Interprocedural Data Flow Analysis

Uday P. Khedker

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

October 2009
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
• Modified call strings approach
Part 3

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

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

\[ a + b \]

\[ \text{Call } p \]

\[ \text{End}_p \]

\[ \text{End}_\text{main} \]

\[ \text{Start}_\text{main} \]

\[ \text{Start}_p \]

\[ n_1 \quad d = a + b \]

\[ \text{Call } p \]

\[ \text{End}_q \]

\[ \text{End}_\text{q} \]

\[ \text{Start}_q \]

\[ a = 1 \]

\[ n_2 \]

\[ n_3 \]

\[ n_4 \]

Supergraphs of procedures

Call multi-graph

Oct 2009
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: Supergraph
Program Representation for Interprocedural Data Flow Analysis: Supergraph

```
Start_{main}
```

```
a + b
```

```
C_1 Call p
```

```
R_1
```

```
End_{main}
```

```
Start_p
```

```
a
```

```
b
```

```
C_2 Call q
```

```
R_2
```

```
End_p
```

```
n_1 d = a + b
```

```
C_3 Call p
```

```
R_3
```

```
n_3
```

```
End_p
```

```
Start_{q}
```

```
R_4
```

```
n_4
```

```
a = 1
```

```
n_2
```

```
C_4
```

```
End_{q}
```
Program Representation for Interprocedural Data Flow Analysis: Supergraph
Program Representation for Interprocedural Data Flow Analysis: Supergraph

\[
\begin{align*}
Start_{main} & \quad a + b \\
C_1 & \quad \text{Call p} \\
End_{main} & \quad R_1 \\
\quad & \quad R_2 \quad \text{End}_p \\
Start_p & \quad C_2 \quad \text{Call q} \\
\quad & \quad C_3 \quad \text{Call p} \\
\quad & \quad R_3 \\
n_3 & \quad n_4 \\
\quad & \quad R_4 \\
\quad & \quad \text{End}_q \\
Start_q & \quad a = 1 \\
\quad & \quad n_2 \\
\quad & \quad n_4 \\
\quad & \quad \text{Call p} \\
\quad & \quad C_4
\end{align*}
\]
Program Representation for Interprocedural Data Flow Analysis: Supergraph

Start

\[ \text{main} \]
\[ a + b \]
\[ \text{Call p} \]
\[ \text{C}_1 \]
\[ \text{R}_1 \]
\[ \text{End}_{\text{main}} \]

\[ \text{Start}_{\text{p}} \]
\[ \text{Call p} \]
\[ \text{C}_2 \]
\[ \text{Call q} \]
\[ \text{R}_2 \]
\[ \text{End}_{\text{p}} \]

\[ n_1 \]
\[ d = a + b \]
\[ \text{Call p} \]
\[ \text{C}_3 \]
\[ \text{Call p} \]
\[ \text{R}_3 \]
\[ \text{End}_{\text{p}} \]

\[ a = 1 \]
\[ \text{Call p} \]
\[ \text{C}_4 \]
\[ \text{Call p} \]
\[ \text{R}_4 \]
\[ \text{End}_{\text{q}} \]

Start

\[ q \]
\[ n_2 \]
\[ \text{End}_{\text{q}} \]
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
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.

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.

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.

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

A path which represents legal control flow

- 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 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 \textit{intraprocedurally} valid paths

- Context sensitive analysis:
  Considers \textit{interprocedurally} valid paths

- For \textit{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
Context Sensitivity in Interprocedural Analysis

$$S_s \xrightarrow{x} C_i \xrightarrow{x'} E_s \xleftarrow{x} S_r \xrightarrow{f_r} S_t \xrightarrow{y} C_j \xrightarrow{y'} E_t$$
Context Sensitivity in Interprocedural Analysis

\[ S_s \xrightarrow{c_i} C_i \xrightarrow{x} S_r \xrightarrow{y} C_j \xleftarrow{fr} E_r \xleftarrow{x'} E_s \]

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

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

\[ C_i \xrightarrow{c_i} E_s \]

\[ S_r \xrightarrow{f_r} R_j \xrightarrow{y'} E_t \]

\[ S_t \xrightarrow{y} C_j \]

\[ S_t \xrightarrow{y} C_j \]

\[ E_t \xrightarrow{y'} C_j \]
Context Sensitivity in Presence of Recursion

\[ \text{Diagram with nodes } S_p, S_k, S_r, S_j, S_i, S_q, E_i, E_q, E_p, E_k, E_r, u, v \]
Context Sensitivity in Presence of Recursion

\[
\begin{align*}
u & \rightarrow S_p \\
& \rightarrow S_i \\
& \rightarrow S_q \\
& \rightarrow S_j \\
& \rightarrow S_k \\
& \rightarrow S_r \\
& \rightarrow u
\end{align*}
\]

\[
\begin{align*}
f' & \rightarrow E_i \\
& \rightarrow E_q \\
& \rightarrow E_j \\
& \rightarrow E_p \\
& \rightarrow E_k \\
& \rightarrow E_r \\
& \rightarrow f'
\end{align*}
\]
Context Sensitivity in Presence of Recursion
Context Sensitivity in Presence of Recursion
Context Sensitivity in Presence of Recursion

The diagram illustrates the context sensitivity in the presence of recursion. It shows a cycle involving functions and environments, with arrows indicating the flow of context sensitivity.

- **f**: A function representing a recursive call.
- **g**: Another function possibly representing another recursive call.
- **S_p**, **S_k**, **S_r**, **S_j**, **S_q**, **S_i**: Various states or contexts.
- **E_p**, **E_k**, **E_r**, **E_i**, **E_q**, **E_j**: Environments or contexts.
- **u**, **v**, **f'**, **g'**, **h**: Various inputs or transitions.

The diagram highlights the interplay between different functions and contexts, emphasizing the importance of context sensitivity in analyzing recursive programs.
Context Sensitivity in Presence of Recursion

\[ f \rightarrow S_k \rightarrow S_r \rightarrow S_j \rightarrow S_p \rightarrow f' \rightarrow h \rightarrow g' \rightarrow E_i \rightarrow E_q \rightarrow E_p \rightarrow E_k \rightarrow E_r \rightarrow E_j \]
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

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 $f$. 
Staircase Diagrams of Interprocedurally Valid Paths
Staircase Diagrams of Interprocedurally Valid Paths

\[ \begin{align*}
C_p & \rightarrow C_k \rightarrow C_r \\
C_i & \rightarrow C_q \\
R_i & \rightarrow R_q \\
R_P & \rightarrow R_k \rightarrow R_r \\
v & \rightarrow f' \rightarrow u
\end{align*} \]
Staircase Diagrams of Interprocedurally Valid Paths

Oct 2009 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
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.
Flow Insensitivity in Data Flow Analysis

- Assumption: Statements can be executed in any order.
- Instead of computing point-specific data flow information, summary data flow information is computed. The summary information is required to be a safe approximation of point-specific information for each point.
- \( \text{Kill}_n(x) \) component is ignored.
  If statement \( n \) kills data flow information, there is an alternate path that excludes \( n \).
Flow Insensitivity in Data Flow Analysis

Assuming that $\text{DepGen}_n(x) = \emptyset$, and $\text{Kill}_n(X)$ is ignored for all $n$.

Control flow graph

Flow insensitive analysis
Flow Insensitivity in Data Flow Analysis

Assuming that $\text{DepGen}_n(x) = \emptyset$, and $\text{Kill}_n(X)$ is ignored for all $n$

Control flow graph

Flow insensitive analysis

*Function composition is replaced by function confluence*
Flow Insensitivity in Data Flow Analysis

If $\text{DepGen}_n(x) \neq \emptyset$ for some basic block

$\text{DepGen}_0(x) \neq \emptyset$

$\text{DepGen}_1(x) \neq \emptyset$

$\text{DepGen}_2(x) = \emptyset$

$\text{DepGen}_3(x) = \emptyset$

$\text{DepGen}_4(x) = \emptyset$

$\text{DepGen}_5(x) \neq \emptyset$

Control flow graph

Flow insensitive analysis
Flow Insensitivity in Data Flow Analysis

An alternative model if $\text{DepGen}_n(x) \neq \emptyset$
Flow Insensitivity in Data Flow Analysis

An alternative model if $\text{DepGen}_n(x) \neq \emptyset$

Allows arbitrary compositions of flow functions in any order $\Rightarrow$ Flow insensitivity
Flow Insensitivity in Data Flow Analysis

An alternative model if $\text{DepGen}_n(x) \neq \emptyset$

In practice, dependent constraints are collected in a global repository in one pass and then are solved independently.
Example of Flow Insensitive Analysis

Flow insensitive points-to analysis
⇒ Same points-to information at each program point
Example of Flow Insensitive Analysis

Flow insensitive points-to analysis
⇒ Same points-to information at each program point

Program

1. \( a = &b \)
2. \( c = a \)
3. \( a = &d \)
4. \( a = &e \)
5. \( b = a \)
Example of Flow Insensitive Analysis

Flow insensitive points-to analysis
⇒ Same points-to information at each program point

Program

Constraints

<table>
<thead>
<tr>
<th>Node</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$P_a \supseteq {b}$</td>
</tr>
<tr>
<td>2</td>
<td>$P_c \supseteq P_a$</td>
</tr>
<tr>
<td>3</td>
<td>$P_a \supseteq {d}$</td>
</tr>
<tr>
<td>4</td>
<td>$P_a \supseteq {e}$</td>
</tr>
<tr>
<td>5</td>
<td>$P_b \supseteq P_a$</td>
</tr>
</tbody>
</table>
Example of Flow Insensitive Analysis

Flow insensitive points-to analysis
⇒ Same points-to information at each program point

Program

Constraints

<table>
<thead>
<tr>
<th>Node</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$P_a \supseteq {b}$</td>
</tr>
<tr>
<td>2</td>
<td>$P_c \supseteq P_a$</td>
</tr>
<tr>
<td>3</td>
<td>$P_a \supseteq {d}$</td>
</tr>
<tr>
<td>4</td>
<td>$P_a \supseteq {e}$</td>
</tr>
<tr>
<td>5</td>
<td>$P_b \supseteq P_a$</td>
</tr>
</tbody>
</table>

Points-to Graph

- a
- b
- c
- d
- e

Oct 2009
**Example of Flow Insensitive Analysis**

Flow insensitive points-to analysis
⇒ Same points-to information at each program point

Program:

1. \( a = \&b \)
2. \( c = a \)
3. \( a = \&d \)
4. \( a = \&e \)
5. \( b = a \)

Constraints:

<table>
<thead>
<tr>
<th>Node</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( P_a \supseteq { b } )</td>
</tr>
<tr>
<td>2</td>
<td>( P_c \supseteq P_a )</td>
</tr>
<tr>
<td>3</td>
<td>( P_a \supseteq { d } )</td>
</tr>
<tr>
<td>4</td>
<td>( P_a \supseteq { e } )</td>
</tr>
<tr>
<td>5</td>
<td>( P_b \supseteq P_a )</td>
</tr>
</tbody>
</table>

Points-to Graph:

```
  d
 /\  \\
/   \\
/     \
 a     b
|     |
|     |
 c     e
|     |
|     |
 a     d
|     |
|     |
 b     e
|     |
|     |
 c     f
|     |
|     |
 a     g
|     |
|     |
 b     h
```

Oct 2009
Example of Flow Insensitive Analysis

Flow insensitive points-to analysis
⇒ Same points-to information at each program point

Program

<table>
<thead>
<tr>
<th>Node</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(P_a \supseteq {b})</td>
</tr>
<tr>
<td>2</td>
<td>(P_c \supseteq P_a)</td>
</tr>
<tr>
<td>3</td>
<td>(P_a \supseteq {d})</td>
</tr>
<tr>
<td>4</td>
<td>(P_a \supseteq {e})</td>
</tr>
<tr>
<td>5</td>
<td>(P_b \supseteq P_a)</td>
</tr>
</tbody>
</table>

Points-to Graph

Oct 2009
Example of Flow Insensitive Analysis

Flow insensitive points-to analysis
⇒ Same points-to information at each program point

Program

<table>
<thead>
<tr>
<th>Node</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$P_a \supseteq {b}$</td>
</tr>
<tr>
<td>2</td>
<td>$P_c \supseteq P_a$</td>
</tr>
<tr>
<td>3</td>
<td>$P_a \supseteq {d}$</td>
</tr>
<tr>
<td>4</td>
<td>$P_a \supseteq {e}$</td>
</tr>
<tr>
<td>5</td>
<td>$P_b \supseteq P_a$</td>
</tr>
</tbody>
</table>

Constraints Points-to Graph

- c does not point to any location in block 1
- c does not point b in block 5
- b does not point to itself at any time
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 sensitive

- Flow sensitive
  - Context insensitive
  - Context sensitive

- Context insensitive
  - Flow insensitive
  - Flow sensitive

- Context sensitive
  - Flow insensitive
  - Flow sensitive

Actually, only caller sensitive
Part 4

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
Notation for Summary Flow Function

For simplicity forward flow is assumed.

Procedure $r$

\[
\begin{align*}
&f_4 \\
&f_3 \\
&f_2 \\
&f_1
\end{align*}
\]
Notation for Summary Flow Function

For simplicity forward flow is assumed.

