_2, 0 \rangle
\end{align*}
\]

\[
\begin{align*}
E_{main} & \rightarrow \langle c_1, 1 \rangle \\
E_p & \rightarrow \langle c_1 c_2, 1 \rangle
\end{align*}
\]
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

\( S_{\text{main}} \)
- read \( a, b \)
- \( t := a \times b \)

\( C_1 \)
- call \( p \)

\( R_1 \)
- print \( a \times b \)

\( E_{\text{main}} \)

\( S_p \)
- if \( a == 0 \)

\( n_2 \)
- \( a = a - 1 \)

\( C_2 \)
- call \( p \)

\( R_2 \)
- \( t = a \times b \)

\( E_p \)
Tutorial Problem #1

Perform available expressions analysis for the following program

```c
main()
{
    a = b*c;

    p();  /* C1 */
    d = b*c;  /* avail b*c? */
    q();  /* C2 */
    b = b*c;  /* avail b*c? */
}

p()
{
}

q()
{
    b = 5;
    p();  /* C3 */
}
```
The Need for Multiple Occurrences of a Call Site

Even if data flow values in cyclic call sequence do not change

```c
1. int a, b, c;
2. void main()
3. {
   c = a * b;
   p();
3. }
4. p();
5. }
6. void p()
7. {
   if (...) 
   8.   { p();
   9.     /* Is a*b available?*/
10.     a = a * b;
11.    }
12. }
```
The Need for Multiple Occurrences of a Call Site

Even if data flow values in cyclic call sequence do not change

```
1. int a, b, c;
2. void main()
3. {
4.     c = a * b;
5. }
6. void p()
7. {
8.     p();
9.     /* Is a * b available? */
10.    a = a * b;
11. }
12. }
```

Recursive calls: 1

Path 1

Oct 2013
The Need for Multiple Occurrences of a Call Site

Even if data flow values in cyclic call sequence do not change

1. int a, b, c;
2. void main()
3. {
4.     c = a * b;
5. }
6. void p()
7. {
8.     p();
9.     /* Is a*b available? */
10.     a = a * b;
11. }
12. }

Path 1
Recursive calls: 1

Path 2
Recursive calls: 2

Oct 2013
IIT Bombay
The Need for Multiple Occurrences of a Call Site

Even if data flow values in cyclic call sequence do not change

1. int a, b, c;
2. void main()
3. {
4.     c = a * b;
5.     p();
6. }
7. void p()
8. {
9.     p();
10.    Is a*b available?
11.    a = a * b;
12. }
The Need for Multiple Occurrences of a Call Site

Even if data flow values in cyclic call sequence do not change

1. int a,b,c;
2. void main()
3. {  c = a*b;
4.     p();
5. }
6. void p()
7. {  if (...)  
8.     {  p();
9.         Is a*b available?
10.        a = a*b;
11.     }
12.  }
The Need for Multiple Occurrences of a Call Site

Even if data flow values in cyclic call sequence do not change

1. int a,b,c;
2. void main()
3. {   c = a*b;
4.     p();
5. }
6. void p()
7. {   if (...)
8.     {   p();
9.     Is a*b available?
10.    a = a*b;
11. }
12. }
The Need for Multiple Occurrences of a Call Site

Even if data flow values in cyclic call sequence do not change

1. int a, b, c;
2. void main()
3. { c = a * b;
4. p();
5. }
6. void p()
7. { if (...) 
8. { p();
9. Is a*b available? 
10. a = a * b;
11. }
12. }

- Interprocedurally valid IFP
The Need for Multiple Occurrences of a Call Site

Even if data flow values in cyclic call sequence do not change

1. int a, b, c;
2. void main()
3. { c = a*b;
4. p();
5. }
6. void p()
7. { if (…) 
8. { p();
9. Is a*b available?
10. a = a*b;
11. }
12. }

- Interprocedurally valid IFP

\[ C_2, S_p, E_p, R_2, \quad \text{Kill} \quad n_2, E_p, R_2, n_2 \]
Even if data flow values in cyclic call sequence do not change

1. int a, b, c;
2. void main()
3. {   c = a * b;
4.     p();
5. }
6. void p()
7. {   if (...) 
8.     {     p();
9.     }   Is a*b available?
10.    a = a * b;
11. }
12. }

- Interprocedurally valid IFP

$C_2, S_p, C_2, S_p, E_p, R_2, n_2$
The Need for Multiple Occurrences of a Call Site

Even if data flow values in cyclic call sequence do not change

1. int a,b,c;
2. void main()
3. {
   c = a*b;
4.   p();
5. }
6. void p()
7. {
   if (...)  
   8.     { p();
   9.     Is a*b available?
   10.    a = a*b;
   11. }
12. }

- Interprocedurally valid IFP

\[ S_{main}, n_1, C_1, S_p, C_2, S_p, C_2, S_p, E_p, R_2, \text{Kill } n_2, E_p, R_2, n_2 \]
The Need for Multiple Occurrences of a Call Site

Even if data flow values in cyclic call sequence do not change

In terms of staircase diagram

- Interprocedurally valid IFP

\[ S_m, n_1, C_1, S_p, C_2, S_p, C_2, S_p, E_p, R_2, \text{Kill} n_2, E_p, R_2, n_2 \]
The Need for Multiple Occurrences of a Call Site

Even if data flow values in cyclic call sequence do not change

In terms of staircase diagram

- Interprocedurally valid IFP

\[ S_m, n_1, C_1, S_p, C_2, S_p, C_2, S_p, E_p, R_2, \text{Kill } n_2, E_p, R_2, n_2 \]

- You cannot descend twice, unless you ascend twice
The Need for Multiple Occurrences of a Call Site

Even if data flow values in cyclic call sequence do not change

In terms of staircase diagram

- Interprocedurally valid IFP

\[ S_m, n_1, C_1, S_p, C_2, S_p, C_2, S_p, E_p, R_2, \text{Kill } n_2, E_p, R_2, n_2 \]

- You cannot descend twice, unless you ascend twice

- Even if the data flow values do not change while ascending, you need to ascend because they may change while descending
### Tutorial Problem #2

Is `a*b` available on line 18 in the following program? On line 15? Construct its supergraph and argue in terms of interprocedurally valid paths

```plaintext
1. main()
2. {
3.   c = a*b;
4.   p();
5.   a = a*b;
6. }
7. p()
8. {   if (...)
9.    {   a = a*b;
10.   p();
11.   }
12.   else if (...)
13.    {   c = a * b;
14.    p();
15.    c = a;
16.   }
17.   else
18.    ; /* ignore */
19. }
```
Terminating Call String Construction

- For non-recursive programs: Number of call strings is finite
Terminating Call String Construction

- For non-recursive programs: Number of call strings is finite

- For recursive programs: Number of call strings could be infinite
  Fortunately, the problem is decidable for finite lattices
Terminating Call String Construction

- For non-recursive programs: Number of call strings is finite

- For recursive programs: Number of call strings could be infinite
  Fortunately, the problem is decidable for finite lattices
  - All call strings up to the following length must be constructed
Terminating Call String Construction

- For non-recursive programs: Number of call strings is finite

- For recursive programs: Number of call strings could be infinite
  Fortunately, the problem is decidable for finite lattices

  - All call strings up to the following length must be constructed
    - $K \cdot (|L| + 1)^2$ for general bounded frameworks
      ($L$ is the overall lattice of data flow values)
Terminating Call String Construction

- For non-recursive programs: Number of call strings is finite
- For recursive programs: Number of call strings could be infinite
  Fortunately, the problem is decidable for finite lattices
  ▶ **All** call strings upto the following length *must be* constructed
    ○ $K \cdot (|L| + 1)^2$ for general bounded frameworks
      ($L$ is the overall lattice of data flow values)
    ○ $K \cdot (|\hat{L}| + 1)^2$ for separable bounded frameworks
      ($\hat{L}$ is the component lattice for an entity)
Terminating Call String Construction

- For non-recursive programs: Number of call strings is finite
- For recursive programs: Number of call strings could be infinite

Fortunately, the problem is decidable for finite lattices

- All call strings upto the following length must be constructed
  - $K \cdot (|L| + 1)^2$ for general bounded frameworks
    ($L$ is the overall lattice of data flow values)
  - $K \cdot (|\hat{L}| + 1)^2$ for separable bounded frameworks
    ($\hat{L}$ is the component lattice for an entity)
  - $K \cdot 3$ for bit vector frameworks
Terminating Call String Construction

• For non-recursive programs: Number of call strings is finite

• For recursive programs: Number of call strings could be infinite
  Fortunately, the problem is decidable for finite lattices
  
  ▶ All call strings up to the following length must be constructed
    ○ $K \cdot (|L| + 1)^2$ for general bounded frameworks
      ($L$ is the overall lattice of data flow values)
    ○ $K \cdot (|\hat{L}| + 1)^2$ for separable bounded frameworks
      ($\hat{L}$ is the component lattice for an entity)
    ○ $K \cdot 3$ for bit vector frameworks
    ○ 3 occurrences of any call site in a call string for bit vector frameworks

  ⇒ Not a bound but prescribed necessary length
Terminating Call String Construction

- For non-recursive programs: Number of call strings is finite

- For recursive programs: Number of call strings could be infinite

Fortunately, the problem is decidable for finite lattices

- All call strings up to the following length must be constructed
  - $K \cdot (|L| + 1)^2$ for general bounded frameworks
    ($L$ is the overall lattice of data flow values)
  - $K \cdot (|\hat{L}| + 1)^2$ for separable bounded frameworks
    ($\hat{L}$ is the component lattice for an entity)
  - $K \cdot 3$ for bit vector frameworks
  - 3 occurrences of any call site in a call string for bit vector frameworks

$\Rightarrow$ Not a bound but prescribed necessary length

$\Rightarrow$ Large number of long call strings
Part 5

IPDFA Using Value Contexts
The Limitation of the Classical Call Strings Method

Required length of the call string is:

- \( K \) for non-recursive programs
- \( K \cdot (|L| + 1)^2 \) for recursive programs
The Role of Call Strings Length in Recursion (1)

- We consider self recursion for simplicity; the principles are general and are also applicable to indirect recursion
The Role of Call Strings Length in Recursion (1)

- We consider self recursion for simplicity; the principles are general and are also applicable to indirect recursion.
- Context sensitivity requires processing each recursive path independently.
The Role of Call Strings Length in Recursion (1)

- We consider self recursion for simplicity; the principles are general and are also applicable to indirect recursion.
- Context sensitivity requires processing each recursive path independently.
- For a given recursive path:
  - The *recursive call sequence (RCS)* refers to the subpath that builds recursion.
The Role of Call Strings Length in Recursion (1)

- We consider self recursion for simplicity; the principles are general and are also applicable to indirect recursion.
- Context sensitivity requires processing each recursive path independently.
- For a given recursive path:
  - The *recursive call sequence* (RCS) refers to the subpath that builds recursion.
  - The *recursive return sequence* (RRS) refers to the subpath that unfolds recursion.
The Role of Call Strings Length in Recursion (1)

- We consider self recursion for simplicity; the principles are general and are also applicable to indirect recursion.
- Context sensitivity requires processing each recursive path independently.
- For a given recursive path:
  - The \textit{recursive call sequence} (RCS) refers to the subpath that builds recursion.
  - The \textit{recursive return sequence} (RRS) refers to the subpath that unfolds recursion.
  - The \textit{recursion terminating path} (RTP) refers to the subpath from RCS to RRS.
The Role of Call Strings Length in Recursion (1)

- We consider self recursion for simplicity; the principles are general and are also applicable to indirect recursion.
- Context sensitivity requires processing each recursive path independently.
- For a given recursive path:
  - The *recursive call sequence* (RCS) refers to the subpath that builds recursion.
  - The *recursive return sequence* (RRS) refers to the subpath that unfolds recursion.
  - The *recursion terminating path* (RTP) refers to the subpath from RCS to RRS.
    (We assume that a static analysis can assign arbitrary data flow values to the unreachable parts of the program in the absence of an RTP.)
The Role of Call Strings Length in Recursion (2)

Data flow value reaching from outside
The Role of Call Strings Length in Recursion (2)

- The data flow value $\langle \sigma, x_0 \rangle$ is propagated over RCS
The data flow value $\langle \sigma, x_0 \rangle$ is propagated over RCS.
The Role of Call Strings Length in Recursion (2)

- The data flow value \(\langle \sigma, x_0 \rangle\) is propagated over RCS
- We get the new context \(\sigma c_i\) and the new data flow value \(x_1\)
The Role of Call Strings Length in Recursion (2)

- The new pair \( \langle \sigma c_i, x_1 \rangle \) is propagated over RCS
The new pair $\langle \sigma c_i, x_1 \rangle$ is propagated over RCS
The Role of Call Strings Length in Recursion (2)

- The new pair $\langle \sigma c_i, x_1 \rangle$ is propagated over RCS
- We get $\langle \sigma c_i c_i, x_2 \rangle$ at $S_p$
The Role of Call Strings Length in Recursion (2)

- Now the third pair \( \langle \sigma C_i C_i, x_2 \rangle \) is propagated over RCS
The Role of Call Strings Length in Recursion (2)

- Now the third pair $\langle \sigma c_i c_i, x_2 \rangle$ is propagated over RCS
The Role of Call Strings Length in Recursion (2)

- Now the third pair \( \langle \sigma c_i c_i, x_2 \rangle \) is propagated over RCS
- We get the new context \( \sigma c_i c_i c_i \) at \( S_p \)
- Assume that the data flow ceases to change
The Role of Call Strings Length in Recursion (2)

- Even if the data flow values do not change any further, we still need to propagate them in RCS in order to build call strings that are sufficiently large.
The Role of Call Strings Length in Recursion (2)

- Even if the data flow values do not change any further, we still need to propagate them in RCS in order to build call strings that are sufficiently large.
Even if the data flow values do not change any further, we still need to propagate them in RCS in order to build call strings that are sufficiently large.

