Interprocedural Data Flow Analysis

Uday P. Khedker

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

May 2011
Part 1

About These Slides
Copyright

These slides constitute the lecture notes for

- MACS L111 Advanced Data Flow Analysis course at Cambridge University, and
- CS 618 Program Analysis course at IIT Bombay.

They have been made available under GNU FDL v1.2 or later (purely for academic or research use) as teaching material accompanying the book:


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

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>Inherited by procedure ( r ) from call site ( c_i ) in procedure ( s )</td>
</tr>
<tr>
<td><strong>y</strong></td>
</tr>
<tr>
<td>Inherited by procedure ( r ) from call site ( c_j ) in procedure ( t )</td>
</tr>
<tr>
<td><strong>x'</strong></td>
</tr>
<tr>
<td>Synthesized by procedure ( r ) in ( s ) at call site procedure ( c_i )</td>
</tr>
<tr>
<td><strong>y'</strong></td>
</tr>
<tr>
<td>Synthesized by procedure ( r ) in ( t ) at call site procedure ( c_j )</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

Start_{main}  \rightarrow  a + b  \rightarrow  Call \ p  \rightarrow  End_{p}  \rightarrow  End_{main}

Start_{p}  \rightarrow  Call \ q  \rightarrow  n_{1}  \rightarrow  d = a + b  \rightarrow  Call \ p  \rightarrow  n_{3}  \rightarrow  n_{4}  \rightarrow  End_{q}

Start_{q}  \rightarrow  a = 1  \rightarrow  n_{2}
Program Representation for Interprocedural Data Flow Analysis: Call Multi-Graph

Supergraphs of procedures

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

Supergraphs of procedures

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

Supergraphs of procedures

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

Supergraphs of procedures

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

Supergraphs of procedures

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

\[ \text{Start}_{\text{main}} \]
\[ a + b \]
\[ \text{End}_{\text{main}} \]

\[ \text{Start}_p \]
\[ \text{Call } p \]
\[ \text{End}_p \]

\[ \text{Start}_p \]
\[ n_1 \]
\[ d = a + b \]
\[ n_3 \]
\[ \text{Call } p \]
\[ \text{End}_q \]

\[ \text{Start}_q \]
\[ n_2 \]
\[ a = 1 \]
\[ n_4 \]
Program Representation for Interprocedural Data Flow Analysis: Supergraph

Start_{main}:
\[ a + b \]
\[ C_1 \text{ Call p} \]
\[ R_1 \]
\[ \text{End}_{main} \]

Start_p:
\[ C_2 \text{ Call q} \]
\[ R_2 \]
\[ \text{End}_p \]

Start_q:
\[ n_1 \text{ d} = a + b \]
\[ a = 1 \]
\[ n_2 \]
\[ C_3 \text{ Call p} \]
\[ R_3 \]
\[ n_3 \]
\[ n_4 \]
\[ C_4 \]
\[ \text{End}_q \]

May 2011
Uday Khedker
Program Representation for Interprocedural Data Flow Analysis: Supergraph

```
Start\_main
\[ a + b \]
\[ C_1 \] Call p
\[ R_1 \]
\[ End\_main \]

Start\_p
\[ C_2 \] Call q
\[ R_2 \]
\[ End\_p \]

n_1 \[ d = a + b \]
\[ C_3 \] Call p
\[ R_3 \]
\[ n_3 \]
\[ End\_q \]

Start\_q
\[ n_1 \]
\[ C_4 \] Call p
\[ n_2 \]
\[ n_3 \]
\[ n_4 \]
```

May 2011

Uday Khedker
Program Representation for Interprocedural Data Flow Analysis: Supergraph

```
Start
main

a + b
C1
Call p

R1
End

End
main

Start
p

Start
q

n1
d = a + b
C2
Call q

R2

n3
d + 1
C3
Call p

R3

n4
d = c
C4
Call p

R4

a = 1
n2

End
q

End
p
```

May 2011  Uday Khedker
Program Representation for Interprocedural Data Flow Analysis: Supergraph
Program Representation for Interprocedural Data Flow Analysis: Supergraph
Validity of Interprocedural Control Flow Paths

Interprocedurally valid control flow path
Validity of Interprocedural Control Flow Paths

Interprocedurally valid control flow path
Validity of Interprocedural Control Flow Paths

Interprocedurally valid control flow path
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
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.
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.