Procedure $r$

$\Phi_r(u_1) \equiv \phi_{id}$
Notation for Summary Flow Function

For simplicity forward flow is assumed.

Procedure $r$

- $\Phi_r(u_1) \equiv \phi_{id}$
- $\Phi_r(u_2) \equiv f_1$
Notation for Summary Flow Function

For simplicity forward flow is assumed.

Procedure $r$

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

\[
\Phi_r(u_2) \equiv f_1
\]

\[
\Phi_r(u_3) \equiv f_1
\]

\[
\Phi_r(u_4) \equiv f_1
\]
Notation for Summary Flow Function

For simplicity forward flow is assumed.

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

\[
\Phi_r(u_2) \equiv f_1
\]

\[
\Phi_r(u_3) \equiv f_1
\]

\[
\Phi_r(u_4) \equiv f_1
\]

\[
\Phi_r(u_5) \equiv f_2 \circ f_1
\]
Notation for Summary Flow Function

For simplicity forward flow is assumed.

\[
\begin{align*}
\Phi_r(u_1) & \equiv \phi_{id} \\
\Phi_r(u_2) & \equiv f_1 \\
\Phi_r(u_3) & \equiv f_1 \\
\Phi_r(u_4) & \equiv f_1 \\
\Phi_r(u_5) & \equiv f_2 \circ f_1 \\
\Phi_r(u_6) & \equiv f_3 \circ f_1
\end{align*}
\]
Notation for Summary Flow Function

For simplicity forward flow is assumed.

Procedure \( r \)

- \( \Phi_r(u_1) \equiv \phi_{id} \)
- \( \Phi_r(u_2) \equiv f_1 \)
- \( \Phi_r(u_3) \equiv f_1 \)
- \( \Phi_r(u_4) \equiv f_1 \)
- \( \Phi_r(u_5) \equiv f_2 \circ f_1 \)
- \( \Phi_r(u_6) \equiv f_3 \circ f_1 \)
- \( \Phi_r(u_7) \equiv f_2 \circ f_1 \cap f_3 \circ f_1 \)
Notation for Summary Flow Function

For simplicity forward flow is assumed.

\[
\begin{align*}
\Phi_r(u_1) & \equiv \phi_{id} \\
\Phi_r(u_2) & \equiv f_1 \\
\Phi_r(u_3) & \equiv f_1 \\
\Phi_r(u_4) & \equiv f_1 \\
\Phi_r(u_5) & \equiv f_2 \circ f_1 \\
\Phi_r(u_6) & \equiv f_3 \circ f_1 \\
\Phi_r(u_7) & \equiv f_2 \circ f_1 \sqcap f_3 \circ f_1 \\
\Phi_r(u_8) & \equiv f_4 \circ (f_2 \circ f_1 \sqcap f_3 \circ f_1)
\end{align*}
\]
Constructing Summary Flow Function

For simplicity forward flow is assumed.

\[
\Phi_r(Entry(n)) = \begin{cases} 
\phi_{id} & \text{if } n \text{ is Start}_r \\
\prod_{p \in \text{pred}(n)} (\Phi_r(Exit(p))) & \text{otherwise}
\end{cases}
\]

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

\[ \text{Start}_r \]

\[ f_1 \]

\[ f_2 \]
Constructing Summary Flow Functions

Iteration #1

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

\( \Phi_r(u_2) = f_1 \)

\( \Phi_r(u_3) = f_1 \)

\( \Phi_r(u_4) = f_2 \circ f_1 \)
Constructing Summary Flow Functions

Iteration #2

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

\[ \Phi_r(u_2) = f_1 \]

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

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

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

\[ \Phi_r(u_2) = f_1 \]

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

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

**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>
</table>
| \( \hat{\top} = \emptyset \) | \( \begin{array}{ccc}
\text{Gen}_n & \text{Kill}_n & \hat{f}_n \\
\emptyset & \emptyset & \hat{\phi}_{id} \\
\emptyset & \{a\} & \hat{\phi}_T \\
\{a\} & \emptyset & \hat{\phi}_\bot \\
\end{array} \) | \( \hat{\phi}_T \) |
| \( \hat{\bot} = \{a\} \) |                             | \( \hat{\phi}_{id} \) |

Oct 2009
Flow functions for two variables

<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>( \top = \emptyset )</td>
<td>( \emptyset ) ( \phi_{\top} ) ( { b } ) ( \emptyset ) ( \phi_{\bot} )</td>
<td>( \phi_{\top} ) ( \phi_{\bot} )</td>
</tr>
<tr>
<td>{ a } { b }</td>
<td>( \emptyset ) ( { a } ) ( \phi_{\bot} ) ( { b } ) ( \emptyset ) ( \phi_{\top} )</td>
<td>( \phi_{\bot} ) ( \phi_{\top} )</td>
</tr>
<tr>
<td>( \bot = { a, b } )</td>
<td>( \emptyset ) ( { b } ) ( \phi_{\top} ) ( { b } ) ( { a, b } ) ( \phi_{\bot} )</td>
<td>( \phi_{\bot} ) ( \phi_{\top} )</td>
</tr>
</tbody>
</table>
Reducing Function Compositions

Assumption: No dependent parts (as in bit vector frameworks).

Killₙ is $\text{ConstKill}ₙ$ and Genₙ is $\text{ConstGen}ₙ$.

\[
f₃(x) = f₂(f₁(x)) = f₂((x - \text{Kill}₁) \cup \text{Gen}₁) = \left(\left((x - \text{Kill}₁) \cup \text{Gen}₁\right) - \text{Kill}₂\right) \cup \text{Gen}₂ = (x - (\text{Kill}₁ \cup \text{Kill}₂)) \cup (\text{Gen}₁ - \text{Kill}₂) \cup \text{Gen}₂
\]

Hence,

\[
\text{Kill}₃ = \text{Kill}₁ \cup \text{Kill}₂
\]
\[
\text{Gen}₃ = (\text{Gen}₁ - \text{Kill}₂) \cup \text{Gen}₂
\]
Reducing Function Confluences

Assumption: No dependent parts (as in bit vector frameworks). Kill$_n$ is ConstKill$_n$ and Gen$_n$ is ConstGen$_n$.

- When $\sqcap$ is $\cup$,

\[ f_3(x) = f_2(x) \cup f_1(x) \]
\[ = ((x - \text{Kill}_2) \cup \text{Gen}_2) \cup ((x - \text{Kill}_1) \cup \text{Gen}_1) \]
\[ = (x - (\text{Kill}_1 \cap \text{Kill}_2)) \cup (\text{Gen}_1 \cup \text{Gen}_2) \]

Hence,

\[ \text{Kill}_3 = \text{Kill}_1 \cap \text{Kill}_2 \]
\[ \text{Gen}_3 = \text{Gen}_1 \cup \text{Gen}_2 \]
Reducing Function Confluences

Assumption: No dependent parts (as in bit vector frameworks). Kill\(_n\) is \(ConstKill\_n\) and Gen\(_n\) is \(ConstGen\_n\).

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

\[
\begin{align*}
  f_3(x) & = f_2(x) \cap f_1(x) \\
  & = ((x - \text{Kill}_2) \cup \text{Gen}_2) \cap ((x - \text{Kill}_1) \cup \text{Gen}_1) \\
  & = (x - (\text{Kill}_1 \cup \text{Kill}_2)) \cup (\text{Gen}_1 \cap \text{Gen}_2)
\end{align*}
\]

Hence

\[
\begin{align*}
  \text{Kill}_3 & = \text{Kill}_1 \cup \text{Kill}_2 \\
  \text{Gen}_3 & = \text{Gen}_1 \cap \text{Gen}_2
\end{align*}
\]
An Example of Interprocedural Liveness Analysis

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

\[
\begin{align*}
& n_1 \\
& \quad \text{Call } p \\
& \quad a = a + 2 \\
& \quad \text{print } c + d \\
\end{align*}
\]

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

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

\[
\begin{align*}
& c_2 \\
& \quad \text{Call } q \\
\end{align*}
\]

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

\[
\begin{align*}
& n_3 \\
& \quad c = a + b \\
& n_2 \\
& \quad \text{Call } q \\
\end{align*}
\]

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

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

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

\[
\begin{align*}
E_q & : \quad a = a \times b \\
\end{align*}
\]
# Summary Flow Functions for Interprocedural Liveness Analysis

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

\[ b = 2 \]
\[ \text{if } (b < d) \]
\[ n_3 \]
\[ c = a + b \]
\[ c_4 \]
\[ \text{Call q} \]
\[ E_p \]
\[ \text{print } c + d \]
\[ S_p \]

Summary Flow Function

<table>
<thead>
<tr>
<th>( \Phi_p(E_p) )</th>
<th>( B\lambda_p \cup {c, d} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Phi_p(n_3) )</td>
<td>( (B\lambda_p - {c}) \cup {a, b, d} )</td>
</tr>
<tr>
<td>( \Phi_p(c_4) )</td>
<td>( (B\lambda_p - {a, b, c}) \cup {d} )</td>
</tr>
<tr>
<td>( \Phi_p(S_p) )</td>
<td>( (B\lambda_p - {b, c}) \cup {a, d} )</td>
</tr>
<tr>
<td>( \Phi_q(E_q) )</td>
<td>( (B\lambda_q - {a}) \cup {a, b} )</td>
</tr>
<tr>
<td>( \Phi_q(c_3) )</td>
<td>( (B\lambda_q - {a, b, c}) \cup {a, d} )</td>
</tr>
<tr>
<td>( \Phi_q(S_q) )</td>
<td>( (B\lambda_q - {a, b, c}) \cup {d} )</td>
</tr>
</tbody>
</table>
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>( \text{In}_{E_m} )</td>
<td>( \Phi_m(E_m) )</td>
<td>( BI_m \cup {a, c} )</td>
</tr>
<tr>
<td>( \text{In}_{c_2} )</td>
<td>( \Phi_m(c_2) )</td>
<td>( (BI_m - {a, b, c}) \cup {d} )</td>
</tr>
<tr>
<td>( \text{In}_{n_2} )</td>
<td>( \Phi_m(n_2) )</td>
<td>( (BI_m - {a, b, c, d}) \cup {a, b} )</td>
</tr>
<tr>
<td>( \text{In}_{n_1} )</td>
<td>( \Phi_m(n_1) )</td>
<td>( (BI_m - {a, b, c, d}) \cup {a, b, c, d} )</td>
</tr>
<tr>
<td>( \text{In}_{c_1} )</td>
<td>( \Phi_m(c_1) )</td>
<td>( (BI_m - {a, b, c, d}) \cup {a, d} )</td>
</tr>
<tr>
<td>( \text{In}_{S_m} )</td>
<td>( \Phi_m(S_m) )</td>
<td>( BI_m - {a, b, c, d} )</td>
</tr>
</tbody>
</table>
## 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>Name</td>
<td>Definition</td>
<td>Value</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td><strong>Procedure $p$, $BL = {a, b, c, d}$</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ln_{E_p}$</td>
<td>$\Phi_p(E_p)$</td>
<td>$BL_p \cup {c, d}$</td>
</tr>
<tr>
<td>$ln_{n3}$</td>
<td>$\Phi_p(n_3)$</td>
<td>$(BL_p - {c}) \cup {a, b, d}$</td>
</tr>
<tr>
<td>$ln_{c4}$</td>
<td>$\Phi_p(c_4)$</td>
<td>$(BL_p - {a, b, c}) \cup {d}$</td>
</tr>
<tr>
<td>$ln_{S_p}$</td>
<td>$\Phi_p(S_p)$</td>
<td>$(BL_p - {b, c}) \cup {a, d}$</td>
</tr>
<tr>
<td><strong>Procedure $q$, $BL = {a, b, c, d}$</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ln_{E_q}$</td>
<td>$\Phi_q(E_q)$</td>
<td>$(BL_q - {a}) \cup {a, b}$</td>
</tr>
<tr>
<td>$ln_{c3}$</td>
<td>$\Phi_q(c_3)$</td>
<td>$(BL_q - {a, b, c}) \cup {a, d}$</td>
</tr>
<tr>
<td>$ln_{S_q}$</td>
<td>$\Phi_q(S_q)$</td>
<td>$(BL_q - {a, b, c}) \cup {d}$</td>
</tr>
</tbody>
</table>
Result of Interprocedural Liveness Analysis

\[
\begin{align*}
S_{\text{main}} & \quad \emptyset \\
& a = 5; b = 3 \\
& c = 7; \text{read } d \\
& \{a, d\} \\
& \text{Call } p \\
& \{a, b, c, d\} \\
n_1 & a = a + 2 \\
& \text{print } c + d \\
& \{a, b\} \\
n_2 & d = a \times b \\
& \{d\} \\
c_2 & \text{Call } q \\
& \{a, c\} \\
E_{\text{main}} & \text{print } a + c
\end{align*}
\]

\[
\begin{align*}
S_p & \quad b = 2 \\
& \text{if } (b < d) \\
& \{a, b, d\} \\
& \{a, d\} \\
n_3 & c = a + b \\
& \{a, b, c, d\} \\
c_4 & \text{Call } q \\
& \{a, b, c, d\} \\
E_p & \text{print } c + d \\
& \{a, b, c, d\} \\
& \{d\}
\end{align*}
\]

\[
\begin{align*}
S_q & \quad a = 1 \\
& \{a, d\} \\
c_3 & \text{Call } p \\
& \{a, b, c, d\} \\
E_q & a = a \times b \\
& \{a, b, c, d\}
\end{align*}
\]
Context Sensitivity of Interprocedural Liveness Analysis

\[ S_{main} \]
\[ \emptyset \]
\[ a = 5; b = 3 \]
\[ c = 7; \text{read } d \]
\[ \{a, d\} \]
\[ \text{Call } p \]
\[ \{a, b, c, d\} \]
\[ n_1 \]
\[ a = a + 2 \]
\[ e = c + d \]
\[ \{a, b, e\} \]
\[ n_2 \]
\[ d = a \times b \]
\[ \{d, e\} \]
\[ c_1 \]
\[ \{a, b, e\} \]
\[ \{a, c, e\} \]
\[ E_{main} \]
\[ \text{print } a + c + e \]

\[ S_p \]
\[ b = 2 \]
\[ \text{if } (b < d) \]
\[ \{a, b, d, e\} \]
\[ T \]
\[ \{a, b, c, d, e\} \]
\[ c_4 \]
\[ \text{Call } q \]
\[ \{a, b, c, d, e\} \]
\[ F \]
\[ \{d, e\} \]
\[ n_3 \]
\[ c = a + b \]
\[ \{a, b, d, e\} \]
\[ E_p \]
\[ \text{print } c + d \]

\[ S_q \]
\[ a = 1 \]
\[ \{a, d, e\} \]
\[ \{a, b, c, d, e\} \]
\[ c_3 \]
\[ \text{Call } p \]
\[ \{a, d, e\} \]
\[ \{d, e\} \]
\[ \{a, b, c, d, e\} \]
\[ E_q \]
\[ a = a \times b \]
Context Sensitivity of Interprocedural Liveness Analysis

\[ S_{main} \]