We need large call strings to allow for all changes in RRS (as will be clear soon).
The Role of Call Strings Length in Recursion (2)

- Even if the data flow values do not change any further, we still need to propagate them in RCS in order to build call strings that are sufficiently large.
- We need large call strings to allow for all changes in RRS (as will be clear soon)

<table>
<thead>
<tr>
<th>Event</th>
<th>Call String</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma$</td>
<td>$x_0$</td>
</tr>
<tr>
<td>$\sigma C_i$</td>
<td>$x_1$</td>
</tr>
<tr>
<td>$\sigma C_i C_i$</td>
<td>$x_2$</td>
</tr>
<tr>
<td>$\sigma C_i C_i C_i$</td>
<td>$x_3$</td>
</tr>
</tbody>
</table>
The Role of Call Strings Length in Recursion (2)

- Even if the data flow values do not change any further, we still need to propagate them in RCS in order to build call strings that are sufficiently large.
- We need large call strings to allow for all changes in RRS (as will be clear soon).

Even if the data flow values do not change any further, we still need to propagate them in RCS in order to build call strings that are sufficiently large.

We need large call strings to allow for all changes in RRS (as will be clear soon).
The Role of Call Strings Length in Recursion (2)

- Assume that we construct all call strings as required by the conventional call strings method.
- Many of these call strings are redundant in that they do not correspond to any new data flow value.
The Role of Call Strings Length in Recursion (2)

- Assume that we construct all call strings as required by the conventional call strings method.
- Many of these call strings are redundant in that they do not correspond to any new data flow value.
- In our case, only the first two traversals over RCS compute new data flow values.
The Role of Call Strings Length in Recursion (2)

- Now we traverse the recursion terminating path

\[
\begin{align*}
\sigma & \rightarrow x_0 \\
\sigma C_i & \rightarrow x_1 \\
\sigma C_i C_i & \rightarrow x_2 \\
\sigma C_i C_i C_i & \rightarrow x_2 \\
\sigma C_i C_i C_i C_i & \rightarrow x_2 \\
\ldots & \rightarrow x_2
\end{align*}
\]
The Role of Call Strings Length in Recursion (2)

- Now we traverse the recursion terminating path.
- For simplicity, we propagate only some call strings to $E_p$ (possibly with changed data flow values).
The Role of Call Strings Length in Recursion (2)

- The call strings and data flow values reach the exit of $E_p$ unchanged.
The Role of Call Strings Length in Recursion (2)

- Now we start processing RRS
- The call strings ending with $c_i$ and their data flow values reach the entry of $R_i$ unchanged
The Role of Call Strings Length in Recursion (2)

- Now we start processing RRS
- The call strings ending with $c_i$ and their data flow values reach the entry of $R_i$ unchanged
- The last occurrence of $c_i$ is removed and the call strings reach the entry of $E_p$ with new data flow values
The Role of Call Strings Length in Recursion (2)

- We need to merge the data values of corresponding call strings reaching the entry of $E_p$ from $S_p$ and $R_i$
The Role of Call Strings Length in Recursion (2)

- We need to merge the data values of corresponding call strings reaching the entry of $E_p$ from $S_p$ and $R_i$
- We give new names to the resulting data flow values
The Role of Call Strings Length in Recursion (2)

- The call strings and their new data flow values reach the exit of \( E_p \) unchanged.

\[
\begin{align*}
\sigma \quad & \sigma c_i \quad & \sigma c_i c_i \quad & \sigma c_i c_i c_i \quad & \sigma c_i c_i c_i c_i \\
X_0 \quad & X_1 \quad & X_2 \quad & X_2 \quad & X_2 \\
\sigma \quad & \sigma c_i \quad & \sigma c_i c_i \quad & \sigma c_i c_i c_i \quad & \sigma c_i c_i c_i c_i \\
Y_0 \quad & Y_1 \quad & Y_2 \quad & Y_2 \quad & Y_2 \\
\sigma \quad & \sigma c_i \quad & \sigma c_i c_i \quad & \sigma c_i c_i c_i \quad & \sigma c_i c_i c_i c_i \\
Y_0 \quad & Y_1 \quad & Y_2 \quad & Y_2 \quad & Y_2 \\
\sigma \quad & \sigma c_i \quad & \sigma c_i c_i \quad & \sigma c_i c_i c_i \quad & \sigma c_i c_i c_i \\
Y_0 \quad & Y_1 \quad & Y_2 \quad & Y_2 \quad & Y_2 \\
\sigma \quad & \sigma c_i \quad & \sigma c_i c_i \quad & \sigma c_i c_i c_i \quad & \sigma c_i c_i c_i c_i \\
Y_0 \quad & Y_1 \quad & Y_2 \quad & Y_2 \quad & Y_2 \\
\sigma \quad & \sigma c_i \quad & \sigma c_i c_i \quad & \sigma c_i c_i c_i \quad & \sigma c_i c_i c_i c_i \\
Y_0 \quad & Y_1 \quad & Y_2 \quad & Y_2 \quad & Y_2 \\
\end{align*}
\]
The Role of Call Strings Length in Recursion (2)

- We process RRS once again
- The call strings ending with $c_i$ and their new data flow values reach the entry of $R_i$ unchanged
The Role of Call Strings Length in Recursion (2)

- We process RRS once again
- The call strings ending with $c_i$ and their new data flow values reach the entry of $R_i$ unchanged
- The last occurrence of $c_i$ is removed and the call strings reach the entry of $E_p$ with new data flow values.

---

Oct 2013 IIT Bombay
The Role of Call Strings Length in Recursion (2)

- We merge the data values at the entry of $E_p$ again and
The Role of Call Strings Length in Recursion (2)

We merge the data values at the entry of $E_p$ again and give new names to the resulting data flow values.
The Role of Call Strings Length in Recursion (2)

- The call strings and their new data flow values reach the exit of $E_p$ unchanged
The Role of Call Strings Length in Recursion (2)

- We process RRS yet another time
- The call strings ending with $c_i$ and their new data flow values reach the entry of $R_i$ unchanged
The Role of Call Strings Length in Recursion (2)

- We process RRS yet another time.
- The call strings ending with $c_i$ and their new data flow values reach the entry of $R_i$ unchanged.
- The last occurrence of $c_i$ is removed and the call strings reach the entry of $E_p$ with new data flow values.
We merge the data values at the entry of $E_p$ again and
The Role of Call Strings Length in Recursion (2)

- We merge the data values at the entry of $E_p$ again and
- give new names to the resulting data flow values
The Role of Call Strings Length in Recursion (2)

- The call strings and their new data flow values reach the exit of $E_p$ unchanged.
The Role of Call Strings Length in Recursion (2)

- We are now processing RRS the fourth time, whereas the data flow values in RCS changed only twice.
- Since the last $c_i$ is removed every time $R_i$ is visited, we can visit it at most as many times as the number of $c_i$ in a call string.
- We build call strings while processing RCS, and since at that time we do not know the number of times RRS may have to be processed, we build large call strings.
The Role of Call Strings Length in Recursion (2)

- We are now processing RRS the fourth time whereas the data flow values in RCS changed only twice.
- Since the last $c_i$ is removed every time $R_i$ is visited, we can visit it at most as many times as the number of $c_i$ in a call string.
- We build call strings while processing RCS, and since at that time we do not know the number of times RRS may have to be processed, we build large call strings.
The Role of Call Strings Length in Recursion (2)

- Assume that we have now got our final values.
- We merge them at the entry of $E_p$ again and...
The Role of Call Strings Length in Recursion (2)

- Assume that we have now got our final values
- We merge them at the entry of $E_p$ again and
- give new names to the resulting values
The Role of Call Strings Length in Recursion (2)

- The call strings and their final data flow values reach the exit of $E_p$ unchanged.
The Role of Call Strings Length in Recursion (2)

- We need to process RRS once again to discover that there are no changes

\[
\begin{align*}
\sigma & \quad \sigma C_i & \quad \sigma C_i C_i & \quad \sigma C_i C_i C_i & \quad \sigma C_i C_i C_i C_i & \quad \sigma C_i C_i C_i C_i C_i \\
X_0 & \quad X_1 & \quad X_2 & \quad X_2 & \quad X_2 & \quad X_2 \\
S_p & \quad C_i & \quad R_i & \quad E_p
\end{align*}
\]
The Role of Call Strings Length in Recursion (2)

- We need to process RRS once again to discover that there are no changes
The Role of Call Strings Length in Recursion (2)
The Role of Call Strings Length in Recursion: Summary

- Context sensitivity in recursion requires matching the number of traversals over RCS and RRS.
- For a forward analysis the call strings are constructed while traversing RCS and are consumed while traversing RRS.
- At the time of traversing RCS, we do not know how many times do we need to traverse the corresponding RRS.
- In order to allow an adequate number of traversals over RRS, we construct large call strings in anticipation while traversing RCS.
The Role of Call Strings Length in Recursion: Summary

- Context sensitivity in recursion requires matching the number of traversals over RCS and RRS
- For a forward analysis the call strings are constructed while traversing RCS and are consumed while traversing RRS
- At the time of traversing RCS, we do not know how many times do we need to traverse the corresponding RRS
- In order the allow an adequate number of traversals over RRS, we construct large call strings in anticipation while traversing RCS

*The main reason behind building long call strings is to allow an adequate number of traversals over RRS*
Overcoming the Problem of Long Call Strings

- Under the assumption of finiteness of data flow values, recursion will eventually generate the same values as already generated.
- A Possible Solution:
  Identify that the same value is being generated and avoid re-analysis.
Value Contexts: Key Ideas

- Data flow value invariant: If $\sigma_1$ and $\sigma_2$ have equal values at $S_p$, then
Value Contexts: Key Ideas

• Data flow value invariant: If $\sigma_1$ and $\sigma_2$ have equal values at $S_p$, then
  ▶ since $\sigma_1$ and $\sigma_2$ are transformed in the same manner by traversing the same set of paths,
  ▶ the values associated with them will also be transformed in the same manner and will continue to remain equal at $E_p$. 
Value Contexts: Key Ideas

• Data flow value invariant: If $\sigma_1$ and $\sigma_2$ have equal values at $S_p$, then
  - since $\sigma_1$ and $\sigma_2$ are transformed in the same manner by traversing the same set of paths,
  - the values associated with them will also be transformed in the same manner and will continue to remain equal at $E_p$.

• We can reduce the amount of effort by
  - Partitioning the call strings at $S_p$ based on values
  - Performing the analysis of $p$ for each such partition separately
Value Contexts: Key Ideas

• Data flow value invariant: If $\sigma_1$ and $\sigma_2$ have equal values at $S_p$, then
  ▶ since $\sigma_1$ and $\sigma_2$ are transformed in the same manner by traversing the same set of paths,
  ▶ the values associated with them will also be transformed in the same manner and will continue to remain equal at $E_p$.

• We can reduce the amount of effort by
  ▶ Partitioning the call strings at $S_p$ based on values
  ▶ Performing the analysis of $p$ for each such partition separately

• Combining two views of a context
Value Contexts: Key Ideas

• Data flow value invariant: If $\sigma_1$ and $\sigma_2$ have equal values at $S_p$, then
  ▶ since $\sigma_1$ and $\sigma_2$ are transformed in the same manner by traversing the same set of paths,
  ▶ the values associated with them will also be transformed in the same manner and will continue to remain equal at $E_p$.

• We can reduce the amount of effort by
  ▶ Partitioning the call strings at $S_p$ based on values
  ▶ Performing the analysis of $p$ for each such partition separately

• Combining two views of a context
  ▶ Call strings as calling contexts
  ▶ Data flow values at the call site as value contexts
Value Contexts: Key Ideas

• Data flow value invariant: If $\sigma_1$ and $\sigma_2$ have equal values at $S_p$, then
  ▶ since $\sigma_1$ and $\sigma_2$ are transformed in the same manner by traversing the same set of paths,
  ▶ the values associated with them will also be transformed in the same manner and will continue to remain equal at $E_p$.