A path which represents legal control flow

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

- Data flow analysis uses static representation of programs to compute summary information along paths.
- **Ensuring Safety.** All valid paths must be covered.
- **Ensuring Precision.** Only valid paths should be covered.
- **Ensuring Efficiency.** Only relevant valid paths should be covered.

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

A path which represents legal control flow.

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

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

- Flow sensitive analysis:
  Considers *intraprocedurally* valid paths

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

- **Flow sensitive analysis:**
  Considers *intraprocedurally* valid paths

- **Context sensitive analysis:**
  Considers *interprocedurally* valid paths

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

- Flow sensitive analysis: Considers *intraprocedurally* valid paths
- Context sensitive analysis: Considers *interprocedurally* valid paths
- For **maximum statically attainable precision**, analysis must be both flow and context sensitive.

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

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

\[ S_s \xrightarrow{x} C_i \xrightarrow{x'} E_s \]
\[ S_r \xrightarrow{y} C_j \xrightarrow{y'} E_t \]

\[ f_r \]

May 2011
Context Sensitivity in Interprocedural Analysis

May 2011

Uday Khedker
Context Sensitivity in Interprocedural Analysis

\[ S_s \rightarrow C_i \rightarrow R_i \rightarrow E_s \]

\[ S_r \rightarrow f_r \rightarrow E_r \rightarrow R_j \rightarrow E_t \]

\[ S_t \rightarrow C_j \rightarrow R_j \rightarrow E_t \]

May 2011
Context Sensitivity in Interprocedural Analysis

May 2011

Uday Khedker
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!"
“You can descend only as much as you have ascended!”

Every descending step must match a corresponding ascending step.
Context Sensitivity in Presence of Recursion

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

May 2011 Uday Khedker
Context Sensitivity in Presence of Recursion
Context Sensitivity in Presence of Recursion
Context Sensitivity in Presence of Recursion

\[
\begin{align*}
&f & f' & g' & g \\
&S_p & S_i & S_q & S_r \\
&E_i & E_q & E_j & E_k & E_r
\end{align*}
\]
Context Sensitivity in Presence of Recursion

Diagram:

- \( u \) to \( S_p \) to \( S_i \) to \( S_q \) to \( S_r \) to \( f \) to \( S_j \) to \( S_k \) to \( u \)
- \( f' \) to \( S_p \) to \( S_i \) to \( S_q \) to \( h \) to \( g \) to \( E_q \)
- \( g' \) to \( E_p \) to \( E_i \) to \( E_q \)
- \( v \) to \( E_p \) to \( E_i \) to \( E_k \) to \( E_r \) to \( v \)
Context Sensitivity in Presence of Recursion

May 2011 Uday Khedker
Context Sensitivity in Presence of Recursion
Context Sensitivity in Presence of Recursion
Context Sensitivity in Presence of Recursion

\[
\begin{align*}
S_k & \rightarrow S_r \\
S_p & \rightarrow S_i \\
S_j & \rightarrow S_q \\
E_i & \rightarrow E_q \\
E_p & \rightarrow E_j \\
E_k & \rightarrow E_r
\end{align*}
\]
Context Sensitivity in Presence of Recursion

- $u \rightarrow f \rightarrow h \rightarrow g \rightarrow v \rightarrow g'$
- $v \rightarrow g \rightarrow h \rightarrow f \rightarrow f'$
- $f' \rightarrow f' \rightarrow f' \rightarrow ...$