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

\[ \emptyset \]

\[ \{a, d\} \]

\[ \{a, d\} \]

\[ \{a, b, c, d\} \]

\[ \{a, b, e\} \]

\[ {a, b, c, d} \]

\[ \{a, b, e\} \]

\[ \{d, e\} \]

\[ \{d, e\} \]

\[ c = a + b \]

\[ \{a, b, c, d, e\} \]

\[ S_p \]

\[ b = 2 \]

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

\[ \{a, b, d, e\} \]

\[ T \]

\[ F \{d, e\} \]

\[ n_1 \]

\[ n_2 \]

\[ \text{Call } p \]

\[ \text{Call } q \]

\[ \text{Call } p \]

\[ \text{Call } q \]

\[ \text{Call } q \]

\[ \text{Call } q \]

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

\[ \begin{align*}
    & c = a + b \\
    & a = a + 2 \\
    & e = c + d
\end{align*} \]

\[ {d, e} \]

\[ {a, b, c, d, e} \]

\[ {a, b, c, d, e} \]

\[ {a, d, e} \]

\[ {a, d, e} \]

\[ {a, b, c, d, e} \]

\[ f_p \text{ and } f_q \text{ remain same} \]

\[ e \in \text{In}_{S_p} \text{ but } e \not\in \text{In}_{c_1} \]
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 when $\text{DepGen}_n \neq \emptyset$
  - May work for some instances of some problems but not for all
Limitations of Functional Approach to Interprocedural Data Flow Analysis

- Problems with constructing summary flow functions
  - Reducing expressions defining flow functions may not be possible when $\text{DepGen}_n \neq \emptyset$
  - May work for some instances of some problems but not for all

- Enumeration based 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
Limitations of Functional Approach to Interprocedural Data Flow Analysis

• Problems with constructing summary flow functions
  ▶ Reducing expressions defining flow functions may not be possible when $\text{DepGen}_n \neq \emptyset$
  ▶ May work for some instances of some problems but not for all

• Enumeration based 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
**Functional Approach for Constant Propagation Example**

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

\[
\begin{array}{c}
c_1 \\
Call \ p
\end{array}
\]

\[
\begin{array}{c}
n_1 \\
a = a + 2 \\
print \ c + d
\end{array}
\]

\[
\begin{array}{c}
n_2 \\
d = a \ast b
\end{array}
\]

\[
\begin{array}{c}
c_2 \\
Call \ q
\end{array}
\]

\[
\begin{array}{c}
E_{main} \\
print \ a + c
\end{array}
\]

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

\[
\begin{array}{c}
n_3 \\
c = a + b
\end{array}
\]

\[
\begin{array}{c}
c_4 \\
Call \ q
\end{array}
\]

\[
\begin{array}{c}
E_p \\
print \ c + d
\end{array}
\]

\[ S_q \]
\[
\begin{array}{c}
a = 1
\end{array}
\]

\[
\begin{array}{c}
c_3 \\
Call \ p
\end{array}
\]

\[
\begin{array}{c}
E_q \\
a = a \ast b
\end{array}
\]
## Summary Flow Functions for Interprocedural Constant Propagation

<table>
<thead>
<tr>
<th>Flow Function</th>
<th>Iteration #1</th>
<th>Changes in iteration #2</th>
<th>Changes in iteration #3</th>
<th>Changes in iteration #4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Phi_p(E_p)$</td>
<td>$\langle v_a, 2, v_c, v_d \rangle$</td>
<td>$\langle 2, 2, 3, v_d \rangle$</td>
<td>$\langle \perp, 2, 3, v_d \rangle$</td>
<td>$\langle \perp, 2, \perp, v_d \rangle$</td>
</tr>
<tr>
<td>$\Phi_p(n_3)$</td>
<td>$\langle v_a, 2, v_a + 2, v_d \rangle$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Phi_p(c_4)$</td>
<td>$\langle \hat{\top}, \hat{\top}, \hat{\top}, \hat{\top} \rangle$</td>
<td>$\langle 2, 2, 3, v_d \rangle$</td>
<td>$\langle \perp, 2, 3, v_d \rangle$</td>
<td>$\langle \perp, 2, \perp, v_d \rangle$</td>
</tr>
<tr>
<td>$\Phi_p(S_p)$</td>
<td>$\langle v_a, 2, v_a + 2, v_d \rangle$</td>
<td>$\langle v_a \sqcap 2, 2, (v_a + 2) \sqcap 3, v_d \rangle$</td>
<td>$\langle \perp, 2, \perp, v_d \rangle$</td>
<td></td>
</tr>
<tr>
<td>$\Phi_q(E_q)$</td>
<td>$\langle 1, v_b, v_c, v_d \rangle$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Phi_q(c_3)$</td>
<td>$\langle 1, 2, 3, v_d \rangle$</td>
<td>$\langle \perp, 2, 3, v_d \rangle$</td>
<td>$\langle \perp, 2, \perp, v_d \rangle$</td>
<td></td>
</tr>
</tbody>
</table>
### Interprocedural Constant Propagation Using the Functional Approach

<table>
<thead>
<tr>
<th>Block</th>
<th>$Out_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_m$</td>
<td>$\langle 5, 3, 7, \perp \rangle$</td>
</tr>
<tr>
<td>$c_1$</td>
<td>$\langle \perp, 2, \perp, \perp \rangle$</td>
</tr>
<tr>
<td>$n_1$</td>
<td>$\langle \perp, 2, \perp, \perp \rangle$</td>
</tr>
<tr>
<td>$n_2$</td>
<td>$\langle \perp, 2, \perp, \perp \rangle$</td>
</tr>
<tr>
<td>$c_2$</td>
<td>$\langle \perp, 2, \perp, \perp \rangle$</td>
</tr>
<tr>
<td>$E_m$</td>
<td>$\langle \perp, 2, \perp, \perp \rangle$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block</th>
<th>$Out_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_p$</td>
<td>$\langle \perp, 2, \perp, \perp \rangle$</td>
</tr>
<tr>
<td>$n_3$</td>
<td>$\langle \perp, 2, \perp, \perp \rangle$</td>
</tr>
<tr>
<td>$c_4$</td>
<td>$\langle \perp, 2, \perp, \perp \rangle$</td>
</tr>
<tr>
<td>$E_p$</td>
<td>$\langle \perp, 2, \perp, \perp \rangle$</td>
</tr>
<tr>
<td>$S_q$</td>
<td>$\langle \perp, 2, \perp, \perp \rangle$</td>
</tr>
<tr>
<td>$c_3$</td>
<td>$\langle \perp, 2, \perp, \perp \rangle$</td>
</tr>
<tr>
<td>$E_2$</td>
<td>$\langle \perp, 2, \perp, \perp \rangle$</td>
</tr>
</tbody>
</table>
Constant Propagation Using Functional Approach

**Code**

```plaintext
a = 5; b = 3
c = 7; read d

main
Call p

Call q

if (b < d)

print c + d

d = a * b

print c + d

E_p

S_p

S_q

E_q

c = a + b

a = 1

Call q

Call p

a = a * b
```

Oct 2009

IIT Bombay
Constant Propagation Using Functional Approach

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

\[\text{Call } p\]

\[a = a + 2\]
\[\text{print } c + d\]

\[\text{Call } q\]

\[d = a \times 2\]

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

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

\[c = a + 2\]

\[\text{Call } q\]

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

\[\text{Call } p\]

\[a = a \times 2\]
Tutorial Problem for Functional Interprocedural Analysis

\[ S_{main} \]

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

\[ n_1 \]

\[
\begin{align*}
  a &= a + 2 \\
  \text{print } c + d
\end{align*}
\]

\[ c_1 \]

\[
\text{Call } p
\]

\[ E_{main} \]

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

\[ c_2 \]

\[
\text{Call } q
\]

\[ n_2 \]

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

\[ S_p \]

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

\[ n_3 \]

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

\[ E_p \]

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

\[ S_q \]

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

\[ c_3 \]

\[
\text{Call } p
\]

\[ E_q \]

\[
\begin{align*}
  a &= a \times b
\end{align*}
\]
Tutorial Problem for Functional Interprocedural Analysis

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

\[ \text{main} \]
\[ n_1 \]
\[ a = 7 \]
\[ \text{print} \ 7 + d \]

\[ c_1 \]
\[ \text{Call p} \]

\[ n_2 \]
\[ d = 14 \]

\[ c_2 \]
\[ \text{Call q} \]

\[ E_{\text{main}} \]
\[ \text{print} \ 2 + c \]

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

\[ n_3 \]
\[ c = a + 2 \]

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

\[ \text{Call q} \]
\[ a = 1 \]

\[ \text{Call p} \]
\[ a = 2 \]
Tutorial Problem for Functional Interprocedural Analysis

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

\[ a = 7 \]
\[ \text{print} \ 7 + d \]

\[ d = 14 \]

\[ c = a + 2 \]

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

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

\[ a = 1 \]

\[ a = 2 \]
Tutorial Problem for Functional Interprocedural Analysis

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

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

\[ c_1 \]
\[ \text{Call p} \]

\[ a = 7 \]
\[ \text{print } 7 + d \]

\[ n_1 \]

\[ d = 14 \]

\[ n_2 \]

\[ c_2 \]
\[ \text{Call q} \]

\[ S_p \]

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

\[ n_3 \]
\[ c = a + 2 \]

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

\[ S_q \]
\[ a = 1 \]

\[ c_3 \]
\[ \text{Call p} \]

\[ E_q \]
\[ a = 2 \]

\[ E_{\text{main}} \]
\[ \text{print } 2 + c \]
Tutorial Problem for Functional Interprocedural Analysis

\[ S_{main} \]

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

\[ n_1 \]

\[ \text{Call } p \]

\[ \text{Call } q \]

\[ d = 14 \]

\[ E_{main} \]

\[ \text{print } 2 + c \]

\[ S_p \]

\[ b = 2 \]

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

\[ T \]

\[ F \]

\[ n_3 \]

\[ c = a + 2 \]

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

\[ n_2 \]

\[ a = 7 \quad \text{print } 7 + d \]

\[ E_p \]

\[ S_q \]

\[ a = 1 \]

\[ c_3 \]

\[ \text{Call } p \]

\[ a = 2 \]

\[ E_q \]
Tutorial Problem for Functional Interprocedural Analysis

\[ S_{main} \]
\[ a = 5; \ b = 3 \]
\[ c = 7; \ read \ d \]
\[ n_1 \]
\[ c_1 \]
\[ Call \ p \]
\[ n_2 \]
\[ d = 14 \]
\[ c_2 \]
\[ Call \ q \]
\[ E_{main} \]
\[ print \ 2 + 3? \]

\[ S_p \]
\[ b = 2 \]
\[ if \ (2 < d) \]
\[ n_3 \]
\[ c = a + 2 \]
\[ E_p \]
\[ print \ c + d \]

\[ S_q \]
\[ a = 1 \]
\[ c_3 \]
\[ Call \ p \]
\[ E_q \]
\[ a = 2 \]
Part 5

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 $S_{main}$

A snap-shot of call stack in terms of call sites
Interprocedural Data Flow Analysis Using Call Strings

- Tagged data flow information
  - $\text{IN}_n$ and $\text{OUT}_n$ are sets of the form \( \{ \langle \sigma, x \rangle \mid \sigma \text{ is a call string }, x \in L \} \)
  - The final data flow information is
    \[
    \text{In}_n = \bigcap_{\langle \sigma, x \rangle \in \text{IN}_n} x
    \]
    \[
    \text{Out}_n = \bigcap_{\langle \sigma, x \rangle \in \text{OUT}_n} x
    \]

- Flow functions to manipulate tagged data flow information
  - Intraprocedural edges manipulate data flow value $x$
  - Interprocedural edges manipulate call string $\sigma$
Overall Data Flow Equations

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

\[ \text{OUT}_n = \text{DepGEN}_n \]

Effectively, \( \text{ConstGEN}_n = \text{ConstKILL}_n = \emptyset \) and \( \text{DepKILL}_n(X) = X \).

\[ X \uplus Y = \{ \langle \sigma, x \cap y \rangle \mid \langle \sigma, x \rangle \in X, \langle \sigma, y \rangle \in Y \} \cup \\
\{ \langle \sigma, x \rangle \mid \langle \sigma, x \rangle \in X, \forall z \in L, \langle \sigma, z \rangle \notin Y \} \cup \\
\{ \langle \sigma, y \rangle \mid \langle \sigma, y \rangle \in Y, \forall z \in L, \langle \sigma, z \rangle \notin X \} \]

(We merge underlying data flow values only if the contexts are same.)
Interprocedural Validity and Calling Contexts
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.
“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.
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.
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.
Manipulating Values

- Call edge $C_i \rightarrow S_p$ (i.e. call site $c_i$ calling procedure $p$).
  - Append $c_i$ to every $\sigma$.
  - Propagate the data flow values unchanged.
Manipulating Values

- Call edge $C_i \rightarrow S_p$ (i.e. call site $c_i$ calling procedure $p$).
  - Append $c_i$ to every $\sigma$.
  - Propagate the data flow values unchanged.