• We can reduce the amount of effort by
  ▶ Partitioning the call strings at $S_p$ based on values
  ▶ Performing the analysis of $p$ for each such partition separately

• Combining two views of a context
  ▶ Call strings as calling contexts
  ▶ Data flow values at the call site as value contexts

Maintain distinct data flow values for each context of a procedure
Interprocedural Data Flow Analysis Using Value Contexts

- A value context is defined by a particular input data flow value reaching a procedure.
- It is used to enumerate the summary flow functions in terms of (input $\rightarrow$ output) pairs.
- In order to compute these pairs, data flow analysis within a procedure is performed separately for each context (i.e. input data flow value).
- When a new call to a procedure is encountered, the pairs are consulted to decide if the procedure needs to be analysed again.
  - If it was already analysed once for the input value, output can be directly processed.
  - Otherwise, a new context is created and the procedure is analysed for this new context.
Understanding Value Contexts
Understanding Value Contexts

Separate contexts are created for each unique data flow value.
Understanding Value Contexts

Call q

$s_0 \xrightarrow{\sigma_0} x_0$

$s_1 \xrightarrow{\sigma_1} x_1$

$s_2 \xrightarrow{\sigma_2} x_2$

$s_3 \xrightarrow{\sigma_3} x_3$

$S_p$

$C_i$

$R_i$

$E_p$

$S_q$

$E_q$
Understanding Value Contexts

Distinct data flow values are maintained for each context (i.e. each procedure is analysed separately for each context)
Understanding Value Contexts

New contexts are created for data flow values reaching $q$

Context transitions on call sites are recorded globally

$S_0$ $S_1$ $S_2$ $S_3$

$\sigma_0 \ x_0$
$\sigma_1 \ x_1$
$\sigma_2 \ x_2$
$\sigma_3 \ x_3$

$S_P$

$C_i$

$S_0 C_i$
$S_1 C_i$
$S_2 C_i$
$S_3 C_i$

$\frac{\sigma_0}{x_0}$
$\frac{\sigma_1}{x_1}$
$\frac{\sigma_2}{x_2}$
$\frac{\sigma_3}{x_3}$

$E_P$

$E_q$
Understanding Value Contexts

New contexts are created for data flow values reaching $q$

Context transitions on call sites are recorded globally
Understanding Value Contexts

\[
\begin{align*}
S_0 &\xrightarrow{\sigma_0} S_0 \\
S_1 &\xrightarrow{\sigma_1} S_1 \\
S_2 &\xrightarrow{\sigma_2} S_2 \\
S_3 &\xrightarrow{\sigma_3} S_3 \\
S_4 &\xrightarrow{\sigma_4} S_4 \\
S_5 &\xrightarrow{\sigma_5} S_5 \\
S_6 &\xrightarrow{\sigma_6} S_6 \\
\end{align*}
\]

\[
\begin{align*}
S_p &\xrightarrow{C_i} S_p \\
C_i &\xrightarrow{R_i} C_i \\
E_p &\xrightarrow{E_q} E_p \\
\end{align*}
\]
Understanding Value Contexts

\[ S_0 \xrightarrow{\sigma_0} X_0 \]
\[ S_1 \xrightarrow{\sigma_1} X_1 \]
\[ S_2 \xrightarrow{\sigma_2} X_2 \]
\[ S_3 \xrightarrow{\sigma_3} X_3 \]

\[ S_p \]

\[ C_i \]

\[ R_i \]

\[ E_p \]

\[ S_q \]

\[ S_0 \xrightarrow{S_4} C_i \]
\[ S_1 \xrightarrow{S_4} C_i \]
\[ S_2 \xrightarrow{S_5} C_i \]
\[ S_3 \xrightarrow{S_6} C_i \]
Context transitions are consulted to transfer data flow values to calling contexts.
Understanding Value Contexts

Context transitions are consulted to transfer data flow values to calling contexts.

Oct 2013 IIT Bombay
Understanding Value Contexts

\[
\begin{align*}
S_0 & \xrightarrow{\sigma_0} S_1 \\
S_1 & \xrightarrow{\sigma_1} S_2 \\
S_2 & \xrightarrow{\sigma_2} S_3 \\
S_3 & \xrightarrow{\sigma_3} S_4 \\
S_4 & \xrightarrow{\sigma_4} S_5 \\
S_5 & \xrightarrow{\sigma_5} S_6 \\
S_6 & \xrightarrow{\sigma_6} S_0
\end{align*}
\]
Understanding Value Contexts

\[ S_0 \quad S_1 \quad S_2 \quad S_3 \]

\[ \sigma_0^{x_0} \quad \sigma_1^{x_1} \quad \sigma_2^{x_2} \quad \sigma_3^{x_3} \]

\[ S_4 \quad S_5 \quad S_6 \]

\[ \sigma_0^{x_0} \quad \sigma_1^{x_1} \quad \sigma_2^{x_2} \quad \sigma_3^{x_3} \]

\[ S_p \]

\[ C_i \]

\[ R_i \]

\[ E_p \]

\[ S_q \]

\[ S_4 \quad S_5 \quad S_6 \]

\[ \sigma_0^{y_0} \quad \sigma_1^{y_1} \quad \sigma_2^{y_2} \quad \sigma_3^{y_3} \]

\[ S_0 \quad S_1 \quad S_2 \quad S_3 \]

\[ C_i \]

Oct 2013 IIT Bombay
Understanding Value Contexts

$S_0 \xrightarrow{x_0} \sigma_0$
$S_1 \xrightarrow{x_1} \sigma_1$
$S_2 \xrightarrow{x_2} \sigma_2$
$S_3 \xrightarrow{x_3} \sigma_3$
$S_4 \xrightarrow{x_0} \sigma_0 C_i$
$S_5 \xrightarrow{x_1} \sigma_1 C_i$
$S_6 \xrightarrow{x_3} \sigma_3 C_i$

$S_p \xrightarrow{C_i} E_p$

$S_q \xrightarrow{C_i} E_q$

$S_0 \xrightarrow{C_i} S_4$
$S_1 \xrightarrow{C_i} S_5$
$S_2 \xrightarrow{C_i} S_6$
$S_3 \xrightarrow{C_i} S_6$

Oct 2013
Defining Value Contexts

- The set of value contexts is $VC = Procs \times L$

A value context $X = \langle proc, entryValue \rangle \in VC$
where $proc \in Procs$ and $entryValue \in L$
Defining Value Contexts

- The set of value contexts is $VC = Procs \times L$
  
  A value context $X = \langle proc, entryValue \rangle \in VC$
  where $proc \in Procs$ and $entryValue \in L$

- Supporting functions ($CS$ is the set of call sites)
  - $exitValue : VC \mapsto L$
  - $transitions : (VC \times CS) \mapsto VC$
Defining Value Contexts

- The set of value contexts is $VC = Procs \times L$
  
  A value context $X = \langle proc, entryValue \rangle \in VC$
  where $proc \in Procs$ and $entryValue \in L$

- Supporting functions ($CS$ is the set of call sites)
  
  - $exitValue : VC \mapsto L$
    
    eg. $exitValue(X) = v$
  
  - $transitions : (VC \times CS) \mapsto VC$
    
    eg. $X \xrightarrow{C_i} Y$
Interprocedural Data Flow Analysis Using Value Contexts

- The method works with a collection of control flow graphs
  - No need of supergraph
    - No need to distinguish between $C_i$ and $R_i$
    - No need of call ($C_i \rightarrow S_p$) and return ($E_p \rightarrow E_i$) edges
- Maintain a work list $WL$ of entries $\langle context, node \rangle$
  (in reverse post order of nodes within a procedure for forward flows)
- Notation:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle p, v \rangle$</td>
<td>Context for procedure $p$ with data flow value $v$</td>
</tr>
<tr>
<td>$X</td>
<td>m$</td>
</tr>
<tr>
<td>$X. v$</td>
<td>Data flow value in context $X$ is $v$</td>
</tr>
<tr>
<td>$Out_m[X]$</td>
<td>Data flow value of context $X$ in $Out_m$</td>
</tr>
<tr>
<td>$X \xrightarrow{C_i} Y$</td>
<td>Transition from context $X$ to context $Y$ at call site $C_i$</td>
</tr>
</tbody>
</table>
Interprocedural Data Flow Analysis Using Value Contexts

- Select $X|n$ from $WL$. Compute $I_n$. Let $X.v$ be in $I_n$
Interprocedural Data Flow Analysis Using Value Contexts

- Select $X|n$ from $WL$. Compute $In_n$. Let $X.v$ be in $In_n$
  - If $n = C_i$ calling procedure $p$
  - If $n = E_p$
  - For all other nodes
Interprocedural Data Flow Analysis Using Value Contexts

- Select $X|n$ from WL. Compute $ln_n$. Let $X.v$ be in $ln_n$
  - If $n = C_i$ calling procedure $p$
    - If some context $\langle p, v \rangle$ exists (say $Y$)
      - If it does not exist
  - If $n = E_p$
  - For all other nodes
Interprocedural Data Flow Analysis Using Value Contexts

- Select $X|n$ from $WL$. Compute $In_n$. Let $X.v$ be in $In_n$
  - If $n = C_i$ calling procedure $p$
    - If some context $\langle p, v \rangle$ exists (say $Y$)
      - record the transition $X \xrightarrow{C_i} Y$
      - $Out_{C_i}[X] = exitValue(Y)$
      - if there is a change, add $X|m, \forall m \in succ(C_i)$ to $WL$
    - If it does not exist
  - If $n = E_p$

- For all other nodes
**Interprocedural Data Flow Analysis Using Value Contexts**

- Select $X | n$ from $WL$. Compute $In_n$. Let $X . v$ be in $In_n$
  - If $n = C_i$ calling procedure $p$
    - If some context $⟨p, v⟩$ exists (say $Y$)
      - record the transition $X \xrightarrow{C_i} Y$
      - $Out_{C_i}[X] = exitValue(Y)$
      - if there is a change, add $X | m$, $∀ m \in succ(C_i)$ to $WL$
    - If it does not exist
      - create a new context $Y = ⟨p, v⟩$
      - initialize $exitValue(Y) = \top$
      - record the transition $X \xrightarrow{C_i} Y$
      - add entries $Y | m$ for all nodes $m$ of procedure $p$ to $WL$
  - If $n = E_p$

- For all other nodes
Interprocedural Data Flow Analysis Using Value Contexts

- Select $X|n$ from $WL$. Compute $In_n$. Let $X.v$ be in $In_n$
  - If $n = C_i$ calling procedure $p$
    - If $n = E_p$
      - Set $exitValue(X) = v$
  - For all other nodes
Interprocedural Data Flow Analysis Using Value Contexts

- Select $X|n$ from $WL$. Compute $ln_n$. Let $X.v$ be in $ln_n$
  - If $n = C_i$ calling procedure $p$

- If $n = E_p$
  - Set $exitValue(X) = v$
  - Find out all transitions $Z \xrightarrow{C_j} X$
    - Set $Out_{C_j}[Z] = v$
    - If there is a change, add $Z|m$, $\forall m \in succ(C_j)$ to $WL$

- For all other nodes
Interprocedural Data Flow Analysis Using Value Contexts

- Select $X|n$ from $WL$. Compute $In_n$. Let $X.v$ be in $In_n$
  - If $n = C_i$ calling procedure $p$
    - If $n = E_p$
      - For all other nodes
        - Set $Out_n[X] = f_n(v)$
Interprocedural Data Flow Analysis Using Value Contexts

- Select $X|n$ from $WL$. Compute $In_n$. Let $X.v$ be in $In_n$
  - If $n = C_i$ calling procedure $p$

- If $n = E_p$

- For all other nodes
  - Set $Out_n[X] = f_n(v)$
  - If there is a change, add $X|m, \forall m \in succ(n)$ to $WL$
Interprocedural Data Flow Analysis Using Value Contexts

- Select $X|n$ from $WL$. Compute $In_n$. Let $X.v$ be in $In_n$
  - If $n = C_i$ calling procedure $p$
    - If some context $\langle p, v \rangle$ exists (say $Y$)
      - record the transition $X \xrightarrow{C_i} Y$
      - $Out_{C_i}[X] = exitValue(Y)$
      - if there is a change, add $X|m$, $\forall m \in succ(C_i)$ to $WL$
    - If it does not exist
      - create a new context $Y = \langle p, v \rangle$
      - initialize $exitValue(Y) = \top$
      - record the transition $X \xrightarrow{C_i} Y$
      - add entries $Y|m$ for all nodes $m$ of procedure $p$ to $WL$
  - If $n = E_p$
    - Set $exitValue(X) = v$
    - Find out all transitions $Z \xrightarrow{C_i} X$
      - Set $Out_{C_i}[Z] = v$
      - If there is a change, add $Z|m$, $\forall m \in succ(C_i)$ to $WL$
  - For all other nodes
    - Set $Out_n[X] = f_n(v)$
    - If there is a change, add $X|m$, $\forall m \in succ(n)$ to $WL$
Available Expressions Analysis Using Value Contexts

\[ S_{main} \]
- read \( a, b \)
- \( t := a \times b \)

\[ C_1 \]
- call \( p \)

\[ n_1 \]
- print \( a \times b \)

\[ E_{main} \]

\[ S_p \]
- if \( a == 0 \)

\[ n_2 \]
- \( a = a - 1 \)

\[ C_2 \]
- call \( p \)

\[ n_3 \]
- \( t = a \times b \)

\[ E_p \]
Available Expressions Analysis Using Value Contexts

\[ S_{main} \]
- read \( a, b \)
- \( t := a \ast b \)
- \( C_1 \) call \( p \)
- \( n_1 \) print \( a \ast b \)
- \( E_{main} \)

\[ S_p \]
- if \( a == 0 \)
- \( n_2 \) \( a = a - 1 \)
- \( C_2 \) call \( p \)
- \( n_3 \) \( t = a \ast b \)
- \( E_p \)

Is \( a \ast b \) available?
Available Expressions Analysis Using Value Contexts

```
int a, b, t;
void p()
{
    if (a == 0)
    {
        a = a - 1;
        p();
        t = a * b;
    }
}
```
Available Expressions Analysis Using Value Contexts

\[ S_{\text{main}} \]
- read \( a, b \)
- \( t := a \times b \)

\[ C_1 \]
- call \( p \)

\[ n_1 \]
- print \( a \times b \)

\[ E_{\text{main}} \]

\[ S_p \]
- if \( a == 0 \)

\[ n_2 \]
- \( a = a - 1 \)

\[ C_2 \]
- call \( p \)

\[ n_3 \]
- \( t = a \times b \)

\[ E_p \]

Is \( a \times b \) available?