$S_p \rightarrow S_i \rightarrow S_q \rightarrow S_r \rightarrow S_k$
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

\[ u \rightarrow C_p \rightarrow C_i \rightarrow C_q \rightarrow C_j \rightarrow C_r \rightarrow C_k \rightarrow v \]

\[ R_p \rightarrow R_k \rightarrow R_r \rightarrow R_q \rightarrow R_i \rightarrow R_j \]
Staircase Diagrams of Interprocedurally Valid Paths
Staircase Diagrams of Interprocedurally Valid Paths

\[ u \xrightarrow{f'} C_p \xrightarrow{C_i} C_q \xrightarrow{h} C_i \xrightarrow{C_q} C_p \]

\[ R_p \xrightarrow{v} R_k \xrightarrow{R_r} R_j \]
Staircase Diagrams of Interprocedurally Valid Paths
Staircase Diagrams of Interprocedurally Valid Paths
Staircase Diagrams of Interprocedurally Valid Paths
Staircase Diagrams of Interprocedurally Valid Paths
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 $\text{DepGen}_n(x) \neq \emptyset$

```
0 f_0
1 f_1
2 f_2
3 f_3
i f_i
m f_m
```

```
0 f_0 1 f_1 2 f_2 3 f_3 \ldots i f_i \ldots m f_m
```

`Start` — `End`
Flow Insensitivity in Data Flow Analysis

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

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

<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

<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
- d
- e

Constraints Points-to Graph

May 2011 Uday Khedker
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>$P_a \supset {b}$</td>
</tr>
<tr>
<td>2</td>
<td>$P_c \supset P_a$</td>
</tr>
<tr>
<td>3</td>
<td>$P_a \supset {d}$</td>
</tr>
<tr>
<td>4</td>
<td>$P_a \supset {e}$</td>
</tr>
<tr>
<td>5</td>
<td>$P_b \supset P_a$</td>
</tr>
</tbody>
</table>

Points-to Graph

May 2011

Uday Khedker
Example of Flow Insensitive Analysis

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

Program

Constraints

Points-to Graph

<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

Points-to Graph

- c does not point to any location in block 1
- a 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
  - Intraprocedural
- Flow sensitive
  - Intraprocedural
- Context insensitive
  - Flow insensitive
  - Flow sensitive
  - Context 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.
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.

\[
\begin{align*}
\Phi_r(u_1) &\equiv \phi_{id} \\
\Phi_r(u_2) &\equiv f_1
\end{align*}
\]
Notation for Summary Flow Function

For simplicity forward flow is assumed.

Procedure $r$

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

For simplicity forward flow is assumed.

Procedure \( r \)

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

For simplicity forward flow is assumed.

Procedure $r$

$\Phi_r(u_1) \equiv \phi_{id}$

$\Phi_r(u_2) \equiv f_1$

$\Phi_r(u_3) \equiv f_1$

$\Phi_r(u_4) \equiv f_1$

$\Phi_r(u_5) \equiv f_2 \circ f_1$

$\Phi_r(u_6) \equiv f_3 \circ f_1$
Notation for Summary Flow Function

For simplicity forward flow is assumed.

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

For simplicity forward flow is assumed.

Procedure $r$

$\Phi_r(u_1) \equiv \phi_{id}$

$\Phi_r(u_2) \equiv f_1$

$\Phi_r(u_3) \equiv f_1$

$\Phi_r(u_4) \equiv f_1$

$\Phi_r(u_5) \equiv f_2 \circ f_1$

$\Phi_r(u_6) \equiv f_3 \circ f_1$

$\Phi_r(u_7) \equiv f_2 \circ f_1 \sqcap f_3 \circ f_1$

$\Phi_r(u_8) \equiv f_4 \circ (f_2 \circ f_1 \sqcap f_3 \circ f_1)$
Reducing Flow Compositions and Meets

\[ f_2 \circ f_1 = f_3 \iff \forall x \in L,\ f_2(f_1(x)) = f_3(x) \]

\[ f_2 \sqcap f_1 = f_3 \iff \forall x \in L,\ f_2(x) \sqcap f_1(x) = f_3(x) \]
Reducing Function Compositions

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

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

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 \(\text{ConstKill}_n\) and Gen\(_n\) is \(\text{ConstGen}_n\).