- Return edge $E_p \rightarrow R_i$ (i.e. $p$ returning the control to call site $c_i$).
  - If the last call site is $c_i$, remove it and propagate the data flow value unchanged.
  - Block other data flow values.
Manipulating Values

- Call edge $C_i \rightarrow S_p$ (i.e. call site $c_i$ calling procedure $p$).
  - Append $c_i$ to every $\sigma$.
  - Propagate the data flow values unchanged.

- Return edge $E_p \rightarrow R_i$ (i.e. $p$ returning the control to call site $c_i$).
  - If the last call site is $c_i$, remove it and propagate the data flow value unchanged.
  - Block other data flow values.
Manipulating Values

- Call edge $C_i \rightarrow S_p$ (i.e. call site $c_i$ calling procedure $p$).
  - Append $c_i$ to every $\sigma$.
  - Propagate the data flow values unchanged.

- Return edge $E_p \rightarrow R_i$ (i.e. $p$ returning the control to call site $c_i$).
  - If the last call site is $c_i$, remove it and propagate the data flow value unchanged.
  - Block other data flow values.

$$\text{DepGEN}_n(X) = \begin{cases} 
\{ \langle \sigma \cdot c_i, x \rangle | \langle \sigma, x \rangle \in X \} & n \text{ is } C_i \\
\{ \langle \sigma, x \rangle | \langle \sigma \cdot c_i, x \rangle \in X \} & n \text{ is } R_i \\
\{ \langle \sigma, f_n(x) \rangle | \langle \sigma, x \rangle \in X \} & \text{otherwise}
\end{cases}$$
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 $p$

$R_2$

$n_3$
- $t = a \times b$

$E_p$
Available Expressions Analysis Using Call Strings Approach

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

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

\[ R_1 \]
- Is \( a \times b \) available?

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

\[ E_{main} \]

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

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

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

\[ R_2 \]

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

\[ E_p \]

Oct 2009

IIT Bombay
Available Expressions Analysis Using Call Strings Approach

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

```
read a, b

S_{main}

C_1

C_2

R_1

R_2

n_1

n_2

n_3

is a \times b available?

S_p

if a == 0

n_2

a = a - 1

C_2

call p

E_{main}

print a \times b

E_p

t = a \times b

E_p
\n```
Available Expressions Analysis Using Call Strings Approach

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

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

\(R_1\)
- Is \(a \times b\) available?
  - Yes!

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

\(E_{main}\)

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

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

\(R_2\)

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

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

\(E_p\)

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

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 $a == 0$

$n_2$
- $a = a - 1$

$C_2$
- call $p$

$R_2$

$n_3$
- $t = a \times b$

$E_p$

Kill Oct 2009 IIT Bombay
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

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

\[ C_1 \]
- call p

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

\[ n_1 \]

\[ E_{main} \]

\[ S_p \]
- if a == 0

\[ n_2 \]
- a = a - 1

\[ C_2 \]
- call p

\[ R_2 \]

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

\[ E_p \]

\[ \text{Kill} \]
Available Expressions Analysis Using Call Strings Approach

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

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

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

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

\[ S_p \rightarrow \text{if } a == 0 \]
\[ n_2 \rightarrow a = a - 1 \]

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

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

\[ E_p \rightarrow \text{Gen} \]
\[ \text{Kill} \]
Available Expressions Analysis Using Call Strings Approach

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

\[ C_1 \]
call \( p \)

\[ C_2 \]
call \( p \)

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

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

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

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

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

\[ E_p \]

\[ R_1 \]

\[ R_2 \]

Kill

Gen
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 \( a \times b \)

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

\( S_p \)
- 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

$$S_{main} \quad \text{read } a, b \quad t := a \ast b$$

$$C_1 \quad \text{call } p$$

$$R_1$$

$$n_1 \quad \text{print } a \ast b$$

$$E_{main}$$

$$S_p \quad \text{if } a == 0$$

$$n_2 \quad a = a - 1$$

$$C_2 \quad \text{call } p$$

$$R_2$$

$$n_3 \quad t = a \ast b$$

$$E_p$$

⟨λ|1⟩
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

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

\( \langle \lambda|1 \rangle \)

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

\( n_1 \)
- **print** \( a \ast b \)

\( E_{main} \)

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

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

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

\( R_1 \)

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

\( E_p \)
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

\( S_{main} \):
- read \( a, b \)
- \( t := a \ast b \)
- \( S_p \):
  - if \( a == 0 \)
- \( C_1 \):
  - call \( p \)
- \( n_1 \):
  - print \( a \ast b \)
- \( E_{main} \)

\( C_2 \):
- call \( p \)
- \( n_2 \):
  - \( a = a - 1 \)
- \( R_1 \)

\( n_3 \):
- \( t = a \ast b \)
- \( R_2 \)

\( E_p \)
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

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

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

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

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

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

\[ n_2 \]
\[ a = a - 1 \]
\[ \langle c_1 | 0 \rangle \]

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

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

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

Maintain a worklist of nodes to be processed

$S_{main}$
- 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 $p$

$R_2$

$n_3$
- $t = a \times b$

$E_p$

$\langle c_1|1 \rangle$

$\langle c_1|2|0 \rangle$

$\langle c_1|0 \rangle$

$\langle c_1|2|0 \rangle$

$\langle c_1|0 \rangle$

$\langle c_1|1 \rangle$

$\langle \lambda|1 \rangle$

$\langle c_1|1 \rangle$

$\langle c_1|1 \rangle$

$\langle c_1|1 \rangle$

Oct 2009
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 \( p \)

\[ R_1 \]
- \[ n_1 \]
- 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 \( p \)

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

\[ E_p \]

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

\[ \langle c_1 | 0 \rangle \]
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 \( p \)

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

\[ E_p \]

\[ \langle c_1 | 1 \rangle, \langle c_1 c_2 | 0 \rangle, \langle c_1 c_2 c_2 | 0 \rangle, \ldots \]

\[ \langle \lambda | 1 \rangle, \langle c_1 c_1 | 0 \rangle, \langle c_1 c_2 | 0 \rangle, \ldots \]

\[ \langle c_1 c_1 c_1 | 0 \rangle, \langle c_1 c_2 c_2 | 0 \rangle, \ldots \]
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

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

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

\[ R_1 \]

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

\[ E_{main} \]

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

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

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

\[ R_2 \]

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

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

Maintain a worklist of nodes to be processed

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

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

\[ R_1 \]

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

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

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

\[ \langle c_1 | 1 \rangle \]
\[ \langle c_1 c_2 | 0 \rangle \]
\[ \langle c_1 c_2 c_2 | 0 \rangle \]

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

\[ R_2 \]

\[ n_2 \]
\[ a = a - 1 \]
\[ \langle c_1 | 0 \rangle \]
\[ \langle c_1 c_2 | 0 \rangle \]

\[ E_p \]

\[ n_3 \]
\[ t = a \ast b \]
\[ \langle c_1 c_2 | 0 \rangle \]
\[ \langle c_1 c_2 c_2 | 0 \rangle \]

Oct 2009
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

\[ S_{main} \]

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

\[ \langle \lambda | 1 \rangle \]

\[ C_1 \]

\[ \text{call } p \]

\[ R_1 \]

\[ n_1 \]

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

\[ E_{main} \]

\[ \langle c_1 | 1 \rangle \]

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

\[ S_p \]

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

\[ \langle \lambda | 1 \rangle \]

\[ n_2 \]

\[ a = a - 1 \]

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

\[ C_2 \]

\[ \text{call } p \]

\[ R_2 \]

\[ n_3 \]

\[ t = a \times b \]

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

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

\[ \ldots \]

\[ E_p \]

\[ \langle c_1 | 1 \rangle \]

\[ \langle c_1 c_2 | 1 \rangle \]
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

$S_{main}$

read $a, b$
$t := a \times b$

$C_1$

$\langle \lambda | 1 \rangle$

call $p$

$R_1$

$\langle c_1 | 1 \rangle$

$n_1$

print $a \times b$

$E_{main}$

$S_p$

if $a == 0$

$n_2$

$a = a - 1$

$C_2$

call $p$

$R_2$

$\langle c_1 | 0 \rangle, \langle c_1 c_2 | 0 \rangle, \ldots$

$\langle c_1 c_2 | 0 \rangle, \langle c_1 c_2 c_2 | 0 \rangle, \ldots$

$t = a \times b$

$E_p$

$\langle c_1 | 1 \rangle$

$\langle c_1 c_2 | 1 \rangle$
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

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

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

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

\[ E_{main} \]

\[ \langle c_1 | 1 \rangle \]

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

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

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

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

\[ R_2 \quad \text{print } a \ast b \]

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

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

\[ E_p \quad \langle c_1 | 1 \rangle \]

\[ \langle c_1 c_2 | 1 \rangle \]
### Tutorial Problem

Generate a trace of the preceding example in the following format:

<table>
<thead>
<tr>
<th>Step No.</th>
<th>Selected Node</th>
<th>Qualified Data Flow Value</th>
<th>Remaining Work List</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>( \text{IN}_n ) \quad \text{OUT}_n</td>
<td></td>
</tr>
</tbody>
</table>

- Assume that call site \( c_i \) appended to a call string \( \sigma \) only if there are at most 2 occurrences of \( c_i \) in \( \sigma \)
- What about work list organization?
The Need for Multiple Occurrences of a Call Site

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