Yes!

```
int a, b, t;
void p()
{
    if (a == 0)
        {
            a = a-1;
            p();
            t = a*b;
        }
}
```
Available Expressions Analysis Using Value Contexts

\[ S_{main} \quad \text{read } a, b \]
\[ t := a \times b \]
\[ C_1 \quad \text{call } p \]
\[ n_1 \quad \text{print } a \times b \]
\[ E_{main} \]

\[ S_p \quad \text{if } a == 0 \]
\[ n_2 \quad a = a - 1 \]
\[ C_2 \quad \text{call } p \]
\[ n_3 \quad t = a \times b \]
\[ E_p \]
Available Expressions Analysis Using Value Contexts

Create a new context $X_0$ with $BI$ which is 0 for available expressions analysis.
Available Expressions Analysis Using Value Contexts

\[ WL = [X_0|S_m, X_0|C_1, X_0|n_1, X_0|E_m] \]

Create a new context \( X_0 \) with \( BI \) which is 0 for available expressions analysis
Initialize \( exitValue(X_0) \) to \( \top = 1 \)
Initialize the work list with all nodes in procedure main for \( X_0 \)
Available Expressions Analysis Using Value Contexts

\[ WL = [X_0|S_m, X_0|C_1, X_0|n_1, X_0|E_m] \]

<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X_0 = \langle \text{main},0 \rangle)</td>
<td>1</td>
</tr>
</tbody>
</table>

Compute the data flow values for \(S_m\) for context \(X_0\)
Available Expressions Analysis Using Value Contexts

\[ WL = [X_0|C_1, X_0|n_1, X_0|E_m] \]

Context | exitValue
---|---
\( X_0 = \langle \text{main}, 0 \rangle \) | 1

Compute the data flow values for \( S_m \) for context \( X_0 \).
Available Expressions Analysis Using Value Contexts

\[ WL = [X_0|C_1, X_0|n_1, X_0|E_m] \]

<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_0 = \langle \text{main}, 0 \rangle )</td>
<td>1</td>
</tr>
</tbody>
</table>
Available Expressions Analysis Using Value Contexts

\[ WL = [X_1|S_p, X_1|n_2, X_1|C_2, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

Create a new context \( X_1 \) with entry value 1
Record the transition to \( X_1 \)
Initialize \( exitValue(X_1) \) to \( T = 1 \)
Add all nodes of procedure \( p \) to the work list for \( X_1 \)
Available Expressions Analysis Using Value Contexts

\[ WL = \left[ X_1 | S_p, X_1 | n_2, X_1 | C_2, X_1 | n_3, X_1 | E_p, X_0 | n_1, X_0 | E_m \right] \]

\[ X_0 = \langle \text{main,0} \rangle \]
\[ X_1 = \langle \text{p,1} \rangle \]

\[ \begin{array}{|c|c|}
\hline
\text{Context} & \text{exitValue} \\
\hline
X_0 & 1 \\
X_1 & 1 \\
\hline
\end{array} \]
Available Expressions Analysis Using Value Contexts

\[ WL = [X_1|n_2, X_1|C_2, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ X_{0.0} \]

\[ S_{main} \]

read \( a, b \)

\( t := a \times b \)

\[ X_{0.1} \]

\[ C_1 \]

call \( p \)

\[ n_1 \]

print \( a \times b \)

\[ X_{1.0} \]

\[ X_{1.1} \]

\[ S_p \]

if \( a == 0 \)

\[ X_{1.1} \]

\[ n_2 \]

\( a = a - 1 \)

\[ X_{1.1} \]

\[ n_3 \]

\( t = a \times b \)

\[ E_{main} \]

\[ E_p \]

\[ Context \]

<table>
<thead>
<tr>
<th>exitValue</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_0 = \langle \text{main},0 \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_1 = \langle p,1 \rangle )</td>
<td>1</td>
</tr>
</tbody>
</table>
Available Expressions Analysis Using Value Contexts

\[ WL = [X_1|n_2, X_1|C_2, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_0 = \langle \text{main},0 \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_1 = \langle p,1 \rangle )</td>
<td>1</td>
</tr>
</tbody>
</table>
Available Expressions Analysis Using Value Contexts

\[
WL = [X_1|n_2, X_1|C_2, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m]
\]

- **Context** | **exitValue**
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X₀ = \langle \text{main,0} \rangle</td>
</tr>
<tr>
<td>X₁ = \langle p,1 \rangle</td>
</tr>
</tbody>
</table>
Available Expressions Analysis Using Value Contexts

\[ WL = [X_1|C_2, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_0 = \langle \text{main},0 \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_1 = \langle p,1 \rangle )</td>
<td>1</td>
</tr>
</tbody>
</table>

Oct 2013
Available Expressions Analysis Using Value Contexts

\[ WL = [X_1|C_2, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

Context | exitValue
---|---
\(X_0 = \langle \text{main}, 0 \rangle\) | 1
\(X_1 = \langle p, 1 \rangle\) | 1
Available Expressions Analysis Using Value Contexts

\[ WL = [X_1|C_2, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

Since there is no context for \( p \) with value 0, create context \( X_2 \)

Record the transition to \( X_2 \)

Initialize \( \text{exitValue}(X_2) \) to \( 1 \)

Add all nodes of procedure \( p \) to the work list for \( X_2 \)
Available Expressions Analysis Using Value Contexts

\[ WL = [X_2|S_p, X_2|n_2, X_2|C_2, X_2|n_3, X_2|E_p, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ X_0 = \langle \text{main,0} \rangle \]
\[ X_1 = \langle \text{p,1} \rangle \]
\[ X_2 = \langle \text{p,0} \rangle \]

\[ \text{Context} \quad | \quad \text{exitValue} \]
\|-----------------|------|
\| \langle \text{main,0} \rangle \| 1 \]
\| \langle \text{p,1} \rangle \| 1 \]
\| \langle \text{p,0} \rangle \| 1 \]

\( S_{\text{main}} \)

read \( a, b \)
\( t := a \times b \)

\( \mathcal{X}_0.0 \)

\( X_0 \rightarrow C_1 \rightarrow X_1 \rightarrow C_2 \rightarrow X_2 \)

\( n_1 \)

print \( a \times b \)

\( \mathcal{X}_0.1 \)

\( C_1 \)

\( \mathcal{X}_1.1 \)

\( S_p \quad \text{if} \ a == 0 \)

\( n_2 \)

\( a = a - 1 \)

\( \mathcal{X}_1.0 \)

\( C_2 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)

\( \mathcal{X}_2.0 \)
Available Expressions Analysis Using Value Contexts

\[ WL = [X_2|S_p, X_2|n_2, X_2|C_2, X_2|n_3, X_2|E_p, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_0 = \langle \text{main}, 0 \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_1 = \langle \text{p}, 1 \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_2 = \langle \text{p}, 0 \rangle )</td>
<td>1</td>
</tr>
</tbody>
</table>

\( X_0 \): read \( a, b \) \( t := a \times b \) \n\( C_1 \): call \( p \) \n\( n_1 \): print \( a \times b \) 
\( X_1 \): if \( a == 0 \) \n\( E_p \): print \( a \times b \) 
\( C_2 \): call \( p \) 
\( n_3 \): \( t = a \times b \)
Available Expressions Analysis Using Value Contexts

\[ WL = [X_2|n_2, X_2|C_2, X_2|n_3, X_2|E_p, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>X_0 = ⟨main,0⟩</td>
<td>1</td>
</tr>
<tr>
<td>X_1 = ⟨p,1⟩</td>
<td>1</td>
</tr>
<tr>
<td>X_2 = ⟨p,0⟩</td>
<td>1</td>
</tr>
</tbody>
</table>
Available Expressions Analysis Using Value Contexts

\[ WL = [X_2|n_2, X_2|C_2, X_2|n_3, X_2|E_p, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ X_0 = \langle \text{main},0 \rangle \]
\[ X_1 = \langle p,1 \rangle \]
\[ X_2 = \langle p,0 \rangle \]

\[ \text{Context} \quad \text{exitValue} \]
\[ X_0 = \langle \text{main},0 \rangle \quad 1 \]
\[ X_1 = \langle p,1 \rangle \quad 1 \]
\[ X_2 = \langle p,0 \rangle \quad 1 \]
Available Expressions Analysis Using Value Contexts

\[WL = [X_2|C_2, X_2|n_3, X_2|E_p, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m]\]
Available Expressions Analysis Using Value Contexts

\[ WL = [X_2|C_2, X_2|n_3, X_2|E_p, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ S_{main} \]

\[ read \ a, b \]

\[ t := a \ast b \]

\[ \begin{align*}
X_0.0 & \quad \text{read } a, b \\
X_0.1 & \quad t := a \ast b
\end{align*} \]

\[ C_1 \]

\[ \text{call } p \]

\[ n_1 \]

\[ \text{print } a \ast b \]

\[ E_{main} \]

\[ S_p \]

\[ \text{if } a == 0 \]

\[ \begin{align*}
X_0.0 & \quad \text{read } a, b \\
X_1.1 & \quad t := a \ast b
\end{align*} \]

\[ X_1.0 \quad a = a - 1 \]

\[ X_1.1 \quad \text{call } p \]

\[ X_2.0 \quad \text{call } p \]

\[ n_2 \quad a = a - 1 \]

\[ n_3 \quad t = a \ast b \]

\[ E_p \]

\[ Context \quad \text{exitValue} \]

\[ \begin{array}{|c|c|}
\hline
X_0 & \langle \text{main,0} \rangle & 1 \\
\hline
X_1 & \langle \text{p,1} \rangle & 1 \\
\hline
X_2 & \langle \text{p,0} \rangle & 1 \\
\hline
\end{array} \]

\( p \) has context \( X_2 \) with value 0 so no need to create a new context.
Available Expressions Analysis Using Value Contexts

\[ WL = [X_2|C_2, X_2|n_3, X_2|E_p, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ X_0 = \langle \text{main}, 0 \rangle \]
\[ X_1 = \langle p, 1 \rangle \]
\[ X_2 = \langle p, 0 \rangle \]

\[ S_{\text{main}} \]
- \text{read } a, b
- \text{t := } a \ast b

\[ S_p \]
- if \ a == 0

\[ n_2 \]
- a = a - 1

\[ n_3 \]
- t = a \ast b

\text{p} \text{ has context } X_2 \text{ with value 0 so no need to create a new context}

Record the transition from context \( X_2 \) to itself
Available Expressions Analysis Using Value Contexts

\[ WL = [X_2|C_2, X_2|n_3, X_2|E_p, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

- **Context** | **exitValue**
  - \( X_0 = \langle \text{main}, 0 \rangle \) | 1
  - \( X_1 = \langle p, 1 \rangle \) | 1
  - \( X_2 = \langle p, 0 \rangle \) | 1

\( p \) has context \( X_2 \) with value 0 so no need to create a new context
- Record the transition from context \( X_2 \) to itself
- Use the \( \text{exitValue}(X_2) \) to compute \( \text{Out}_{C_2}[X_2] \)
Available Expressions Analysis Using Value Contexts

\[ WL = [X_2|n_3, X_2|E_p, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_0 = \langle \text{main}, 0 \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_1 = \langle p, 1 \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_2 = \langle p, 0 \rangle )</td>
<td>1</td>
</tr>
</tbody>
</table>
Available Expressions Analysis Using Value Contexts

\[ WL = \{ X_2|n_3, X_2|E_p, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m \} \]

<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X_0 = \langle \text{main},0\rangle)</td>
<td>1</td>
</tr>
<tr>
<td>(X_1 = \langle \text{p},1\rangle)</td>
<td>1</td>
</tr>
<tr>
<td>(X_2 = \langle \text{p},0\rangle)</td>
<td>1</td>
</tr>
</tbody>
</table>
Available Expressions Analysis Using Value Contexts

\[ WL = [X_2|E_p, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_0 = \langle \text{main}, 0 \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_1 = \langle p, 1 \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_2 = \langle p, 0 \rangle )</td>
<td>1</td>
</tr>
</tbody>
</table>
Available Expressions Analysis Using Value Contexts

\[ WL = [X_2|E_p, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

At \( E_p \) the values from \( S_p \) and \( n_3 \) are merged for context \( X_2 \)

<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_0 = \langle \text{main}, 0 \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_1 = \langle p, 1 \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_2 = \langle p, 0 \rangle )</td>
<td>1</td>
</tr>
</tbody>
</table>

\( S_{\text{main}} \):
- read \( a, b \)
- \( t := a \times b \)

\( C_1 \):
- call \( p \)

\( n_1 \):
- print \( a \times b \)

\( E_{\text{main}} \):
Available Expressions Analysis Using Value Contexts

\[ WL = [X_2|E_p, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_2|E_p, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