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

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

Hence,

\[
\begin{align*}
\text{Kill}_3 &= \text{Kill}_1 \cap \text{Kill}_2 \\
\text{Gen}_3 &= \text{Gen}_1 \cup \text{Gen}_2
\end{align*}
\]
Reducing Function Confluences

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

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

\[
\begin{align*}
    f_3(x) &= f_2(x) \cap f_1(x) \\
    &= ((x - Kill_2) \cup Gen_2) \cap ((x - Kill_1) \cup Gen_1) \\
    &= (x - (Kill_1 \cup Kill_2)) \cup (Gen_1 \cap Gen_2)
\end{align*}
\]

Hence

\[
\begin{align*}
    Kill_3 &= Kill_1 \cup Kill_2 \\
    Gen_3 &= Gen_1 \cap Gen_2
\end{align*}
\]
Constructing Summary Flow Function

For simplicity forward flow is assumed.

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

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

Start_{r}

\[ r \]

\[ f_1 \]

\[ f_2 \]
Constructing Summary Flow Functions

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

Iteration #2

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

$\Phi_r(u_2) = f_1$

$\Phi_r(u_3) = f_1 \sqcap f_2 \circ f_1$

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

Iteration #3

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

\[ \Phi_r(u_2) = f_1 \]

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

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

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

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

<table>
<thead>
<tr>
<th>Lattice of data flow values</th>
<th>All possible flow functions</th>
<th>Lattice of flow functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{\top} = \emptyset$</td>
<td>$\hat{\top}$</td>
<td>$\hat{\phi}_T$</td>
</tr>
<tr>
<td>$\downarrow$</td>
<td>$\hat{\phi}_{id}$</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td>$\hat{\bot} = {a}$</td>
<td>$\hat{\phi}_{id}$</td>
<td>$\downarrow$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gen$_n$</th>
<th>Kill$_n$</th>
<th>$\hat{f}_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\hat{\phi}_{id}$</td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>${a}$</td>
<td>$\hat{\phi}_T$</td>
</tr>
<tr>
<td>${a}$</td>
<td>$\emptyset$</td>
<td>$\hat{\phi}_{\bot}$</td>
</tr>
</tbody>
</table>
## Lattice of Flow Functions for Live Variables Analysis

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>$\emptyset$</td>
<td>$\phi_{II}$</td>
</tr>
<tr>
<td>${a}$</td>
<td>${a}$</td>
<td>$\phi_{TT}$</td>
</tr>
<tr>
<td>$\perp = {a, b}$</td>
<td>${b}$</td>
<td>$\phi_{IT}$</td>
</tr>
<tr>
<td>$\top$</td>
<td>$\emptyset$</td>
<td>$\phi_{II}$</td>
</tr>
<tr>
<td>$\bottom$</td>
<td>${a, b}$</td>
<td>$\phi_{TT}$</td>
</tr>
<tr>
<td>$\top$</td>
<td>${a}$</td>
<td>$\phi_{IT}$</td>
</tr>
<tr>
<td>$\top$</td>
<td>${a}$</td>
<td>$\phi_{IT}$</td>
</tr>
<tr>
<td>$\top$</td>
<td>${a}$</td>
<td>$\phi_{IT}$</td>
</tr>
<tr>
<td>$\top$</td>
<td>${a}$</td>
<td>$\phi_{IT}$</td>
</tr>
<tr>
<td>$\top$</td>
<td>${a, b}$</td>
<td>$\phi_{TT}$</td>
</tr>
</tbody>
</table>

May 2011 Uday Khedker
Lattice of Flow Functions for Live Variables Analysis

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>{ a }</td>
<td>{ a, b }</td>
</tr>
<tr>
<td>{ a, b }</td>
<td>{ a }</td>
<td>{ a }</td>
</tr>
<tr>
<td>{ a, b }</td>
<td>{ b }</td>
<td>{ b }</td>
</tr>
<tr>
<td>{ a, b }</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>

Essentially, a product lattice of the two component lattices
An Example of Interprocedural Liveness Analysis

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

\[ \text{Call } p \]

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

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

\[ d = a \times b \]

\[ \text{Call } q \]

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

\[ \text{Call } p \]

\[ a = 1 \]

\[ \text{Call } q \]

\[ \text{Call } p \]