```c
int a, b, c;
void main()
{
    c = a * b;
p();
}

void p()
{
    if (...)
    {
        p();
    }
    // Is a*b available?
    a = a * b;
}
```
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. \text{Is } a*b \text{ 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. }

Path 1

Path 2
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;
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?](highlight)
10.    a = a*b;
11. }
12. }

\[
\begin{align*}
S_{main} & \quad \downarrow \quad n_1 \quad \downarrow \quad C_1 \quad \downarrow \quad R_1 \quad \downarrow \quad E_{main} \\
& \quad c = a \times b \\
C_1 & \quad \downarrow \quad R_1 \\
& \quad a = a \times b \\
E_{main} & \quad \uparrow \quad \langle c_1, 1 \rangle \\
S_p & \quad \uparrow \quad \langle c_1, 1 \rangle \quad \uparrow \quad \langle c_1 \, c_2 \, c_2, 1 \rangle
\end{align*}
\]
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.     if (...) 
9.         Is a*b available?
10.     
11.     a = a * b;
12. }

- Interprocedurally valid IFP

\[ S_{main} \]
\[ n_1 \quad c = a \times b \]
\[ C_1 \]
\[ R_1 \]
\[ E_{main} \]
\[ S_p \]
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.     }
10.   Is a*b available?
11. }
12. }

- Interprocedurally valid IFP

\[
C_2, S_p, E_p, R_2, n_2, E_p, R_2, n_2
\]
The Need for Multiple Occurrences of a Call Site

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

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

- Interprocedurally valid IFP

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

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

1. int a, b, c;
2. void main()
3. {
   c = a*b;
4.   p();
5. }
6. void p()
7. {
   if (...) 
   { 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
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 upto 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 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)
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
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
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

⇒ Large number of long call strings
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 \( f_{\rho}(Bl) = f_{\rho'}(Bl) \).

  \( \Rightarrow \rho \), the longer path, is redundant for data flow analysis
Classical Call String Length

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
Classical Call String Length

\[ M \]

\[ \langle c_i, \alpha_i, \beta_i \rangle \]

\[ \rho_0 \]

\[ \rho \]
Classical Call String Length

\[
\langle c_j, \alpha_i, \Omega \rangle
\]
Classical Call String Length

- Number of distinct triples \( \langle c_i, \alpha_i, \beta_i \rangle \) is \( M = K \cdot (|L| + 1)^2 \).
Classical Call String Length

- 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.
When $\beta_i$ is not $\Omega$
Classical Call String Length

When $\beta_i$ is not $\Omega$
When $\beta_i$ is not $\Omega$
When $\beta_i$ is $\Omega$
When $\beta_i$ is $\Omega$
When $\beta_i$ is $\Omega$
Tighter Bound for Bit Vector Frameworks

- \( \hat{L} \) is \( \{0, 1\} \), \( L \) is \( \{0, 1\}^m \)
- \( \hat{\otimes} \) is either boolean AND or boolean OR
- \( \hat{\top} \) and \( \hat{\bot} \) are 0 or 1 depending on \( \hat{\otimes} \).
- \( \hat{h} \) 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>
</tbody>
</table>

Oct 2009
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 of 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

Which paths in a supergraph are sufficient to construct maximal call strings?

- All paths from $C_i$ to $R_i$ are abstracted away when a new call node $C_j$ is to be suffixed to a call string
- We should consider maximal interprocedurally valid paths in which there is no path from a return node to a call node
- 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
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 $R_T$

- $P(Entry \rightsquigarrow C_S \rightsquigarrow C_T \rightsquigarrow R_T)$
  - 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
  - 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
  - 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 up to a given length $m$. 

$C_a$

$R_a$
Classical Approximate Approach

- Maintain call string suffixes of up to a given length $m$.

\[ \langle C_1 \cdot C_2 \ldots C_{m-1} \ | \ x \rangle \]

\[
\begin{array}{c}
C_a \\
\downarrow \\
R_a
\end{array}
\]
Classical Approximate Approach

- Maintain call string suffixes of upto a given length $m$.

Call string of length $m - 1$:
\[
\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_{m-1}} \mid x \rangle
\]

\[
C_a
\]

Call string of length $m$:
\[
\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_{m-1}} \cdot C_a \mid x \rangle
\]

\[
R_a
\]
Classical Approximate Approach

- Maintain call string suffixes of up to a given length $m$.

Call string of length $m - 1$:

$$\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_{m-1}} \mid x \rangle$$

Call string of length $m$:

$$\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_{m-1}} \cdot C_a \mid x \rangle$$

$$\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_{m-1}} \cdot C_a \mid y \rangle$$

$Ca$

$Ra$
Classical Approximate Approach

- Maintain call string suffixes of upto a given length $m$.

Call string of length $m - 1$

$\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_{m-1}} \mid x \rangle$

$\sum \langle C_{i_1} \cdot C_{i_2} \ldots C_{i_{m-1}} \cdot C_a \mid x \rangle$

$\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_{m-1}} \cdot C_a \mid y \rangle$

$\sum \langle C_{i_1} \cdot C_{i_2} \ldots C_{i_{m-1}} \mid y \rangle$
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 up to a given length $m$.

\[
\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_m} \mid x \rangle
\]

\[
\downarrow
\]

\[
C_a
\]

\[
\langle C_{i_2} \ldots C_{i_m} \cdot C_a \mid x \rangle
\]

\[
\downarrow
\]

\[
R_a
\]

(First call site $c_{i_1}$ removed from incoming call string and call site $c_a$ attached)
Classical Approximate Approach

- Maintain call string suffixes of up to a given length \( m \).

\[
\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_m} \mid x \rangle \\
C_a \\
\langle C_{i_2} \ldots C_{i_m} \cdot C_a \mid x \rangle \\
Ra
\]

\[
\langle C_{i_2} \ldots C_{i_m} \cdot C_a \mid y \rangle
\]

(First call site \( c_{i_1} \) removed from incoming call string and call site \( c_a \) attached)
Classical Approximate Approach

- Maintain call string suffixes of upto a given length $m$.

Call string of length $m$

\[ \langle C_{i_1} \cdot C_{i_2} \ldots C_{i_m} \mid x \rangle \]

\[ \xrightarrow{C_a} \]

\[ \langle C_{i_2} \ldots C_{i_m} \cdot C_a \mid x \rangle \]

\[ \xrightarrow{R_a} \]

\[ \langle C_{i_1} \cdot C_{i_2} \ldots C_{i_m} \mid y \rangle \]

(First call site $c_{i_1}$ removed from incoming call string and call site $c_a$ attached)
Classical Approximate Approach

- Maintain call string suffixes of up to a given length $m$.

$$\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_m} \mid x_1 \rangle$$

$C_a$

$R_a$
Classical Approximate Approach

- Maintain call string suffixes of up to a given length $m$.

\[ \langle C_{i_1} \cdot C_{i_2} \ldots C_{i_m} \mid x_1 \rangle \quad \langle C_{j_1} \cdot C_{i_2} \ldots C_{i_m} \mid x_2 \rangle \]
Classical Approximate Approach

- Maintain call string suffixes of up to a given length $m$.

\[
\langle C_{i_1} \cdot C_{i_2} \cdots C_{i_m} \mid x_1 \rangle \quad \langle C_{j_1} \cdot C_{i_2} \cdots C_{i_m} \mid x_2 \rangle
\]

\[
\langle C_{i_2} \cdot C_{i_3} \cdots C_{i_m} \cdot C_a \mid x_1 \sqcap x_2 \rangle
\]

\[
R_a
\]
Classical Approximate Approach

- Maintain call string suffixes of up to a given length $m$.

$$\langle C_1 \cdot C_2 \ldots C_m | x_1 \rangle \quad \langle C_{j_1} \cdot C_{i_2} \ldots C_{i_m} | x_2 \rangle$$

$$\langle C_{i_2} \cdot C_{i_3} \ldots C_{i_m} \cdot C_a | x_1 \cap x_2 \rangle$$

$$\langle C_{i_2} \cdot C_{i_3} \ldots C_{i_m} \cdot C_a | y \rangle$$
Classical Approximate Approach

- Maintain call string suffixes of upto a given length $m$.

$$\langle C_{i_1} \cdot C_{i_2} \cdots C_{i_m} | x_1 \rangle \quad \langle C_{j_1} \cdot C_{i_2} \cdots C_{i_m} | x_2 \rangle$$

$$\langle C_{i_2} \cdot C_{i_3} \cdots C_{i_m} \cdot C_a | x_1 \cap x_2 \rangle$$

$$\langle C_{i_2} \cdot C_{i_3} \cdots C_{i_m} \cdot C_a | y \rangle$$

$$\langle C_{i_1} \cdot C_{i_2} \cdots C_{i_m} | y \rangle \quad \langle C_{j_1} \cdot C_{i_2} \cdots C_{i_m} | y \rangle$$
**Classical Approximate Approach**

- Maintain call string suffixes of upto a given length $m$.

\[
\langle C_{i_1} \cdot C_{i_2} \cdots C_{i_m} \mid x_1 \rangle \quad \langle C_{j_1} \cdot C_{i_2} \cdots C_{i_m} \mid x_2 \rangle
\]

- Practical choices of $m$ have been 1 or 2.
Approximate Call Strings in Presence of Recursion

• For simplicity, assume $m = 2$

$$\langle C_b \mid x_1 \rangle$$

[Diagram with nodes $C_a$ and $R_a$]
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

\[
\langle C_b \mid x_1 \rangle
\]

\[
C_a
\]

\[
\langle C_b \cdot C_a \mid x_1 \rangle
\]

\[
R_a
\]
Approximate Call Strings in Presence of Recursion

For simplicity, assume $m = 2$

\[ \langle C_b \mid x_1 \rangle \rightarrow \langle C_b \cdot C_a \mid x_2 \rangle \]

\[ \langle C_b \cdot C_a \mid x_1 \rangle \rightarrow R_a \]
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

\[
\langle C_b \mid x_1 \rangle \quad \langle C_b \cdot C_a \mid x_2 \rangle
\]

\[
\langle C_b \cdot C_a \mid x_1 \rangle, \; \langle C_a \cdot C_a \mid x_2 \rangle
\]

$R_a$
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

\[
\langle C_b \mid x_1 \rangle \quad \langle C_b \cdot C_a \mid x_2 \rangle, \quad \langle C_a \cdot C_a \mid x_3 \rangle
\]

\[
\langle C_b \cdot C_a \mid x_1 \rangle, \quad \langle C_a \cdot C_a \mid x_2 \rangle
\]

\[
R_a
\]
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

$$\langle C_b \mid x_1 \rangle, \langle C_b \cdot C_a \mid x_2 \rangle, \langle C_a \cdot C_a \mid x_3 \rangle$$

$$\langle C_b \cdot C_a \mid x_1 \rangle, \langle C_a \cdot C_a \mid x_2 \sqcap x_3 \rangle$$

$$R_a$$
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

\[
\langle C_b \mid x_1 \rangle, \quad \langle C_b \cdot C_a \mid x_2 \rangle, \quad \langle C_b \cdot C_a \mid x_1 \rangle, \quad \langle C_a \cdot C_a \mid x_4 \rangle
\]

\[
\langle C_b \cdot C_a \mid x_1 \rangle, \quad \langle C_a \cdot C_a \mid x_2 \setminus x_3 \rangle
\]
Approximate Call Strings in Presence of Recursion

• For simplicity, assume $m = 2$

\[
\langle C_b \mid x_1 \rangle, \quad \langle C_b \cdot C_a \mid x_2 \rangle, \quad \langle C_a \cdot C_a \mid x_4 \rangle
\]

\[
\langle C_b \cdot C_a \mid x_1 \rangle, \quad \langle C_a \cdot C_a \mid x_5 \rangle
\]

\[
\text{\small $C_a$}
\]

\[
\text{\small $R_a$}
\]

\[
\text{\small Oct 2009 IIT Bombay}
\]
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 $m = 2$

$$\langle C_b \mid x_1 \rangle, \quad \langle C_b \cdot C_a \mid x_2 \rangle, \quad \langle C_a \cdot C_a \mid x_4 \rangle$$

$$\langle C_a \rangle$$

$$\langle C_b \cdot C_a \mid x_1 \rangle, \quad \langle C_a \cdot C_a \mid x_5 \rangle$$

$$\langle C_b \cdot C_a \mid y_1 \rangle, \quad \langle C_a \cdot C_a \mid y_2 \rangle$$

$$\langle C_a \cdot C_a \mid y_2 \rangle$$

$$\langle C_b \mid y_1 \rangle, \quad \langle C_b \cdot C_a \mid y_2 \rangle, \quad \langle C_a \cdot C_a \mid y_2 \rangle$$
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

$$\langle C_b \mid x_1 \rangle, \langle C_b \cdot C_a \mid x_2 \rangle, \langle C_a \cdot C_a \mid x_4 \rangle$$

$$\langle C_b \cdot C_a \mid x_1 \rangle, \langle C_a \cdot C_a \mid x_5 \rangle$$

$$\langle C_b \cdot C_a \mid y_1 \rangle, \langle C_a \cdot C_a \mid y_2 \rangle$$

$$\langle C_b \mid y_1 \rangle, \langle C_b \cdot C_a \mid y_2 \rangle, \langle C_a \cdot C_a \mid y_2 \rangle$$
Part 6

Modified Call Strings Method
An Overview

- Clearly identifies the exact set of call strings required.
An Overview

- Clearly identifies the exact set of call strings required.
- Value based termination of call string construction. No need to construct call strings up to a fixed length.
An Overview

- Clearly identifies the exact set of call strings required.