- **Context**
  - \( X_0 = \langle \text{main}, 0 \rangle \)
  - \( X_1 = \langle p, 1 \rangle \)
  - \( X_2 = \langle p, 0 \rangle \)

- **exitValue**
  - \( X_0 = 1 \)
  - \( X_1 = 1 \)
  - \( X_2 = 0 \)

At \( E_p \) the values from \( S_p \) and \( n_3 \) are merged for context \( X_2 \)

\( \text{exitValue}(X_2) \) is set to 0
Available Expressions Analysis Using Value Contexts

\[ WL = [X_2|E_p, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

At \( E_p \) the values from \( S_p \) and \( n_3 \) are merged for context \( X_2 \)

\[ \text{exitValue}(X_2) \] is set to 0

Since \( X_2 \) has transitions \( X_1 \xrightarrow{C_2} X_2 \) and \( X_2 \xrightarrow{C_2} X_2 \), \( Out_{C_2}[X_1] \) and \( Out_{C_2}[X_2] \) become 0
Available Expressions Analysis Using Value Contexts

\[ WL = [X_2|n_3, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

At \( E_p \) the values from \( S_p \) and \( n_3 \) are merged for context \( X_2 \)

\( \text{exitValue}(X_2) \) is set to 0

Since \( X_2 \) has transitions \( X_1 \xrightarrow{C_2} X_2 \) and \( X_2 \xrightarrow{C_2} X_2 \), \( \text{Out}_{C_2}[X_1] \) and \( \text{Out}_{C_2}[X_2] \) become 0

Since \( \text{Out}_{C_2}[X_2] \) changes, \( X_2|n_3 \) is added to the work list.
Available Expressions Analysis Using Value Contexts

\[ WL = [X_2|n_3, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X_0 = \langle \text{main}, 0 \rangle)</td>
<td>1</td>
</tr>
<tr>
<td>(X_1 = \langle p, 1 \rangle)</td>
<td>1</td>
</tr>
<tr>
<td>(X_2 = \langle p, 0 \rangle)</td>
<td>0</td>
</tr>
</tbody>
</table>

There is no change in \(Out_{n_3}[X_2]\).
Available Expressions Analysis Using Value Contexts

\[ WL = [X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \]

There is no change in \( Out_{n_3}[X_2] \)

<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_0 = \langle \text{main,0} \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_1 = \langle p,1 \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_2 = \langle p,0 \rangle )</td>
<td>0</td>
</tr>
</tbody>
</table>
Available Expressions Analysis Using Value Contexts

\( WL = [X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m] \)

<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_0 = \langle \text{main},0 \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_1 = \langle \text{p},1 \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_2 = \langle \text{p},0 \rangle )</td>
<td>0</td>
</tr>
</tbody>
</table>

There is no change in \( Out_{n_3}[X_1] \) also.
Available Expressions Analysis Using Value Contexts

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

There is no change in \( Out_{n_3}[X_1] \) also
Available Expressions Analysis Using Value Contexts

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]
Available Expressions Analysis Using Value Contexts

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

At \( E_p \) the values from \( S_p \) and \( n_3 \) are merged for context \( X_1 \)

\[ \text{exitValue}(X_1) \text{ remains 1} \]

<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_0 = \langle \text{main,0} \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_1 = \langle \text{p,1} \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_2 = \langle \text{p,0} \rangle )</td>
<td>0</td>
</tr>
</tbody>
</table>
Available Expressions Analysis Using Value Contexts

\[ WL = [X_1|E_p, X_0|n_1, X_0|E_m] \]

Context | exitValue
---------|----------
\( X_0 = \langle \text{main}, 0 \rangle \) | 1
\( X_1 = \langle p, 1 \rangle \) | 1
\( X_2 = \langle p, 0 \rangle \) | 0

At \( E_p \) the values from \( S_p \) and \( n_3 \) are merged for context \( X_1 \). 
exitValue(\( X_1 \)) remains 1.

Since \( X_1 \) has transition \( X_0 \xrightarrow{C_1} X_1 \), \( Out_{C_1}[X_0] \) becomes 1.
Available Expressions Analysis Using Value Contexts

\[ WL = [X_0|n_1, X_0|E_m] \]

\[ X_0 \xrightarrow{C_1} X_1 \xrightarrow{C_2} X_2 \xrightarrow{C_2} \]

<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_0 = \langle \text{main}, 0 \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_1 = \langle p, 1 \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_2 = \langle p, 0 \rangle )</td>
<td>0</td>
</tr>
</tbody>
</table>
Available Expressions Analysis Using Value Contexts

\[ WL = [X_0|n_1, X_0|E_m] \]

Context | exitValue
---|---
\( X_0 = \langle \text{main}, 0 \rangle \) | 1
\( X_1 = \langle p, 1 \rangle \) | 1
\( X_2 = \langle p, 0 \rangle \) | 0
Available Expressions Analysis Using Value Contexts

\[ \text{WL} = [X_0 | E_m] \]

\[
\begin{align*}
X_0 & \xrightarrow{C_1} X_1 \xrightarrow{C_2} X_2 \xrightarrow{C_2}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_0 = \langle \text{main},0 \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_1 = \langle p,1 \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_2 = \langle p,0 \rangle )</td>
<td>0</td>
</tr>
</tbody>
</table>

\( S_{\text{main}} \)

1. read \( a, b \)  
   \( t := a \ast b \)

\( X_0.0 \)

2. \( S_p \) if \( a == 0 \)

\( X_1.1 \)

3. \( n_2 \)  \( a = a - 1 \)

\( X_1.0 \)

4. \( C_2 \) call \( p \)

\( X_2.0 \)

5. \( n_3 \)  \( t = a \ast b \)

\( X_1.1 \)

6. \( E_p \)

\( X_2.0 \)

7. \( E_{\text{main}} \)

\( X_0.1 \)

8. \( X_0.1 \)

9. \( X_0.1 \)

10. \( X_0.1 \)
Available Expressions Analysis Using Value Contexts

\[ WL = [X_0 | E_m] \]
Available Expressions Analysis Using Value Contexts

\[ WL = [X_0|E_m] \]

\[
\begin{align*}
X_0 &\xrightarrow{C_1} X_1 \\
X_1 &\xrightarrow{C_2} X_2 \\
X_2 &\xrightarrow{C_2} X_1
\end{align*}
\]

<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_0 = \langle \text{main}, 0 \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_1 = \langle p, 1 \rangle )</td>
<td>1</td>
</tr>
<tr>
<td>( X_2 = \langle p, 0 \rangle )</td>
<td>0</td>
</tr>
</tbody>
</table>

\( S_{\text{main}} \)

read \( a, b \)
\( t := a \ast b \)

\( S_p \)

if \( a == 0 \)

\( n_2 \)

\( a = a - 1 \)

\( n_3 \)

\( t = a \ast b \)
Available Expressions Analysis Using Value Contexts

\[ WL = [ ] \]

```
S_{main} =
read a, b
\begin{align*}
t &:= a \times b \\
C_1 &\text{call } p
\end{align*}
```

```
X_0.0
X_0.1
```

```
X_0.0
X_0.1
n_1
print a \times b
```

```
X_0.0
X_0.1
```

```
X_1.1
X_2.0
```

```
X_1.1
X_2.0
```

```
X_1.1
X_2.0
```

```
X_1.0
X_2.0
```

```
X_1.0
X_2.0
```

```
X_1.1
X_2.1
```

```
X_1.1
X_2.1
```

```
X_1.1
X_2.1
```

```
X_1.0
X_2.0
```

```
X_1.0
X_2.0
```

```
X_1.1
X_2.1
```

```
X_1.1
X_2.1
```

```
X_1.1
X_2.1
```

```
X_1.1
X_2.1
```

<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>X_0 = \langle \text{main,0} \rangle</td>
<td>1</td>
</tr>
<tr>
<td>X_1 = \langle p,1 \rangle</td>
<td>1</td>
</tr>
<tr>
<td>X_2 = \langle p,0 \rangle</td>
<td>0</td>
</tr>
</tbody>
</table>

Work list is empty and the analysis is over
### A Trace of Value Context Based Analysis (1)

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Work List</th>
<th>Sel. node</th>
<th>Data flow value</th>
<th>New context</th>
<th>New trans.</th>
<th>exit value</th>
<th>Addition to the work list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>$X_0 = \langle m,0 \rangle$</td>
<td>$X_0.1$</td>
<td>$X_0</td>
<td>S_m, X_0</td>
</tr>
<tr>
<td>2</td>
<td>$X_0</td>
<td>S_m, X_0</td>
<td>C_1, X_0</td>
<td>n_1, X_0</td>
<td>E_m$</td>
<td>$S_m$</td>
<td>$Out_{S_m}[X_0] = 1$</td>
</tr>
<tr>
<td>3</td>
<td>$X_0</td>
<td>C_1, X_0</td>
<td>n_1, X_0</td>
<td>E_m$</td>
<td>$C_1$</td>
<td>$X_1 = \langle p,1 \rangle$</td>
<td>$X_0 \xrightarrow{C_1} X_1$</td>
</tr>
<tr>
<td>4</td>
<td>$X_1</td>
<td>S_p, X_1</td>
<td>n_2, X_1</td>
<td>C_2, X_1</td>
<td>n_3, X_1</td>
<td>E_p, X_0</td>
<td>n_1, X_0</td>
</tr>
<tr>
<td>5</td>
<td>$X_1</td>
<td>n_2, X_1</td>
<td>C_2, X_1</td>
<td>n_3, X_1</td>
<td>E_p, X_0</td>
<td>n_1, X_0</td>
<td>E_m$</td>
</tr>
<tr>
<td>6</td>
<td>$X_1</td>
<td>C_2, X_1</td>
<td>n_3, X_1</td>
<td>E_p, X_0</td>
<td>n_1, X_0</td>
<td>E_m$</td>
<td>$C_2$</td>
</tr>
<tr>
<td>7</td>
<td>$X_2</td>
<td>S_p, X_2</td>
<td>n_2, X_2</td>
<td>C_2, X_2</td>
<td>n_3, X_1</td>
<td>E_p, X_0</td>
<td>n_1, X_0</td>
</tr>
<tr>
<td>S. No.</td>
<td>Work List</td>
<td>Sel. node</td>
<td>Data flow value</td>
<td>New context</td>
<td>New trans.</td>
<td>exit value</td>
<td>Addition to the work list</td>
</tr>
<tr>
<td>-------</td>
<td>---------------------------------------------------------------------------</td>
<td>-----------</td>
<td>-----------------</td>
<td>-------------</td>
<td>------------</td>
<td>------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>8</td>
<td>$X_2 \mid n_2, X_2 \mid C_2, X_2 \mid n_3, X_2 \mid E_p,$ $X_1 \mid n_3, X_1 \mid E_p, X_0 \mid n_1, X_0 \mid E_m$</td>
<td>$n_2$</td>
<td>$Out_{n_2}[X_2] = 0$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>$X_2 \mid C_2, X_2 \mid n_3, X_2 \mid E_p, X_1 \mid n_3,$ $X_1 \mid E_p, X_0 \mid n_1, X_0 \mid E_m$</td>
<td>$C_2$</td>
<td>$Out_{C_2}[X_2] = 1$</td>
<td>$X_2 \rightarrow C_2 \rightarrow X_2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>$X_2 \mid n_3, X_2 \mid E_p, X_1 \mid n_3, X_1 \mid E_p,$ $X_0 \mid n_1, X_0 \mid E_m$</td>
<td>$n_3$</td>
<td>$Out_{n_3}[X_2] = 1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>$X_2 \mid E_p, X_1 \mid n_3, X_1 \mid E_p, X_0 \mid n_1, X_0 \mid E_m$</td>
<td>$E_p$</td>
<td>$Out_{E_p}[X_2] = 0$</td>
<td>$Out_{C_2}[X_2] = 0$</td>
<td>$Out_{C_2}[X_1] = 0$</td>
<td>$X_2.0$</td>
<td>$X_2 \mid n_3$</td>
</tr>
<tr>
<td>12</td>
<td>$X_2 \mid n_3, X_1 \mid n_3, X_1 \mid E_p, X_0 \mid n_1, X_0 \mid E_m$</td>
<td>$n_3$</td>
<td>No change</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>$X_1 \mid n_3, X_1 \mid E_p, X_0 \mid n_1, X_0 \mid E_m$</td>
<td>$n_3$</td>
<td>$Out_{n_3}[X_1] = 1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>$X_1 \mid E_p, X_0 \mid n_1, X_0 \mid E_m$</td>
<td>$E_p$</td>
<td>$Out_{E_p}[X_1] = 1$</td>
<td>$Out_{C_1}[X_0] = 1$</td>
<td></td>
<td></td>
<td>$X_1.1$</td>
</tr>
<tr>
<td>15</td>
<td>$X_0 \mid n_1, X_0 \mid E_m$</td>
<td>$n_1$</td>
<td>$Out_{n_1}[X_0] = 1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>$X_0 \mid E_m$</td>
<td>$E_m$</td>
<td>$Out_{E_m}[X_0] = 1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. int a, b, c;
2. void main()
3. {   c = a * b;
4.     p();
5. }
6. void p()
7. {   if (...) 
8.     {   p();
9.     }  
10.    Is a*b available?
11.   } 
12. }

```
int a, b, c;
void main()
{
    c = a * b;
    p();
}
void p()
{
    if (...)
    {
        p();
    }
    Is a*b available?
    a = a * b;
}
```
Perform interprocedural live variables analysis using value contexts

```
main()
{
    p();
}
p()
{
    while (...)
    {
        printf ("%d\n",a);
        p();
    }
}
```