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

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

**Summary Flow Function**

<table>
<thead>
<tr>
<th>Summary Flow Function</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Phi_p(E_p)$</td>
<td>$BL_p \cup {c, d}$</td>
</tr>
<tr>
<td>$\Phi_p(n_3)$</td>
<td>$(BL_p - {c}) \cup {a, b, d}$</td>
</tr>
<tr>
<td>$\Phi_p(c_4)$</td>
<td>$(BL_p - {a, b, c}) \cup {d}$</td>
</tr>
<tr>
<td>$\Phi_p(S_p)$</td>
<td>$(BL_p - {b, c}) \cup {a, d}$</td>
</tr>
<tr>
<td>$\Phi_q(E_q)$</td>
<td>$(BL_q - {a}) \cup {a, b}$</td>
</tr>
<tr>
<td>$\Phi_q(c_3)$</td>
<td>$(BL_q - {a, b, c}) \cup {a, d}$</td>
</tr>
<tr>
<td>$\Phi_q(S_q)$</td>
<td>$(BL_q - {a, b, c}) \cup {d}$</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>Name</td>
<td>Definition</td>
<td></td>
</tr>
<tr>
<td>( E_m )</td>
<td>( \Phi_m(E_m) )</td>
<td>( BI_m \cup {a, c} )</td>
</tr>
<tr>
<td>( c_2 )</td>
<td>( \Phi_m(c_2) )</td>
<td>( (BI_m - {a, b, c}) \cup {d} )</td>
</tr>
<tr>
<td>( n_2 )</td>
<td>( \Phi_m(n_2) )</td>
<td>( (BI_m - {a, b, c, d}) \cup {a, b} )</td>
</tr>
<tr>
<td>( 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>( c_1 )</td>
<td>( \Phi_m(c_1) )</td>
<td>( (BI_m - {a, b, c, d}) \cup {a, d} )</td>
</tr>
<tr>
<td>( 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><strong>Name</strong></td>
<td><strong>Definition</strong></td>
<td></td>
</tr>
<tr>
<td>Procedure $p$, $Bl = {a, b, c, d}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ln_{Ep}$</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>$ln_{Sp}$</td>
<td>$\Phi_p(S_p)$</td>
<td>$(Bl_p - {b, c}) \cup {a, d}$</td>
</tr>
<tr>
<td>Procedure $q$, $Bl = {a, b, c, d}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ln_{Eq}$</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>$ln_{Sq}$</td>
<td>$\Phi_q(S_q)$</td>
<td>$(Bl_q - {a, b, c}) \cup {d}$</td>
</tr>
</tbody>
</table>
Result of Interprocedural Liveness Analysis

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

\[ c_1 \]
\[ \text{Call } p \]
\[ \{a, d\} \]

\[ n_1 \]
\[ a = a + 2 \]
\[ \text{print } c + d \]
\[ \{a, b, c, d\} \]

\[ n_2 \]
\[ d = a \times b \]
\[ \{a, b\} \]

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

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

\[ n_3 \]
\[ c = a + b \]
\[ \{a, b, c, d\} \]

\[ E_p \]
\[ \text{print } c + d \]
\[ \{a, b, c, d\} \]

\[ S_q \]
\[ a = 1 \]
\[ \{d\} \]

\[ c_3 \]
\[ \text{Call } p \]
\[ \{a, d\} \]

\[ E_q \]
\[ a = a \times b \]
\[ \{a, b, c, d\} \]
Context Sensitivity of Interprocedural Liveness Analysis

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

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

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

\[ c_1 \]
\[ \text{Call p} \]
\[ \{a, d\} \]

\[ n_1 \]
\[ a = a + 2 \]
\[ e = c + d \]
\[ \{a, b, c, d\} \]

\[ c_2 \]
\[ \text{Call q} \]
\[ \{a, b, e\} \]

\[ n_2 \]
\[ d = a \times b \]
\[ \{a, b, e\} \]

\[ n_3 \]
\[ c = a + b \]
\[ \{a, b, c, d, e\} \]

\[ c_3 \]
\[ \text{Call p} \]
\[ \{a, d, e\} \]