- Value based termination of call string construction. No need to construct call strings up to a fixed length.
- Only as many call strings are constructed as are required.
An Overview

- Clearly identifies the exact set of call strings required.
- Value based termination of call string construction. No need to construct call strings up to a fixed length.
- Only as many call strings are constructed as are required.
- Significant reduction in space and time.
An Overview

• Clearly identifies the exact set of call strings required.
• Value based termination of call string construction. No need to construct call strings up to a fixed length.
• Only as many call strings are constructed as are required.
• Significant reduction in space and time.
• Worst case call string length becomes linear in the size of the lattice instead of the original quadratic.
An Overview

• Clearly identifies the exact set of call strings required.
• Value based termination of call string construction. No need to construct call strings up to a fixed length.
• Only as many call strings are constructed as are required.
• Significant reduction in space and time.
• Worst case call string length becomes linear in the size of the lattice instead of the original quadratic.

All this is achieved by a simple change without compromising on the precision, simplicity, and generality of the classical method.
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 Modified Algorithm

- Use exactly the same method with this small change:
The Modified Algorithm

- Use exactly the same method with this small change:
  - discard redundant call strings at the start of every procedure, and
The Modified Algorithm

• Use exactly the same method with this small change:
  ▶ discard redundant call strings at the start of every procedure, and
  ▶ simulate regeneration of call strings at the end of every procedure.
The Modified Algorithm

- Use exactly the same method with this small change:
  - discard redundant call strings at the start of every procedure, and
  - simulate regeneration of call strings at the end of every procedure.

- Intuition:
The Modified Algorithm

- Use exactly the same method with this small change:
  - discard redundant call strings at the start of every procedure, and
  - simulate regeneration of call strings at the end of every procedure.
- Intuition:
  - If $\sigma_1$ and $\sigma_2$ have equal values at $S_p$, 

The Modified Algorithm

- Use exactly the same method with this small change:
  - discard redundant call strings at the start of every procedure, and
  - simulate regeneration of call strings at the end of every procedure.

- Intuition:
  - 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 Modified Algorithm

- Use exactly the same method with this small change:
  - discard redundant call strings at the start of every procedure, and
  - simulate regeneration of call strings at the end of every procedure.

- Intuition:
  - 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 $E_p$. 
The Modified Algorithm

• Use exactly the same method with this small change:
  ▶ discard redundant call strings at the start of every procedure, and
  ▶ simulate regeneration of call strings at the end of every procedure.

• Intuition:
  ▶ 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 $E_p$.

• Can equivalence classes change?
The Modified Algorithm

- Use exactly the same method with this small change:
  - discard redundant call strings at the start of every procedure, and
  - simulate regeneration of call strings at the end of every procedure.

- Intuition:
  - 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 $E_p$.

- Can equivalence classes change?
  - During the analysis, equivalence classes may change in the sense that some call strings may move out of one class and may belong to some other class.
The Modified Algorithm

- Use exactly the same method with this small change:
  - discard redundant call strings at the start of every procedure, and
  - simulate regeneration of call strings at the end of every procedure.

- Intuition:
  - 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 $E_p$.

- Can equivalence classes change?
  - During the analysis, equivalence classes may change in the sense that some call strings may move out of one class and may belong to some other class.
  - However, the invariant that the equivalence classes are same at $S_p$ and $E_p$ still holds.
Representation and Regeneration of Call Strings

• Let \( \text{shortest}(\sigma, u) \) denote the shortest call string which has the same value as \( \sigma \) at \( u \).

\[
\text{represent}(\langle \sigma, x \rangle, S_p) = \langle \text{shortest}(\sigma, S_p), x \rangle
\]

\[
\text{regenerate}(\langle \sigma, y \rangle, E_p) = \{ \langle \sigma', y \rangle \mid \sigma \text{ and } \sigma' \text{ have the same value at } S_p \}\]

• Correctness requirement: Whenever representation is performed at \( S_p \), \( E_p \) must be added to the work list.

• Efficiency consideration: Desirable order of processing of nodes
  Intraprocedural nodes → call nodes → return nodes
Safety and Precision of Representation and Regeneration

\[
\langle \sigma \cdot \sigma_c^\omega \mid x_\omega \rangle \quad \langle \sigma \cdot \sigma_{c+1}^\omega \mid x_\omega \rangle
\]

\[S_p\]
Safety and Precision of Representation and Regeneration

\[ \langle \sigma \cdot \sigma_c^\omega | x_\omega \rangle \quad \langle \sigma \cdot \sigma_c^{\omega+1} | x_\omega \rangle \]

\[ \langle \sigma \cdot \sigma_c^\omega \cdot c_i | x_\omega \rangle \quad \langle \sigma \cdot \sigma_c^{\omega+1} \cdot c_i | x_\omega \rangle \]
Safety and Precision of Representation and Regeneration

\[ \langle \sigma \cdot \sigma_c^\omega \mid x_\omega \rangle \quad \langle \sigma \cdot \sigma_c^{\omega+1} \mid x_\omega \rangle \]

\[ \langle \sigma \cdot \sigma_c^\omega \cdot c_i \mid x_\omega \rangle \quad \langle \sigma \cdot \sigma_c^{\omega+1} \cdot c_i \mid x_\omega \rangle \]

\[ \langle \sigma \cdot \sigma_c^\omega \cdot c_i \mid Z_m \rangle \quad \langle \sigma \cdot \sigma_c^{\omega+1} \cdot c_i \mid Z_m \rangle \]
Safety and Precision of Representation and Regeneration

\[
\begin{aligned}
\langle \sigma \cdot \sigma_{\omega} | x_\omega \rangle & \quad \langle \sigma \cdot \sigma_{\omega+1} | x_\omega \rangle \\
\langle \sigma \cdot \sigma_{\omega} \cdot c_i | x_\omega \rangle & \quad \langle \sigma \cdot \sigma_{\omega+1} \cdot c_i | x_\omega \rangle \\
\langle \sigma \cdot \sigma_{\omega} \cdot c_i | Z_m \rangle & \quad \langle \sigma \cdot \sigma_{\omega+1} \cdot c_i | Z_m \rangle \\
\langle \sigma \cdot \sigma_{\omega} | Z_m \rangle & \quad \langle \sigma \cdot \sigma_{\omega+1} | Z_m \rangle \\
S_p & \\
E_p
\end{aligned}
\]
Safety and Precision of Representation and Regeneration

\[ \langle \sigma \cdot \sigma^w_c \mid \chi_\omega \rangle \quad \langle \sigma \cdot \sigma^{w+1}_c \mid \chi_\omega \rangle \]

\[ \langle \sigma \cdot \sigma^w_c \cdot c_i \mid \chi_\omega \rangle \quad \langle \sigma \cdot \sigma^{w+1}_c \cdot c_i \mid \chi_\omega \rangle \]

\[ \langle \sigma \cdot \sigma^w_c \cdot c_i \mid Z_m \rangle \quad \langle \sigma \cdot \sigma^{w+1}_c \cdot c_i \mid Z_m \rangle \]

\[ \langle \sigma \cdot \sigma^w_c \mid Z_m \rangle \quad \langle \sigma \cdot \sigma^{w+1}_c \mid Z_m \rangle \]
Safety and Precision of Representation and Regeneration

\[
\langle \sigma \cdot \sigma_c^\omega | x_\omega \rangle \rightarrow \langle \sigma \cdot \sigma_c^\omega+1 | x_\omega \rangle
\]

\[
\langle \sigma \cdot \sigma_c^\omega \cdot c_i | x_\omega \rangle \rightarrow \langle \sigma \cdot \sigma_c^\omega+1 \cdot c_i | x_\omega \rangle
\]

\[
\langle \sigma \cdot \sigma_c^\omega \cdot c_i | z_m \rangle \rightarrow \langle \sigma \cdot \sigma_c^\omega+1 \cdot c_i | z_m \rangle
\]

Represent

\[
\langle \sigma \cdot \sigma_c^\omega | z_m \rangle \rightarrow \langle \sigma \cdot \sigma_c^\omega+1 | z_m \rangle
\]
Safety and Precision of Representation and Regeneration

\[
\langle \sigma \cdot \sigma^c_\omega \mid x_\omega \rangle \quad \langle \sigma \cdot \sigma^{c+1}_\omega \mid x_\omega \rangle
\]

Represent

\[
\langle \sigma \cdot \sigma^c_\omega \cdot c_i \mid x_\omega \rangle \quad \langle \sigma \cdot \sigma^{c+1}_\omega \cdot c_i \mid x_\omega \rangle
\]

\[
\langle \sigma \cdot \sigma^c_\omega \cdot c_i \mid z_m \rangle \quad \langle \sigma \cdot \sigma^{c+1}_\omega \cdot c_i \mid z_m \rangle
\]

Regenerate

\[
\langle \sigma \cdot \sigma^c_\omega \mid z_m \rangle \quad \langle \sigma \cdot \sigma^{c+1}_\omega \mid z_m \rangle
\]
Safety and Precision of Representation and Regeneration

\[ \langle \sigma \cdot \sigma_c^\omega \mid x_\omega \rangle \quad \langle \sigma \cdot \sigma_c^{\omega+1} \mid x_\omega \rangle \]

Represent

\[ \langle \sigma \cdot \sigma_c^\omega \cdot c_i \mid x_\omega \rangle \quad \langle \sigma \cdot \sigma_c^{\omega+1} \cdot c_i \mid x_\omega \rangle \]

\[ \langle \sigma \cdot \sigma_c^\omega \cdot c_i \mid Z_m \rangle \quad \langle \sigma \cdot \sigma_c^{\omega+1} \cdot c_i \mid Z_m \rangle \]

\[ \langle \sigma \cdot \sigma_c^\omega \mid Z_m \rangle \quad \langle \sigma \cdot \sigma_c^{\omega+1} \mid Z_m \rangle \]

Regenerate
Safety and Precision of Representation and Regeneration

\[ \langle \sigma \cdot \sigma_c^\omega \mid x_\omega \rangle \quad \langle \sigma \cdot \sigma_c^{\omega+1} \mid x_\omega \rangle \]

[Diagram with nodes and arrows indicating representation and regeneration processes]
Safety and Precision of Representation and Regeneration

\[ \langle \sigma \cdot \sigma^\omega_c | x_\omega \rangle \]

\[ \langle \sigma \cdot \sigma^{\omega+1}_c | x_\omega \rangle \]

Represent

\[ \langle \sigma \cdot \sigma^\omega_c \cdot c_i | x_\omega \rangle \]

\[ \langle \sigma \cdot \sigma^\omega_c \cdot c_i | z_m \cap g(z_m) \rangle \]

\[ \langle \sigma \cdot \sigma^\omega_c | Z_m \rangle \]

\[ \langle \sigma \cdot \sigma^{\omega+1}_c | Z_m \rangle \]

Regenerate
Safety and Precision of Representation and Regeneration

\[
\langle \sigma \cdot \sigma^\omega_c \mid x_\omega \rangle \quad \langle \sigma \cdot \sigma^{\omega+1}_c \mid x_\omega \rangle
\]

\[
S_p
\]

Represent

\[
\langle \sigma \cdot \sigma^\omega_c \cdot c_i \mid x_\omega \rangle
\]

\[
\langle \sigma \cdot \sigma^\omega_c \cdot c_i \mid z_m \cap g(z_m) \rangle
\]

\[
E_p
\]

Regenerate

\[
z_{m-1} = z_m \cap g(z_m)
\]
Safety and Precision of Representation and Regeneration

\[ \langle \sigma \cdot \sigma^\omega_c \ | \ x_\omega \rangle \]

Represent

\[ \langle \sigma \cdot \sigma^\omega_c \cdot c_i \ | \ x_\omega \rangle \]

\[ \langle \sigma \cdot \sigma^\omega_c \cdot c_i \ | \ z_m \ \ominus \ g(z_m) \rangle \]

\[ z_{m-1} = z_m \ \ominus \ g(z_m) \]

Regenerate

\[ \langle \sigma \cdot \sigma^\omega_c \ | \ Z_m \rangle \]

\[ \langle \sigma \cdot \sigma^\omega_c+1 \ | \ Z_m \rangle \]
Safety and Precision of Representation and Regeneration

\[
\langle \sigma \cdot \sigma^\omega_c \mid x_\omega \rangle \quad \langle \sigma \cdot \sigma^\omega_{c+1} \mid x_\omega \rangle
\]

\[
\langle \sigma \cdot \sigma^\omega_c \cdot c_i \mid x_\omega \rangle
\]

\[
\langle \sigma \cdot \sigma^\omega_c \cdot c_i \mid z_m \square g(z_m) \rangle
\]

\[
z_{m-1} = z_m \square g(z_m)
\]

\[
\langle \sigma \cdot \sigma^\omega_c, Z_{m-1} \rangle \quad \langle \sigma \cdot \sigma^\omega_{c+1}, Z_{m-1} \rangle
\]

Represent

Regenerate
Safety and Precision of Representation and Regeneration

\[ \langle \sigma \cdot \sigma^\omega \cdot c \cdot c_i \mid x_\omega \rangle \]

\[ \langle \sigma \cdot \sigma^\omega \cdot c \mid x_\omega \rangle \]

\[ \langle \sigma \cdot \sigma^\omega \cdot c_i \mid Z_{m-2} \rangle \]

\[ z_{m-2} = z_m \sqcap g(z_{m-1}) \]

\[ \langle \sigma \cdot \sigma^\omega \mid Z_{m-2} \rangle \]

\[ \langle \sigma \cdot \sigma^\omega +1 \mid Z_{m-2} \rangle \]

\[ S_p \]

Represent

\[ E_p \]

Regenerate

Oct 2009
Safety and Precision of Representation and Regeneration

\[ \langle \sigma \cdot \sigma^\omega_c | x_\omega \rangle \quad \langle \sigma \cdot \sigma^\omega_{c+1} | x_\omega \rangle \]