Observe the change in edges in the transition diagram
Perform interprocedural available expressions analysis using value contexts

```
main()
{
    c = a*b;
    p();
}
```

```
p()
{
    while (a > b)
    {
        p();
        a = a*b;
    }
}
```

Observe the change in edges in the transition diagram
Tutorial Problem #4 for Value Contexts

Perform interprocedural available expressions analysis using value contexts

1. main()
2. {
3.     c = a*b;
4.     p();
5.     a = a*b;
6. }

7. p()
8. {   if (...)
9.     { a = a*b;
10.     p();
11.     }
12. else if (...)
13.     { c = a * b;
14.     p();
15.     c = a;
16.     }
17. else
18.     ; /* ignore */
19. }
Tutorial Problem #5 for Value Contexts

Perform interprocedural live variables analysis using value contexts

main()
{
    a = 5; b = 3;
    c = 7; d = 2;
    p();
    a = a + 2;
    e = c+d;
    d = a*b;
    q();
    print a+c+e;
}

p()
{
    b = 2;
    if (b<d)
        c = a+b;
    else
        q();
    print c+d;
}

q()
{
    a = 1;
    p();
    a = a*b;
}

Context sensitivity: e is live on entry to p but not before its call in main
Result of Tutorial #5

main()
{
    a = 5; b = 3;
    c = 7; d = 2;
    /*{a,d}*/
    p();
    /*{a,b,c,d}*/
    a = a + 2;
    e = c+d;
    /*{a,b,e}*/
    d = a*b;
    /*{d,e}*/
    q();
    /*{a,c,e}*/
    print a+c+e;
}

p()
{
    /*{a,d,e}*/
    b = 2;
    if (b<d)
        /*{a,b,d,e}*/
        c = a+b;
    else
        /*{d,e}*/
        q();
        /*{a,b,c,d,e}*/
        print c+d;
}

q()
{
    /*{d,e}*/
    a = 1;
    /*{a,d,e}*/
    p();
    /*{a,b,c,d,e}*/
    a = a*b;
}
Tutorial Problem #6: Interprocedural Points-to Analysis

main()
{   x = &y;
    z = &x;
    y = &z;
    p(); /* C1 */
}

p()
{   if (...) 
    {       p(); /* C2 */
        x = *x;
    }
}

• Number of distinct call sites in a call chain $K = 2$.
• Number of variables: 3
• Number of distinct points-to pairs: $3 \times 3 = 9$
• $L$ is powerset of all points-to pairs
• $|L| = 2^9$
• Length of the longest call string in Sharir-Pnueli method 
  $2 \times (|L| + 1)^2 = 2^{19} + 2^{11} + 2 = 5, 26, 338$
• All call strings upto this length must be constructed by the Sharir-Pnueli method!
Tutorial Problem #6: Interprocedural Points-to Analysis

main()
{  x = &y;
   z = &x;
   y = &z;
   p(); /* C1 */
}

p()
{  if (...) {
     { p(); /* C2 */
        x = *x;
     }
  }
}

Value contexts method requires only three contexts as shown below in the transition diagram

$X_0 \xrightarrow{C_1} X_1$ $C_2$
Equivalence of The Two Methods

- For non-recursive programs, equivalence is obvious
- For recursive program, we prove equivalence using staircase diagrams
Let $\sigma_c$ and $\sigma_r$ be the call site sequences for RCS and RRS

- $\sigma_c \equiv c_j c_r c_k c_p c_i c_q$ (flow function $f$)
- $\sigma_r \equiv r_q r_i r_p r_k r_r r_j$ (flow function $g$)

Assume that we allow upto $m$ occurrences of $\sigma_c$
Staircase Diagram of Computation along Recursive Paths

Traversing RCS m times

\[ x_1 = f(x_0) \]

Data flow value at \( C_q \)
Staircase Diagram of Computation along Recursive Paths

*Traversing RCS m times*

\[ x_2 = f^2(x_0) \]

*Data flow value at \( C_q \)*
Staircase Diagram of Computation along Recursive Paths

Traversing RCS $m$ times

$x_i = f^i(x_0)$

Data flow value at $C_q$
Staircase Diagram of Computation along Recursive Paths

Traversing RCS \(m\) times

\[
x_i = f^i(x_0)
\]

Data flow value at \(C_q\)
**Staircase Diagram of Computation along Recursive Paths**

*Traversing RCS* \( m \) *times*

\[
x_i = f^i(x_0)
\]

*Data flow value at* \( C_q \)
Staircase Diagram of Computation along Recursive Paths

Traversing RCS $m$ times

$x_i = f^i(x_0)$

Data flow value at $C_q$
Traversing RCS $m$ times

$x_i = f^i(x_0)$

Data flow value at $C_q$
Staircase Diagram of Computation along Recursive Paths

Traversing RCS $m$ times

$$x_i = f^i(x_0)$$

Data flow value at $C_q$

$$z_m = h(x_m)$$

Data flow value at $R_q$
Staircase Diagram of Computation along Recursive Paths

**Traversing RCS** $m$ times

**Traversing RRS** $m$ times

$x_i = f^i(x_0)$

Data flow value at $C_q$

$z_{m-1} = h(x_{m-1}) \cap g(z_m)$

Data flow value at $R_q$
Staircase Diagram of Computation along Recursive Paths

\[ x_i = f^i(x_0) \]

Data flow value at \( C_q \)

\[ z_{m-2} = h(x_{m-2}) \cap g(z_{m-1}) \]

Data flow value at \( R_q \)
Staircase Diagram of Computation along Recursive Paths

Traversing RCS $m$ times

Traversing RRS $m$ times

$$x_i = f^i(x_0)$$

Data flow value at $C_q$

$$z_{m-j} = h(x_{m-j}) \cap g(z_{m-j+1})$$

Data flow value at $R_q$
Staircase Diagram of Computation along Recursive Paths

Traversing RCS $m$ times

Traversing RRS $m$ times

$x_i = f^i(x_0)$

Data flow value at $C_q$

$z_{m-j} = h(x_{m-j}) \cap g(z_{m-j+1})$

Data flow value at $R_q$
Staircase Diagram of Computation along Recursive Paths

Traversing RCS $m$ times

Traversing RRS $m$ times

$x_i = f^i(x_0)$

Data flow value at $C_q$

$z_{m-j} = h(x_{m-j}) \cap g(z_{m-j+1})$

Data flow value at $R_q$
Staircase Diagram of Computation along Recursive Paths

**Traversing RCS m times**

\[ x_i = f_i(x_0) \]

*Data flow value at C_q*

**Traversing RRS m times**

\[ z_{m-j} = h(x_{m-j}) \cap g(z_{m-j+1}) \]

*Data flow value at R_q*
Staircase Diagram of Computation along Recursive Paths

**Traversing RCS m times**

\[ x_i = f^i(x_0) \]

*Data flow value at C_q*

**Traversing RRS m times**

\[ z_{m-j} = h(x_{m-j}) \cap g(z_{m-j+1}) \]

*Data flow value at R_q*
• $n > 0$ is the fixed point closure bound of $h : L \mapsto L$ if it is the smallest number such that

$$\forall x \in L, \ h^{n+1}(x) = h^n(x)$$
Computation of Data Flow Values along Recursive Paths
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

$\omega$

$X_0 \xrightarrow{\sigma_c} X_1 \xrightarrow{\sigma_c} \cdots \xrightarrow{\sigma_c} X_\omega$
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

$\omega + 1$

$\omega$

$\sigma_c$

$x_0$

$\sigma_c$

$f$

$\sigma_c$

$x_\omega$

$\sigma_c$

$x_\omega$

$\sigma_c$

$f$

$\sigma_c$

$X_0$

$\sigma_c$

$X_1$

$\sigma_c$

$\cdots$

$\sigma_c$

$X_\omega$

$\sigma_c$

Oct 2013 IIT Bombay
Computation of Data Flow Values along Recursive Paths

FP closure bound of \( f \)

\[
\begin{align*}
\sigma_c \cdot X_0 & \quad \sigma_c \cdot X_1 & \quad \ldots & \quad \sigma_c \cdot X_\omega \\
\omega & \quad \omega + 1 & \quad \omega + 2
\end{align*}
\]
Computation of Data Flow Values along Recursive Paths

FP closure bound of \( f \)

\[
\begin{align*}
&\omega + 2 \\
&\omega + 1 \\
&\omega \\
&X_0 \\
&X_1 \\
&\cdots \\
\end{align*}
\]
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

$m-1$

$\ldots$

$\omega + 2$

$\omega + 1$

$\omega$

$X_0$ - $X_1$ - $X_\omega$
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

Identical data flow values
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

Identical data flow values

$X_0 \xrightarrow{\sigma_c} X_1 \xrightarrow{\sigma_c} \ldots \xrightarrow{\sigma_c} X_0$
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

Identical data flow values

$m$

$m-1$

$\omega+2$

$\omega+1$

$\omega$

$X_0 \xrightarrow{\sigma_c} X_1 \xrightarrow{\sigma_c} \ldots \xrightarrow{\sigma_c} X_\omega$

Oct 2013 IIT Bombay
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

FP closure bound of $g$

Identical data flow values

Oct 2013
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

Identical data flow values

FP closure bound of $g$

Oct 2013 IIT Bombay
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

Identical data flow values

FP closure bound of $g$

Oct 2013 IIT Bombay
Computation of Data Flow Values along Recursive Paths

Identical data flow values

FP closure bound of $f$

FP closure bound of $g$

Oct 2013
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

FP closure bound of $g$

$X_0 \xrightarrow{\sigma_c} X_1 \xrightarrow{\sigma_c} \cdots \xrightarrow{\sigma_c} X_\omega$
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

FP closure bound of $g$
Bounding the Call String Length Using Data Flow Values

FP closure bound of $f$

FP closure bound of $g$

Identical data flow values

Oct 2013 IIT Bombay
Bounding the Call String Length Using Data Flow Values

FP closure bound of $f$

$\omega \rightarrow X_0 \rightarrow X_1 \rightarrow \cdots \rightarrow X_\omega$

$\sigma_c \rightarrow f \rightarrow \sigma_c \rightarrow h \rightarrow \sigma_c \rightarrow g \rightarrow \sigma_r \rightarrow z_0$

Oct 2013
Bounding the Call String Length Using Data Flow Values
Bounding the Call String Length Using Data Flow Values

FP closure bound of $f$
Bounding the Call String Length Using Data Flow Values

FP closure bound of \( f \)

\[ \omega \rightarrow X_\omega \rightarrow Z_{m-\eta} \rightarrow \cdots \rightarrow Z_0 \]

\[ h \]

\[ \sigma_c \]

\[ X_0 \rightarrow X_1 \rightarrow \cdots \rightarrow X_\omega \]
Bounding the Call String Length Using Data Flow Values

FP closure bound of $f$
Bounding the Call String Length Using Data Flow Values

- This amounts to simulating all call strings that would have otherwise been constructed while traversing RCS
- It can also be seen as virtually climbing up the steps in RRS as much as needed and then climbing down
- This is possible only because all these call strings would have the same data flow value associated with them
Bounding the Call String Length Using Data Flow Values

- This amounts to simulating all call strings that would have otherwise been constructed while traversing RCS.
- It can also be seen as virtually climbing up the steps in RRS as much as needed and then climbing down.
- This is possible only because all these call strings would have the same data flow value associated with them.
Bounding the Call String Length Using Data Flow Values

- This amounts to simulating all call strings that would have otherwise been constructed while traversing RCS.
- It can also be seen as virtually climbing up the steps in RRS as much as needed and then climbing down.
- This is possible only because all these call strings would have the same data flow value associated with them.
Worst Case Length Bound

- Consider a call string $\sigma = \ldots (c_i)_1 \ldots (c_i)_2 \ldots (c_i)_3 \ldots (c_i)_j \ldots$
  where $(c_i)_j$ denotes the $j^{th}$ occurrence of $c_i$
  Let $j \geq |L| + 1$
  Let $C_i$ call procedure $p$
Worst Case Length Bound

- Consider a call string $\sigma = \ldots (c_i)_1 \ldots (c_i)_2 \ldots (c_i)_3 \ldots (c_i)_j \ldots$
  where $(c_i)_j$ denotes the $j^{th}$ occurrence of $c_i$
  Let $j \geq |L| + 1$
  Let $C_i$ call procedure $p$
- All call string ending with $C_i$ reach entry $S_p$
Worst Case Length Bound

- Consider a call string \( \sigma = \ldots (c_i)_1 \ldots (c_i)_2 \ldots (c_i)_3 \ldots (c_i)_j \ldots \)
  where \((c_i)_j\) denotes the \(j^{th}\) occurrence of \(c_i\)
  Let \(j \geq |L| + 1\)
  Let \(C_i\) call procedure \(p\)

- All call string ending with \(C_i\) reach entry \(S_p\)

- Since only \(|L|\) distinct values are possible, by the pigeon hole principle, at least two prefixes ending with \(C_i\) will carry the same data flow value to \(S_p\).
Worst Case Length Bound