\[ \langle \sigma \cdot \sigma^\omega_c \cdot c_i | x_\omega \rangle \]

\[ \langle \sigma \cdot \sigma^\omega_c \cdot c_i | Z_{m-i} \rangle \]

\[ Z_{m-i} = Z_m \cap g(Z_{m-(i+1)}) \]

These values are identical to the values computed by the full call strings method

\[ \langle \sigma \cdot \sigma^\omega_c | Z_{m-i} \rangle \quad \langle \sigma \cdot \sigma^\omega_{c+1} | Z_{m-i} \rangle \]

Represent

Regenerate

Oct 2009
Safety and Precision of Representation and Regeneration

\[
\langle \sigma \cdot \sigma^\omega_c \mid x_\omega \rangle \quad \langle \sigma \cdot \sigma^\omega_{c+1} \mid x_\omega \rangle
\]

\[S_p\]

Represent

\[
\langle \sigma \cdot \sigma^\omega_c \cdot c_i \mid x_\omega \rangle
\]

\[
\langle \sigma \cdot \sigma^\omega_c \mid Z_m-i \rangle
\]

\[E_p\]

Regenerate

\[
\text{Stop regeneration after the values converge}
\]

\[
Z_{m-i} = Z_m \cap g(Z_{m-(i+1)})
\]

Other values are computed with smaller call strings similar to the full call strings method

Oct 2009
Equivalence of The Two Methods

- For non-recursive programs, equivalence is obvious
- For recursive program, we prove equivalence using staircase diagrams
Call Strings for Recursive Contexts

Let

- \( \sigma_c \equiv c_j c_r c_k c_p c_i c_q \)
- \( \sigma_r \equiv r_q r_i r_p r_k r_r r_j \)

Assume that we allow up to \( m \) occurrences of \( \sigma_c \)
Computing Data Flow Values along Recursive Paths

\[ x_1 = f(x_0) \]
Computing Data Flow Values along Recursive Paths

\[ x_2 = f^2(x_0) \]
Computing Data Flow Values along Recursive Paths

\[ x_i = f^i(x_0) \]
Computing Data Flow Values along Recursive Paths

\[ x_i = f^i(x_0) \]
Computing Data Flow Values along Recursive Paths

\[ x_i = f^i(x_0) \]
Computing Data Flow Values along Recursive Paths

\[ x_i = f^i(x_0) \]
Computing Data Flow Values along Recursive Paths

\[ x_i = f^i(x_0) \]
Computing Data Flow Values along Recursive Paths

\[ x_i = f^i(x_0) \]

\[ z_m = h(x_m) \]
Computing Data Flow Values along Recursive Paths

\[
x_i = f^i(x_0)
\]

\[
z_{m-1} = h(x_{m-1}) \cap g(z_m)
\]
Computing Data Flow Values along Recursive Paths

\[ x_i = f^i(x_0) \quad \text{and} \quad z_{m-2} = h(x_{m-2}) \sqcap g(z_{m-1}) \]
Computing Data Flow Values along Recursive Paths

\[ x_i = f^i(x_0) \]

\[ z_{m-j} = h(x_{m-j}) \cap g(z_{m-j+1}) \]
Computing Data Flow Values along Recursive Paths

\[ x_i = f^i(x_0) \]

\[ z_{m-j} = h(x_{m-j}) \cap g(z_{m-j+1}) \]
Computing Data Flow Values along Recursive Paths

\[ x_i = f^i(x_0) \]

\[ z_{m-j} = h(x_{m-j}) \cap g(z_{m-j+1}) \]
Computing Data Flow Values along Recursive Paths

\[ x_i = f^i(x_0) \]

\[ z_{m-j} = h(x_{m-j}) \cap g(z_{m-j+1}) \]
Computing Data Flow Values along Recursive Paths

\[ x_i = f^i(x_0) \]

\[ z_{m-j} = h(x_{m-j}) \cap g(z_{m-j+1}) \]
Fixed Bound Closure Bound of Flow Function

- $n > 0$ 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

\[ x_1 = f(x_0) \]
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

$x_2 = f^2(x_0)$
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

$x_\omega = f^\omega(x_0)$
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

\[
x_i = \begin{cases} 
f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases}
\]
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

$$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases}$$
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

$x_i = \begin{cases} 
  f^i(x_0) & i < \omega \\
  f^\omega(x_0) & \text{otherwise}
\end{cases}$
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

$x_i = \begin{cases} 
  f^i(x_0) & i < \omega \\
  f^\omega(x_0) & \text{otherwise}
\end{cases}$
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

$$x_i = \begin{cases} 
  f^i(x_0) & i < \omega \\
  f^\omega(x_0) & \text{otherwise}
\end{cases}$$

$$z_m = h(x_\omega)$$
FP closure bound of $f$

$x_i = \begin{cases} 
  f^i(x_0) & i < \omega \\
  f^\omega(x_0) & \text{otherwise}
\end{cases}$

$z_{m-1} = h(x_\omega) \sqcap g(z_m)$
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

FP closure bound of $g$

$x_i = \begin{cases} 
    f^i(x_0) & i < \omega \\
    f^\omega(x_0) & \text{otherwise}
\end{cases}$

$z_{m-\eta} = h(x_\omega) \sqcap g(z_{m-\eta+1})$
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

FP closure bound of $g$

\[
\begin{align*}
  x_i &= \begin{cases} 
    f^i(x_0) & i < \omega \\
    f^\omega(x_0) & \text{otherwise} 
  \end{cases} \\
  z_{m-j} &= \begin{cases} 
    h(x_\omega) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\
    h(x_\omega) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\
    h(x_j) \cap g(z_{m-j+1}) & \text{otherwise} 
  \end{cases}
\end{align*}
\]
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

FP closure bound of $g$

$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f(\omega)(x_0) & \text{otherwise} \end{cases}$

$z_{m-j} = \begin{cases} h(x_\omega) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\ h(x_\omega) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\ h(x_j) \cap g(z_{m-j+1}) & \text{otherwise} \end{cases}$
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

FP closure bound of $g$

$x_i = \begin{cases} 
  f^i(x_0) & i < \omega \\
  f^\omega(x_0) & \text{otherwise}
\end{cases}$

$z_{m-j} = \begin{cases} 
  h(x_\omega) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\
  h(x_\omega) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\
  h(x_j) \cap g(z_{m-j+1}) & \text{otherwise}
\end{cases}$
Computation of Data Flow Values along Recursive Paths

\[ x_i = \begin{cases} 
  f^i(x_0) & i < \omega \\
  f^\omega(x_0) & \text{otherwise}
\end{cases} \]

\[ z_{m-j} = \begin{cases} 
  h(x_{\omega}) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\
  h(x_{\omega}) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\
  h(x_j) \cap g(z_{m-j+1}) & \text{otherwise}
\end{cases} \]
Computation of Data Flow Values along Recursive Paths

FP closure bound of \( f \)

FP closure bound of \( g \)

\[ x_i = \begin{cases} 
  f^i(x_0) & \text{if } i < \omega \\
  f^\omega(x_0) & \text{otherwise}
\end{cases} \]

\[ z_{m-j} = \begin{cases} 
  h(x_\omega) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\
  h(x_\omega) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\
  h(x_j) \cap g(z_{m-j+1}) & \text{otherwise}
\end{cases} \]
The Moral of the Story

- In the cyclic call sequence, computation begins from the first call string and influences successive call strings.
The Moral of the Story

• In the cyclic call sequence, computation begins from the **first** call string and influences successive call strings.

• In the cyclic return sequence, computation begins from the **last** call string and influences the preceding call strings.
Bounding the Call String Length Using Data Flow Values

FP closure bound of $f$

FP closure bound of $g$

$$x_i = \begin{cases} 
  f^i(x_0) & i < \omega \\
  f^\omega(x_0) & \text{otherwise}
\end{cases}$$

$$z_{m-j} = \begin{cases} 
  h(x_\omega) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\
  h(x_\omega) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\
  h(x_j) \cap g(z_{m-j+1}) & \text{otherwise}
\end{cases}$$
Bounding the Call String Length Using Data Flow Values

**Theorem:** Data flow values $z_{m-i}$, $0 \leq i \leq \omega$ (computed along $\sigma_r$) follow a strictly descending chain.

$$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases}$$

$$z_{m-j} = \begin{cases} h(x_\omega) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\ h(x_\omega) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\ h(x_j) \cap g(z_{m-j+1}) & \text{otherwise} \end{cases}$$
Bounding the Call String Length Using Data Flow Values

Theorem: Data flow values \( z_{m-i}, 0 \leq i \leq \omega \) (computed along \( \sigma_r \)) follow a strictly descending chain.

Proof Obligation: \( z_{m-(i+1)} \sqsubseteq z_{m-i} \) \( 0 \leq i \leq \omega \)

\[
x_i = \begin{cases} 
  f^i(x_0) & i < \omega \\
  f^\omega(x_0) & \text{otherwise}
\end{cases}
\]

\[
z_{m-j} = \begin{cases} 
  h(x_\omega) \sqcap g(z_{m-j+1}) & 0 \leq j \leq \eta \\
  h(x_\omega) \sqcap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\
  h(x_j) \sqcap g(z_{m-j+1}) & \text{otherwise}
\end{cases}
\]
Bounding the Call String Length Using Data Flow Values

**Theorem:** Data flow values $z_{m-i}$, $0 \leq i \leq \omega$ (computed along $\sigma_r$) follow a strictly descending chain.

**Proof Obligation:** $z_{m-(i+1)} \sqsubseteq z_{m-i}$

**Basis:** $z_{m-1} = h(x_m) \cap g(z_m)$

$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases}$

$z_{m-j} = \begin{cases} h(x_\omega) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\ h(x_\omega) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\ h(x_j) \cap g(z_{m-j+1}) & \text{otherwise} \end{cases}$
Theorem: Data flow values $z_{m-i}$, $0 \leq i \leq \omega$ (computed along $\sigma_r$) follow a strictly descending chain.

Proof Obligation: $z_{m-(i+1)} \sqsubseteq z_{m-i}$  

Basis: $z_{m-1} = h(x_m) \sqcap g(z_m)$

$$z_{m-j} = \begin{cases} h(x_\omega) \sqcap g(z_{m-j+1}) & 0 \leq j \leq \eta \\ h(x_\omega) \sqcap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\ h(x_j) \sqcap g(z_{m-j+1}) & \text{otherwise} \end{cases}$$
**Theorem:** Data flow values $z_{m-i}, 0 \leq i \leq \omega$ (computed along $\sigma_r$) follow a strictly descending chain.

**Proof Obligation:**

\[
\begin{align*}
z_{m-(i+1)} &\sqsubseteq z_{m-i} & \quad 0 \leq i \leq \omega \\
\text{Basis:} & \quad z_{m-1} = h(x_m) \cap g(z_m) \\
& = z_m \cap g(z_m) \\
& \sqsubseteq z_m
\end{align*}
\]

$x_i = \begin{cases} 
  f^i(x_0) & i < \omega \\
  f^\omega(x_0) & \text{otherwise}
\end{cases}$

$z_{m-j} = \begin{cases} 
  h(x_\omega) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\
  h(x_\omega) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\
  h(x_j) \cap g(z_{m-j+1}) & \text{otherwise}
\end{cases}$
Bounding the Call String Length Using Data Flow Values

**Theorem:** Data flow values \( z_{m-i}, 0 \leq i \leq \omega \) (computed along \( \sigma_r \)) follow a strictly descending chain.

**Proof Obligation:**

\[
\begin{align*}
\text{Basis:} & \quad z_{m-1} = h(x_m) \sqcap g(z_m) = z_m \sqcap g(z_m) \\
\quad \subseteq & \quad z_m \\
\text{Inductive step:} & \quad z_{m-k} \subseteq z_{m-(k-1)} \quad \text{(hypothesis)}
\end{align*}
\]

\[
x_i = \begin{cases} 
  f^i(x_0) & i < \omega \\
  f^\omega(x_0) & \text{otherwise}
\end{cases}
\]

\[
z_{m-j} = \begin{cases} 
  h(x_\omega) \sqcap g(z_{m-j+1}) & 0 \leq j \leq \eta \\
  h(x_\omega) \sqcap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\
  h(x_j) \sqcap g(z_{m-j+1}) & \text{otherwise}
\end{cases}
\]
Bounding the Call String Length Using Data Flow Values

Theorem: Data flow values $z_{m-i}$, $0 \leq i \leq \omega$ (computed along $\sigma_r$) follow a strictly descending chain.

Proof Obligation: $z_{m-(i+1)} \sqsubseteq z_{m-i}$, $0 \leq i \leq \omega$

Basis: $z_{m-1} = h(x_m) \sqcap g(z_m) = z_m \sqcap g(z_m) \sqsubseteq z_m$

Inductive step: $z_{m-k} \sqsubseteq z_{m-(k-1)}$ (hypothesis)

$\Rightarrow g(z_{m-k}) \sqsubseteq g(z_{m-(k-1)})$ (monotonicity)

$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^{\omega}(x_0) & \text{otherwise} \end{cases}$

$z_{m-j} = \begin{cases} h(x_\omega) \sqcap g(z_{m-j+1}) & 0 \leq j \leq \eta \\ h(x_\omega) \sqcap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\ h(x_j) \sqcap g(z_{m-j+1}) & \text{otherwise} \end{cases}$
Bounding the Call String Length Using Data Flow Values

**Theorem:** Data flow values $z_{m-i}$, $0 \leq i \leq \omega$ (computed along $\sigma_r$) follow a strictly descending chain.

**Proof Obligation:** $z_{m-(i+1)} \sqsubseteq z_{m-i}$, $0 \leq i \leq \omega$

**Basis:**

$$z_{m-1} = h(x_m) \sqcap g(z_m) = z_m \sqcap g(z_m)$$

**Inductive step:**

$$z_{m-k} \sqsubseteq z_{m-(k-1)}$$ (hypothesis)

$$g(z_{m-k}) \sqsubseteq g(z_{m-(k-1)})$$ (monotonicity)

$$z_{m-k} = z_m \sqcap g(z_{m-(k-1)})$$

$$z_{m-(k+1)} = z_m \sqcap g(z_{m-k})$$

$X_i = \begin{cases} f^i(x_0) & i < \omega \\ f^{\omega}(x_0) & \text{otherwise} \end{cases}$

$Z_{m-j} = \begin{cases} h(x_\omega) \sqcap g(z_{m-j+1}) & 0 \leq j \leq \eta \\ h(x_\omega) \sqcap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\ h(x_j) \sqcap g(z_{m-j+1}) & \text{otherwise} \end{cases}$