- Consider a call string \( \sigma = \ldots(c_i)_1 \ldots(c_i)_2 \ldots(c_i)_3 \ldots(c_i)_j \ldots \)
  where \((c_i)_j\) denotes the \(j^{th}\) occurrence of \(c_i\)
  Let \(j \geq |L| + 1\)
  Let \(C_i\) call procedure \(p\)

- All call string ending with \(C_i\) reach entry \(S_p\)

- Since only \(|L|\) distinct values are possible, by the pigeon hole principle, at least two prefixes ending with \(C_i\) will carry the same data flow value to \(S_p\).
  - All longer call strings will have the same data flow values and would be represented by the same value context \(j \leq |L|\)
Worst Case Length Bound

- Consider a call string $\sigma = \ldots (c_i)_1 \ldots (c_i)_2 \ldots (c_i)_3 \ldots (c_i)_j \ldots$
  where $(c_i)_j$ denotes the $j^{th}$ occurrence of $c_i$
  Let $j \geq |L| + 1$
  Let $C_i$ call procedure $p$

- All call string ending with $C_i$ reach entry $S_p$

- Since only $|L|$ distinct values are possible, by the pigeon hole principle, at least two prefixes ending with $C_i$ will carry the same data flow value to $S_p$.
  - All longer call strings will have the same data flow values and would be represented by the same value context $j \leq |L|$

- Worst case (logical) length in the proposed variant $= K \times (|L|)$
Worst Case Length Bound

- Consider a call string \( \sigma = \ldots (c_i)_1 \ldots (c_i)_2 \ldots (c_i)_3 \ldots (c_i)_j \ldots \)
  where \( (c_i)_j \) denotes the \( j^{th} \) occurrence of \( c_i \)
- Let \( j \geq |L| + 1 \)
- Let \( C_i \) call procedure \( p \)
- All call string ending with \( C_i \) reach entry \( S_p \)
- Since only \( |L| \) distinct values are possible, by the pigeon hole principle, at least two prefixes ending with \( C_i \) will carry the same data flow value to \( S_p \).
  - All longer call strings will have the same data flow values and would be represented by the same value context \( j \leq |L| \)
- Worst case (logical) length in the proposed variant = \( K \times (|L|) \)
Worst Case Length Bound

- Consider a call string $\sigma = \ldots (c_i)_1 \ldots (c_i)_2 \ldots (c_i)_3 \ldots (c_i)_j \ldots$
  where $(c_i)_j$ denotes the $j^{th}$ occurrence of $c_i$
  Let $j \geq |L| + 1$
  Let $C_i$ call procedure $p$

- All call string ending with $C_i$ reach entry $S_p$

- Since only $|L|$ distinct values are possible, by the pigeon hole principle, at least two prefixes ending with $C_i$ will carry the same data flow value to $S_p$.
  - All longer call strings will have the same data flow values and would be represented by the same value context $j \leq |L|$.

- Worst case (logical) length in the proposed variant $= K \times (|L|)$
Worst Case Length Bound

• Consider a call string \( \sigma = \ldots (c_i)_1 \ldots (c_i)_2 \ldots (c_i)_3 \ldots (c_i)_{j} \ldots \) where \((c_i)_{j}\) denotes the \(j^{th}\) occurrence of \(c_i\)

Let \( j \geq |L| + 1 \)

Let \( C_i \) call procedure \( p \)

• All call string ending with \( C_i \) reach entry \( S_p \)

• Since only \(|L|\) distinct values are possible, by the pigeon hole principle, at least two prefixes ending with \( C_i \) will carry the same data flow value to \( S_p \).

  ▶ All longer call strings will have the same data flow values and would be represented by the same value context \( j \leq |L| \)

• Worst case (logical) length in the proposed variant = \( K \times (|L|) \)

• Original required length = \( K \times (|L| + 1)^2 \)
## Reaching Definitions Analysis in GCC 4.0

<table>
<thead>
<tr>
<th>Program</th>
<th>LoC</th>
<th>#F</th>
<th>#C</th>
<th>3K length bound</th>
<th>Proposed Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K</td>
<td>#CS</td>
<td>Max</td>
<td>Time</td>
<td>#CS</td>
</tr>
<tr>
<td>hanoi</td>
<td>33</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>100000+</td>
</tr>
<tr>
<td>bit_gray</td>
<td>53</td>
<td>5</td>
<td>11</td>
<td>7</td>
<td>100000+</td>
</tr>
<tr>
<td>analyzer</td>
<td>288</td>
<td>14</td>
<td>20</td>
<td>2</td>
<td>21</td>
</tr>
<tr>
<td>distray</td>
<td>331</td>
<td>9</td>
<td>21</td>
<td>6</td>
<td>96</td>
</tr>
<tr>
<td>mason</td>
<td>350</td>
<td>9</td>
<td>13</td>
<td>8</td>
<td>100000+</td>
</tr>
<tr>
<td>fourinarow</td>
<td>676</td>
<td>17</td>
<td>45</td>
<td>5</td>
<td>510</td>
</tr>
<tr>
<td>sim</td>
<td>1146</td>
<td>13</td>
<td>45</td>
<td>8</td>
<td>100000+</td>
</tr>
<tr>
<td>181_mcf</td>
<td>1299</td>
<td>17</td>
<td>24</td>
<td>6</td>
<td>32789</td>
</tr>
<tr>
<td>256_bzip2</td>
<td>3320</td>
<td>63</td>
<td>198</td>
<td>7</td>
<td>492</td>
</tr>
</tbody>
</table>

- LoC is the number of lines of code,
- #F is the number of procedures,
- #C is the number of call sites,
- #CS is the number of call strings
- Max denotes the maximum number of call strings reaching any node.
- Analysis time is in milliseconds.

(Implementation was carried out by Seema Ravandale.)
Some Observations

• Compromising on precision may not be necessary for efficiency.

• Separating the necessary information from redundant information is much more significant.

• Data flow propagation in real programs seems to involve only a small subset of all possible values. Much fewer changes than the theoretically possible worst case number of changes.

• A precise modelling of the process of analysis is often an eye opener.
Some Observations

- Compromising on precision may not be necessary for efficiency.
- Separating the necessary information from redundant information is much more significant.
- Data flow propagation in real programs seems to involve only a small subset of all possible values. Much fewer changes than the theoretically possible worst case number of changes.
- A precise modelling of the process of analysis is often an eye opener.

\[ \# \text{ distinct tagged values} = \min (\# \text{ actual contexts}, \# \text{ actual data flow values}) \]
Part 6

Extra Slides
Classical Call String Length

- **Notation**
  - $IVP(n, m)$: Interprocedurally valid path from block $n$ to block $m$
  - $CS(\rho)$: Number of call nodes in $\rho$ that do not have the matching return node in $\rho$  
    (length of the call string representing $IVP(n, m)$)

- **Claim**
  Let $M = K \cdot (|L| + 1)^2$ where $K$ is the number of distinct call sites in any call chain
  Then, for any $\rho = IVP(S_{main}, m)$ such that
  $CS(\rho) > M$,  
  $\exists \rho' = IVP(S_{main}, m)$ such that
  $CS(\rho') \leq M$, and $f_\rho(BI) = f_{\rho'}(BI)$

  $\Rightarrow \rho$, the longer path, is redundant for data flow analysis
Sharir-Pnueli [1981]

- Consider the smallest prefix $\rho_0$ of $\rho$ such that $CS(\rho_0) > M$
- Consider a triple $\langle c_i, \alpha_i, \beta_i \rangle$ where
  - $\alpha_i$ is the data flow value reaching call node $C_i$ along $\rho$ and
  - $\beta_i$ is the data flow value reaching the corresponding return node $R_i$ along $\rho$

  If $R_i$ is not in $\rho$, then $\beta_i = \Omega$ (undefined)
Classical Call String Length

$M$

$\rho_0$

$\rho$
Classical Call String Length

\[ \langle c_i, \alpha_i, \beta_i \rangle \]

\( M \)

\( \rho_0 \)

\( \rho \)
Classical Call String Length

\[ \langle c_j, \alpha_j, \Omega \rangle \]

\[ \rho_0 \]

\[ \rho \]

\[ M \]
Classical Call String Length

- Number of distinct triples $\langle c_i, \alpha_i, \beta_i \rangle$ is $M = K \cdot (|L| + 1)^2$. 
Number of distinct triples \( \langle c_i, \alpha_i, \beta_i \rangle \) is \( M = K \cdot (|L| + 1)^2 \).

There are at least two calls from the same call site that have the same effect on data flow values.
Classical Call String Length

When $\beta_i$ is not $\Omega$
Classical Call String Length

When $\beta_i$ is not $\Omega$
Classical Call String Length

When $\beta_i$ is not $\Omega$
When $\beta_i$ is $\Omega$
Classical Call String Length

When \( \beta_i \) is \( \Omega \)
Classical Call String Length

When \( \beta_i \) is \( \Omega \)
Tighter Bound for Bit Vector Frameworks

- $\hat{L}$ is $\{0, 1\}$, $L$ is $\{0, 1\}^m$
- $\hat{\sqcap}$ is either boolean AND or boolean OR
- $\hat{\top}$ and $\hat{\bot}$ are 0 or 1 depending on $\hat{\sqcap}$.
- $\hat{h}$ is a bit function and could be one of the following:

<table>
<thead>
<tr>
<th>Raise</th>
<th>Lower</th>
<th>Propagate</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{\top}$</td>
<td>$\hat{\top}$</td>
<td>$\hat{\top}$</td>
</tr>
<tr>
<td>$\hat{\bot}$</td>
<td>$\hat{\bot}$</td>
<td>$\hat{\bot}$</td>
</tr>
<tr>
<td>$\hat{\top}$</td>
<td>$\hat{\top}$</td>
<td>$\hat{\top}$</td>
</tr>
<tr>
<td>$\hat{\bot}$</td>
<td>$\hat{\bot}$</td>
<td>$\hat{\bot}$</td>
</tr>
<tr>
<td>$\hat{\top}$</td>
<td>$\hat{\bot}$</td>
<td>$\hat{\bot}$</td>
</tr>
<tr>
<td>$\hat{\bot}$</td>
<td>$\hat{\top}$</td>
<td>$\hat{\top}$</td>
</tr>
</tbody>
</table>
Tighter Bound for Bit Vector Frameworks

Karkare Khedker 2007

- Validity constraints are imposed by the presence of return nodes
- For every cyclic path consisting on Propagate functions, there exists an acyclic path consisting of Propagate functions
- Source of information is a Raise or Lower function
- Target is a point reachable by a series of Propagate functions
- Identifies interesting path segments that we need to consider for determining a sufficient set of call strings
Relevant Path Segments for Tighter Bound for Bit Vector Frameworks
Relevant Path Segments for Tighter Bound for Bit Vector Frameworks
Relevant Path Segments for Higher Bound for Bit Vector Frameworks

- All paths from $C_i$ to $R_i$ are abstracted away when a call node $C_j$ is reached after $R_i$
- Consider maximal interprocedurally valid paths in which there is no path from a return node to a call node
Relevant Path Segments for Tighter Bound for Bit Vector Frameworks

Consider all four combinations

- Case A: Source is a call node and target is a call node
- Case B: Source is a call node and target is a return node
- Case C: Source is a return node and target is also a return node
- Case D: Source is a return node and target is a call node: Not relevant
Tighter Length for Bit Vector Frameworks

Case A:

Source is a call node and target is also a call node \( P(Entry \leadsto C_S \leadsto C_T) \)

- No return node, no validity constraints
- Paths \( P(Entry \leadsto C_S) \) and Paths \( P(C_S \leadsto C_T) \) can be acyclic
- A call node may be common to both segments
- At most 2 occurrences of a call site
Tighter Length for Bit Vector Frameworks

Case B: 
**Source** is a call node $C_S$ and **target** is some return node $R_T$
Tighter Length for Bit Vector Frameworks

Case B:
The source is a call node \( C_S \) and the target is some return node \( R_T \)

- \( P(\text{Entry} \leadsto C_S \leadsto C_T \leadsto R_T) \)

- \( P(\text{Entry} \leadsto C_T \leadsto C_S \leadsto R_S \leadsto R_T) \)
Tighter Length for Bit Vector Frameworks

Case B:

Source is a call node $C_S$ and target is some return node $R_T$

- $P(Entry \leadsto C_S \leadsto C_T \leadsto R_T)$

  - Call strings are derived from the paths $P(Entry \leadsto C_S \leadsto C_T \leadsto C_L)$ where $C_L$ is the last call node.
Tighter Length for Bit Vector Frameworks

Case B:

Source is a call node $C_S$ and target is some return node $R_T$

- $P(Entry \leadsto C_S \leadsto CT \leadsto R_T)$
  - Call strings are derived from the paths $P(Entry \leadsto C_S \leadsto CT \leadsto CL)$ where $CL$ is the last call node.
  - Thus there are three acyclic segments $P(Entry \leadsto CS), P(CS \leadsto CT), \text{ and } P(CT \leadsto CL)$
Tighter Length for Bit Vector Frameworks

Case B: 
**Source** is a call node $C_S$ and **target** is some return node $R_T$

- $P(Entry \leadsto C_S \leadsto C_T \leadsto R_T)$
  - Call strings are derived from the paths $P(Entry \leadsto C_S \leadsto C_T \leadsto C_L)$ where $C_L$ is the last call node.
  - Thus there are three acyclic segments $P(Entry \leadsto C_S), P(C_S \leadsto C_T), \text{ and } P(C_T \leadsto C_L)$
  - A call node may be shared in all three
    $\Rightarrow$ At most 3 occurrences of a call site
Case B: Source is a call node $C_S$ and target is some return node $R_T$

- $P(Entry \leadsto C_S \leadsto \overline{C_T} \leadsto R_T)$
  - Call strings are derived from the paths $P(Entry \leadsto C_S \leadsto C_T \leadsto C_L)$ where $C_L$ is the last call node
  - Thus there are three acyclic segments $P(Entry \leadsto C_S), P(C_S \leadsto C_T), \text{and } P(C_T \leadsto C_L)$
  - A call node may be shared in all three
    $\Rightarrow$ At most 3 occurrences of a call site

- $P(Entry \leadsto \overline{C_T} \leadsto \overline{C_S} \leadsto R_S \leadsto R_T)$
  - $C_T$ is required because of validity constraints
Tighter Length for Bit Vector Frameworks

Case B:

Source is a call node $C_S$ and target is some return node $R_T$

- $P(Entry \rightsquigarrow C_S \rightsquigarrow C_T \rightsquigarrow R_T)$
  - Call strings are derived from the paths $P(Entry \rightsquigarrow C_S \rightsquigarrow C_T \rightsquigarrow C_L)$ where $C_L$ is the last call node.
  - Thus there are three acyclic segments $P(Entry \rightsquigarrow C_S)$, $P(C_S \rightsquigarrow C_T)$, and $P(C_T \rightsquigarrow C_L)$
  - A call node may be shared in all three
    $\Rightarrow$ At most 3 occurrences of a call site

- $P(Entry \rightsquigarrow C_T \rightsquigarrow C_S \rightsquigarrow R_S \rightsquigarrow R_T)$
  - $C_T$ is required because of validity constraints
  - Call strings are derived from the paths $P(Entry \rightsquigarrow C_T \rightsquigarrow C_S \rightsquigarrow C_L)$ where $C_L$ is the last call node
Tighter Length for Bit Vector Frameworks

Case B:

Source is a call node $C_S$ and target is some return node $R_T$

- $P(Entry \leadsto C_S \leadsto C_T \leadsto R_T)$
  - Call strings are derived from the paths $P(Entry \leadsto C_S \leadsto C_T \leadsto C_L)$ where $C_L$ is the last call node
  - Thus there are three acyclic segments $P(Entry \leadsto C_S), P(C_S \leadsto C_T), \text{ and } P(C_T \leadsto C_L)$
  - A call node may be shared in all three
    ⇒ At most 3 occurrences of a call site

- $P(Entry \leadsto C_T \leadsto C_S \leadsto R_S \leadsto R_T)$
  - $C_T$ is required because of validity constraints
  - Call strings are derived from the paths $P(Entry \leadsto C_T \leadsto C_S \leadsto C_L)$ where $C_L$ is the last call node
  - Again, there are three acyclic segments and at most 3 occurrences of a call site
Case C:

**Source** is a return node \( R_S \) and **target** is also some return node \( R_T \)

- \( P(Entry \leadsto C_T \leadsto C_S \leadsto R_S \leadsto R_T) \)
- \( C_T \) and \( C_S \) are required because of validity constraints
- Call strings are derived from the paths \( P(Entry \leadsto C_T \leadsto C_S \leadsto C_L) \) where \( C_L \) is the last call node
- Again, there are three acyclic segments and at most 3 occurrences of a call site
Classical Approximate Approach

- Maintain call string suffixes of upto a given length $m$
Classical Approximate Approach

- Maintain call string suffixes of up to a given length $m$

Call string of length $m - 1$

$$\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_{m-1}} \rangle$$

$$R_a$$
Classical Approximate Approach

- Maintain call string suffixes of up to a given length $m$

\[ \langle C_{i_1} \cdot C_{i_2} \ldots C_{i_{m-1}} \mid x \rangle \]

\[ \langle C_{i_1} \cdot C_{i_2} \ldots C_{i_{m-1}} \cdot C_a \mid x \rangle \]
Classical Approximate Approach

- Maintain call string suffixes of up to a given length $m$

- Call string of length $m - 1$

\[ \langle C_i \cdot C_{i_2} \ldots C_{i_m-1} \rangle \]

- Call string of length $m$

\[ \langle C_i \cdot C_{i_2} \ldots C_{i_m-1} \cdot C_a \mid x \rangle \]

\[ \langle C_i \cdot C_{i_2} \ldots C_{i_m-1} \cdot C_a \mid y \rangle \]
Classical Approximate Approach

- Maintain call string suffixes of up to a given length $m$

Call string of length $m - 1$

\[
\langle C_i \cdot C_{i_2} \ldots C_{i_{m-1}} | x \rangle
\]

\[
\langle C_i \cdot C_{i_2} \ldots C_{i_{m-1}} \cdot C_a | x \rangle
\]

Call string of length $m$

\[
\langle C_i \cdot C_{i_2} \ldots C_{i_{m-1}} \cdot C_a | y \rangle
\]

\[
\langle C_i \cdot C_{i_2} \ldots C_{i_{m-1}} | y \rangle
\]
Classical Approximate Approach

- Maintain call string suffixes of up to a given length $m$

Call string of length $m$: $\langle C_i_1 \cdot C_i_2 \ldots C_i_m \mid x \rangle$

- $C_a$
- $R_a$
Classical Approximate Approach

- Maintain call string suffixes of upto a given length $m$

- Call string of length $m$

  $\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_m} | x \rangle$

- Call string of length $m$

  $\langle C_{i_2} \ldots C_{i_m} \cdot C_a | x \rangle$

  (First call site $c_{i_1}$ removed from incoming call string and call site $c_a$ attached)

- $R_a$
Classical Approximate Approach

- Maintain call string suffixes of upto a given length $m$

\[ \langle C_{i_1} \cdot C_{i_2} \ldots C_{i_m} | x \rangle \]

Call string of length $m$

\[ \langle C_{i_2} \ldots C_{i_m} \cdot C_a | x \rangle \]

(First call site $c_{i_1}$ removed from incoming call string and call site $c_a$ attached)

\[ \langle C_{i_2} \ldots C_{i_m} \cdot C_a | y \rangle \]

\[ R_a \]

Oct 2013 IIT Bombay
Classical Approximate Approach

- Maintain call string suffixes of upto a given length $m$

\[ \langle C_i \cdot C_{i_2} \ldots C_{i_m} | x \rangle \]

\[ \langle C_{i_2} \ldots C_{i_m} \cdot C_a | x \rangle \]

\[ \langle C_{i_2} \ldots C_{i_m} \cdot C_a | y \rangle \]

\[ \langle C_i \cdot C_{i_2} \ldots C_{i_m} | y \rangle \]
Classical Approximate Approach

- Maintain call string suffixes of up to a given length $m$

$$\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_m} | x_1 \rangle$$
Classical Approximate Approach

- Maintain call string suffixes of up to a given length $m$

$$\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_m} \mid x_1 \rangle \quad \langle C_{f_1} \cdot C_{i_2} \ldots C_{i_m} \mid x_2 \rangle$$
Classical Approximate Approach

- Maintain call string suffixes of up to a given length $m$

\[
\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_m} \mid x_1 \rangle \quad \langle C_{j_1} \cdot C_{j_2} \ldots C_{j_m} \mid x_2 \rangle
\]

\[
\langle C_{i_2} \cdot C_{i_3} \ldots C_{i_m} \cdot C_a \mid x_1 \sqcap x_2 \rangle
\]

$R_a$
Classical Approximate Approach

- Maintain call string suffixes of up to a given length $m$

\[
\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_m} \mid x_1 \rangle \quad \langle C_{f_1} \cdot C_{i_2} \ldots C_{i_m} \mid x_2 \rangle
\]

\[
\langle C_{i_2} \cdot C_{i_3} \ldots C_{i_m} \cdot C_a \mid x_1 \sqcap x_2 \rangle
\]

\[
\langle C_{i_2} \cdot C_{i_3} \ldots C_{i_m} \cdot C_a \mid y \rangle
\]

- $C_a$

- $R_a$
Classical Approximate Approach

- Maintain call string suffixes of upto a given length $m$

$$\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_m} \mid x_1 \rangle \quad \langle C_{j_1} \cdot C_{i_2} \ldots C_{i_m} \mid x_2 \rangle$$

$$\langle C_{i_2} \cdot C_{i_3} \ldots C_{i_m} \cdot C_a \mid x_1 \sqcap x_2 \rangle$$

$$\langle C_{i_2} \cdot C_{i_3} \ldots C_{i_m} \cdot C_a \mid y \rangle$$

$$\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_m} \mid y \rangle \quad \langle C_{j_1} \cdot C_{i_2} \ldots C_{i_m} \mid y \rangle$$
Classical Approximate Approach

- Maintain call string suffixes of upto a given length $m$

\[
\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_m} \mid x_1 \rangle \quad \langle C_{j_1} \cdot C_{i_2} \ldots C_{i_m} \mid x_2 \rangle
\]

\[
\langle C_{i_2} \cdot C_{i_3} \ldots C_{i_m} \cdot C_a \mid x_1 \sqcap x_2 \rangle
\]

\[
\langle C_{i_2} \cdot C_{i_3} \ldots C_{i_m} \cdot C_a \mid y \rangle
\]

- Practical choices of $m$ have been 1 or 2
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

\[ \langle C_b \mid x_1 \rangle \]
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

\[
\langle C_b \mid x_1 \rangle \quad \langle C_b \cdot C_a \mid x_1 \rangle
\]

\[
C_a
\]

\[
R_a
\]
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

\[ \langle C_b | x_1 \rangle \xrightarrow{C_a} \langle C_b \cdot C_a | x_2 \rangle \]

\[ \langle C_b \cdot C_a | x_1 \rangle \xrightarrow{R_a} \]

Oct 2013
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

\[
\langle C_b \mid x_1 \rangle, \langle C_b \cdot C_a \mid x_2 \rangle
\]

\[
\langle C_b \cdot C_a \mid x_1 \rangle, \langle C_a \cdot C_a \mid x_2 \rangle
\]
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

\[
\langle C_b \mid x_1 \rangle, \langle C_b \cdot C_a \mid x_2 \rangle, \langle C_a \cdot C_a \mid x_3 \rangle
\]
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

\[
\langle C_b \mid x_1 \rangle, \langle C_b \cdot C_a \mid x_2 \rangle, \langle C_a \cdot C_a \mid x_3 \rangle
\]

\[
\langle C_b \cdot C_a \mid x_1 \rangle, \langle C_a \cdot C_a \mid x_2 \sqcap x_3 \rangle
\]

\[
R_a
\]
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

\[
\langle C_b \mid x_1 \rangle, \quad \langle C_b \cdot C_a \mid x_2 \rangle, \quad \langle C_a \cdot C_a \mid x_4 \rangle
\]

\[
\langle C_b \cdot C_a \mid x_1 \rangle, \quad \langle C_a \cdot C_a \mid x_2 \cap x_3 \rangle
\]

$R_a$
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

\[
\langle C_b \mid x_1 \rangle, \quad \langle C_b \cdot C_a \mid x_2 \rangle, \quad \langle C_a \cdot C_a \mid x_4 \rangle
\]

\[
\langle C_b \cdot C_a \mid x_1 \rangle, \quad \langle C_a \cdot C_a \mid x_5 \rangle
\]
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

\[
\langle C_b \mid x_1 \rangle, \langle C_b \cdot C_a \mid x_2 \rangle, \langle C_a \cdot C_a \mid x_4 \rangle
\]

\[
\langle C_b \cdot C_a \mid x_1 \rangle, \langle C_a \cdot C_a \mid x_5 \rangle
\]

\[
\langle C_b \cdot C_a \mid y_1 \rangle, \langle C_a \cdot C_a \mid y_2 \rangle
\]

\[
R_a
\]
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

$$
\langle C_b \mid x_1 \rangle, \langle C_b \cdot C_a \mid x_2 \rangle, \langle C_a \cdot C_a \mid x_4 \rangle
\vspace{1cm}
\begin{array}{c}
\langle C_b \cdot C_a \mid x_1 \rangle, \langle C_a \cdot C_a \mid x_5 \rangle \\
\end{array}
\vspace{1cm}
\begin{array}{c}
\langle C_b \cdot C_a \mid y_1 \rangle, \langle C_a \cdot C_a \mid y_2 \rangle \\
\end{array}
\vspace{1cm}
\begin{array}{c}
\langle C_b \mid y_1 \rangle, \langle C_b \cdot C_a \mid y_2 \rangle, \langle C_a \cdot C_a \mid y_2 \rangle \\
\end{array}
$$
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

\[
\langle C_b \mid x_1 \rangle, \langle C_b \cdot C_a \mid x_2 \rangle, \langle C_a \cdot C_a \mid x_4 \rangle
\]

\[
\langle C_b \cdot C_a \mid x_5 \rangle
\]

\[
\langle C_b \cdot C_a \mid y_1 \rangle, \langle C_a \cdot C_a \mid y_2 \rangle
\]

\[
\langle C_b \mid y_1 \rangle, \langle C_b \cdot C_a \mid y_2 \rangle, \langle C_a \cdot C_a \mid y_2 \rangle
\]