Bounding the Call String Length Using Data Flow Values

**Theorem:** Data flow values $z_{m-i}$, $0 \leq i \leq \omega$ (computed along $\sigma_r$) follow a strictly descending chain.

**Proof Obligation:**

**Basis:**

$$z_{m-1} = h(x_m) \cap g(z_m) = z_m \cap g(z_m) \subseteq z_m$$

**Inductive step:**

$$z_{m-k} \subseteq z_{m-(k-1)} \quad \text{(hypothesis)}$$

$$\Rightarrow g(z_{m-k}) \subseteq g(z_{m-(k-1)}) \quad \text{(monotonicity)}$$

$$z_{m-k} = z_m \cap g(z_{m-(k-1)});$$

$$z_{m-(k+1)} = z_m \cap g(z_{m-k})$$

$$\Rightarrow z_{m-(k+1)} \subseteq z_{m-k}$$

**Illustration:**

- FP closure bound of $f$
- FP closure bound of $g$

$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases}$

$z_{m-j} = \begin{cases} h(x_\omega) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\ h(x_\omega) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\ h(x_j) \cap g(z_{m-j+1}) & \text{otherwise} \end{cases}$
Bounding the Call String Length Using Data Flow Values

Theorem: Data flow values $z_{m-i}$, $0 \leq i \leq \omega$ (computed along $\sigma_r$) follow a strictly descending chain.

Conclusion: It is possible to compute these values iteratively by overwriting earlier values. There is no need of constructing call string beyond $\omega + 1$ occurrences of $\sigma$.

$$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases}$$

$$z_{m-j} = \begin{cases} h(x_\omega) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\ h(x_\omega) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\ h(x_j) \cap g(z_{m-j+1}) & \text{otherwise} \end{cases}$$
Bounding the Call String Length Using Data Flow Values

FP closure bound of $f$

$$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases}$$

$$z_{m-j} = \begin{cases} h(x_\omega) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\ h(x_\omega) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\ h(x_j) \cap g(z_{m-j+1}) & \text{otherwise} \end{cases}$$
Bounding the Call String Length Using Data Flow Values

FP closure bound of $f$

$x_i = \begin{cases} 
  f^i(x_0) & i < \omega \\
  f^\omega(x_0) & \text{otherwise}
\end{cases}$

$z_{m-j} = \begin{cases} 
  h(x_\omega) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\
  h(x_\omega) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\
  h(x_j) \cap g(z_{m-j+1}) & \text{otherwise}
\end{cases}$
Bounding the Call String Length Using Data Flow Values

FP closure bound of \( f \)

\[
x_i = \begin{cases} 
  f^i(x_0) & i < \omega \\ 
  f^{\omega}(x_0) & \text{otherwise} 
\end{cases}
\]

\[
z_{m-j} = \begin{cases} 
  h(x_\omega) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\ 
  h(x_\omega) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\ 
  h(x_j) \cap g(z_{m-j+1}) & \text{otherwise} 
\end{cases}
\]
Bounding the Call String Length Using Data Flow Values

FP closure bound of $f$

$x_i = \begin{cases} 
    f^i(x_0) & i < \omega \\
    f^\omega(x_0) & \text{otherwise}
\end{cases}$

$z_{m-j} = \begin{cases} 
    h(x_\omega) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\
    h(x_\omega) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\
    h(x_j) \cap g(z_{m-j+1}) & \text{otherwise}
\end{cases}$
Bounding the Call String Length Using Data Flow Values

FP closure bound of \( f \)

\[
\begin{align*}
\omega + 1 & \quad \omega \\
\sigma_c & \quad x_\omega \\
\sigma_c & \quad f \\
x_0 & \quad h \\
\sigma_c & \quad f \\
x_\omega & \quad h \\
\sigma_r & \quad z_{m-2} \\
\sigma_r & \quad z_{m-2} \\
\sigma_r & \quad \omega \\
\end{align*}
\]

\[
x_i = \begin{cases} 
  f^i(x_0) & i < \omega \\
  f^\omega(x_0) & \text{otherwise}
\end{cases}
\]

\[
z_{m-j} = \begin{cases} 
  h(x_\omega) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\
  h(x_\omega) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\
  h(x_j) \cap g(z_{m-j+1}) & \text{otherwise}
\end{cases}
\]
Bounding the Call String Length Using Data Flow Values

\[ x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases} \]

\[ z_{m-j} = \begin{cases} h(x_\omega) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\ h(x_\omega) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\ h(x_j) \cap g(z_{m-j+1}) & \text{otherwise} \end{cases} \]
Bounding the Call String Length Using Data Flow Values

FP closure bound of $f$

$$\sigma_c \xrightarrow{\omega} x_\omega \xrightarrow{f} x_\omega \xrightarrow{\sigma_c} x_0$$

$$z_{m-j} = \begin{cases} h(x_\omega) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\ h(x_\omega) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\ h(x_j) \cap g(z_{m-j+1}) & \text{otherwise} \end{cases}$$

$$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases}$$
Bounding the Call String Length Using Data Flow Values

FP closure bound of $f$

$x_i = \begin{cases} 
  f^i(x_0) & i < \omega \\
  f^\omega(x_0) & \text{otherwise}
\end{cases}$

$z_{m-j} = \begin{cases} 
  h(x_\omega) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\
  h(x_\omega) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\
  h(x_j) \cap g(z_{m-j+1}) & \text{otherwise}
\end{cases}$
Bounding the Call String Length Using Data Flow Values

FP closure bound of $f$

$$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases} \quad \quad \quad \quad \quad \quad z_{m-j} = \begin{cases} h(x_\omega) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\ h(x_\omega) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\ h(x_j) \cap g(z_{m-j+1}) & \text{otherwise} \end{cases}$$
Worst Case Length Bound

- Consider a call string $\sigma = \ldots (C_i)^1 \ldots (C_i)^2 \ldots (C_i)^3 \ldots (C_i)^j \ldots$
  
  Let $j \geq |L| + 1$

  Let $C_i$ call procedure $p$
Consider a call string $\sigma = \ldots (C_i)^1 \ldots (C_i)^2 \ldots (C_i)^3 \ldots (C_i)^j \ldots$
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 $\sigma = \ldots (C_i)^1 \ldots (C_i)^2 \ldots (C_i)^3 \ldots (C_i)^j \ldots$
  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 $\sigma = \ldots (C_i)^1 \ldots (C_i)^2 \ldots (C_i)^3 \ldots (C_i)^j \ldots$
  - 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 $S_p$.
  - The longer prefix will get represented by the shorter prefix
Worst Case Length Bound

- Consider a call string \( \sigma = \ldots (C_i)^1 \ldots (C_i)^2 \ldots (C_i)^3 \ldots (C_i)^j \ldots \)
  
  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 \( S_p \).
  
  ▶ The longer prefix will get represented by the shorter prefix
  
  ▶ Since one more \( C_i \) is may be suffixed to discover fixed point, \( j \leq |L| + 1 \)
Worst Case Length Bound

- Consider a call string $\sigma = \ldots (C_i)^1 \ldots (C_i)^2 \ldots (C_i)^3 \ldots (C_i)^j \ldots$
  
  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 $S_p$.
  
  - The longer prefix will get represented by the shorter prefix
  - Since one more $C_i$ is may be suffixed to discover fixed point, $j \leq |L| + 1$

- Worst case length in the proposed variant $= K \times (|L| + 1)$
Worst Case Length Bound

- Consider a call string $\sigma = \ldots (C_i)^1 \ldots (C_i)^2 \ldots (C_i)^3 \ldots (C_i)^j \ldots$
  - 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 $S_p$.
  - The longer prefix will get represented by the shorter prefix
  - Since one more $C_i$ is may be suffixed to discover fixed point, $j \leq |L| + 1$

- Worst case length in the proposed variant $= K \times (|L| + 1)$
- Original required length $= K \times (|L| + 1)^2$
Approximate Version

- For framework with infinite lattices, a fixed point for cyclic call sequence may not exist.
Approximate Version

- For framework with infinite lattices, a fixed point for cyclic call sequence may not exist.
- Use a demand driven approach:
  - After a dynamically definable limit (say a number $j$),
  - Start merging the values and associate them with the last call string
Approximate Version

• For framework with infinite lattices, a fixed point for cyclic call sequence may not exist.

• Use a demand driven approach:
  ▶ After a dynamically definable limit (say a number \( j \)),
  ▶ Start merging the values and associate them with the last call string
  ▶ Let

\[
\sigma_j = \ldots (C_i)^1 \ldots (C_i)^2 \ldots (C_i)^3 \ldots (C_i)^j \ldots \\
\sigma_{j+1} = \ldots (C_i)^1 \ldots (C_i)^2 \ldots (C_i)^3 \ldots (C_i)^j \ldots (C_i)^{j+1} \ldots
\]
Approximate Version

- For framework with infinite lattices, a fixed point for cyclic call sequence may not exist.

- Use a demand driven approach:
  - After a dynamically definable limit (say a number $j$),
  - Start merging the values and associate them with the last call string
  - Let
    \[
    \sigma_j = \ldots (C_i)^1 \ldots (C_i)^2 \ldots (C_i)^3 \ldots (C_i)^j \ldots \\
    \sigma_{j+1} = \ldots (C_i)^1 \ldots (C_i)^2 \ldots (C_i)^3 \ldots (C_i)^j \ldots (C_i)^{j+1} \ldots 
    \]
  - Represent $\langle \sigma_j \mid x_j \rangle$ and $\langle \sigma_{j+1} \mid x_{j+1} \rangle$
    by $\langle \sigma^j \mid x_j \sqcap x_{j+1} \rangle$
Approximate Version

- For framework with infinite lattices, a fixed point for cyclic call sequence may not exist.

- Use a demand driven approach:
  - After a dynamically definable limit (say a number \( j \)),
  - Start merging the values and associate them with the last call string
  - Let
    \[
    \sigma_j = \ldots (C_i)^1 \ldots (C_i)^2 \ldots (C_i)^3 \ldots (C_i)^j \ldots \\
    \sigma_{j+1} = \ldots (C_i)^1 \ldots (C_i)^2 \ldots (C_i)^3 \ldots (C_i)^j \ldots (C_i)^{j+1} \ldots
    \]
  - Represent \( \langle \sigma_j \mid x_j \rangle \) and \( \langle \sigma_{j+1} \mid x_{j+1} \rangle \) by \( \langle \sigma^j \mid x_j \sqcap x_{j+1} \rangle \)

- Context sensitive for a depth \( j \) of recursion. Context insensitive beyond that.
Approximate Version

- For framework with infinite lattices, a fixed point for cyclic call sequence may not exist.

- Use a demand driven approach:
  - After a dynamically definable limit (say a number \( j \)),
  - Start merging the values and associate them with the last call string
  - Let
    \[
    \sigma_j = \ldots (C_i)^1 \ldots (C_i)^2 \ldots (C_i)^3 \ldots (C_i)^j \ldots \\
    \sigma_{j+1} = \ldots (C_i)^1 \ldots (C_i)^2 \ldots (C_i)^3 \ldots (C_i)^j \ldots (C_i)^{j+1} \ldots
    \]
  - Represent \( \langle \sigma_j \mid x_j \rangle \) and \( \langle \sigma_{j+1} \mid x_{j+1} \rangle \)
    by \( \langle \sigma^j \mid x_j \sqcap x_{j+1} \rangle \)

- Context sensitive for a depth \( j \) of recursion. Context insensitive beyond that.

- Assumption: Height of the lattice is finite.
# 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 100000+</td>
<td>8</td>
</tr>
<tr>
<td>bit_gray</td>
<td>53</td>
<td>5</td>
<td>11</td>
<td>7 100000+</td>
<td>17</td>
</tr>
<tr>
<td>analyzer</td>
<td>288</td>
<td>14</td>
<td>20</td>
<td>2 21</td>
<td>21</td>
</tr>
<tr>
<td>distraay</td>
<td>331</td>
<td>9</td>
<td>21</td>
<td>6 96</td>
<td>22</td>
</tr>
<tr>
<td>mason</td>
<td>350</td>
<td>9</td>
<td>13</td>
<td>8 100000+</td>
<td>14</td>
</tr>
<tr>
<td>fourinarow</td>
<td>676</td>
<td>17</td>
<td>45</td>
<td>5 510</td>
<td>46</td>
</tr>
<tr>
<td>sim</td>
<td>1146</td>
<td>13</td>
<td>45</td>
<td>8 100000+</td>
<td>211</td>
</tr>
<tr>
<td>181_mcf</td>
<td>1299</td>
<td>17</td>
<td>24</td>
<td>6 32789</td>
<td>41</td>
</tr>
<tr>
<td>256_bzip2</td>
<td>3320</td>
<td>63</td>
<td>198</td>
<td>7 492</td>
<td>406</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.
Tutorial Problem

Perform may points-to analysis using modified call strings method. Make conservative assumptions about must points-to information.

```c
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
  \[ 2 \times (|L| + 1)^2 = 2^{19} + 2^{10} + 1 = 5, 25, 313 \]
- All call strings up to this length must be constructed by the Sharir-Pnueli method!
Tutorial Problem

Perform may points-to analysis using modified call strings method. Make conservative assumptions about must points-to information.

```c
main()
{  x = &y;
    z = &x;
    y = &z;
    p(); /* C1 */
}

p()
{  if (...)
    {  p(); /* C2 */
        x = *x;
    }
}
```

- Modified call strings method requires only three call strings: $\lambda$, $c_1$, and $c_1 c_2$