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

\[ E_p \]
\[ \text{print } c + d \]
\[ \{a, b, c, d, e\} \]

\[ E_q \]
\[ \text{print } a \times b \]
\[ \{d, e\} \]
Context Sensitivity of Interprocedural Liveness Analysis

\[ S_{main} \]
\[
\begin{align*}
  a &= 5; b = 3 \\
  c &= 7; \text{read } d
\end{align*}
\]
\[
\begin{array}{c}
\{a, d\} \\
\text{Call } p
\end{array}
\]
\[
\begin{align*}
  a &= a + 2 \\
  e &= c + d
\end{align*}
\]
\[
\begin{array}{c}
\{a, b, c, d\} \\
\text{n}_1
\end{array}
\]
\[
\begin{array}{c}
\{a, b, e\} \\
\text{n}_2
\end{array}
\]
\[
\begin{array}{c}
\{d, e\} \\
\text{d} = a \times b
\end{array}
\]
\[
\text{Call } q
\]
\[
\{a, c, e\}
\]
\[
E_{main} \quad \text{print } a + c + e
\]

\[ S_p \]
\[
\begin{align*}
  b &= 2 \\
\text{if } (b < d)
\end{align*}
\]
\[
\begin{array}{c}
\{a, b, d, e\} \\
\text{n}_3
\end{array}
\]
\[
\begin{align*}
  c &= a + b
\end{align*}
\]
\[
\begin{array}{c}
\{a, b, c, d, e\} \\
\text{c}_4
\end{array}
\]
\[
\text{Call } q
\]
\[
\{d, e\}
\]

\[ S_q \]
\[
\begin{align*}
  a &= 1
\end{align*}
\]
\[
\begin{array}{c}
\{a, d, e\} \\
\text{c}_3
\end{array}
\]
\[
\text{Call } p
\]
\[
\{a, b, c, d, e\}
\]

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

- \( f_p \) and \( f_q \) remain same
- \( e \in \text{In}_{S_p} \) 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
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 $S_{\text{main}}$

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 $\{\langle \sigma, x \rangle | \sigma \text{ is a call string }, x \in L\}$
  ▶ 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

Example diagram with labeled contexts and return values.
Interprocedural Validity and Calling Contexts

C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow R_2 \rightarrow C_3 \rightarrow R_3 \rightarrow C_4 \rightarrow R_4 \rightarrow C_5 \rightarrow R_5 \rightarrow C_6 \rightarrow R_6 \rightarrow C_7

C_1 \rightarrow C_4 \rightarrow R_4 \rightarrow C_3 \rightarrow R_3 \rightarrow C_6 \rightarrow R_6 \rightarrow C_4 \rightarrow R_4 \rightarrow C_1

C_1 \rightarrow C_4 \rightarrow R_4 \rightarrow C_1

C_1 \rightarrow C_3 \rightarrow R_3 \rightarrow C_6 \rightarrow R_6 \rightarrow C_4 \rightarrow R_4 \rightarrow C_1

C_1 \rightarrow C_4 \rightarrow R_4 \rightarrow C_1

C_1 \rightarrow C_3 \rightarrow R_3 \rightarrow C_6 \rightarrow R_6 \rightarrow C_4 \rightarrow R_4 \rightarrow C_1

C_1 \rightarrow C_4 \rightarrow R_4 \rightarrow C_1
Interprocedural Validity and Calling Contexts

• “You can descend only as much as you have ascended!”
Interprocedural Validity and Calling Contexts

- “You can descend only as much as you have ascended!”
- Every descending step must match a corresponding ascending step.
Interprocedural Validity and Calling Contexts

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

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

Every descending step must match a corresponding ascending step.

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

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

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

Every descending step must match a corresponding ascending step.

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

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

- Every descending step must match a corresponding ascending step.
- Calling context is represented by the remaining descending steps.
Interprocedural Validity and Calling Contexts

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

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

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

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

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

Ascend

Descend
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.

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

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

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

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

\[ E_{main} \]

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

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

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

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

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

\[ S_{main} \]
- 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 \( p \)

\[ R_2 \]

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

\[ E_p \]

May 2011

Uday Khedker
Available Expressions Analysis Using Call Strings Approach

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

May 2011
Available Expressions Analysis Using Call Strings Approach

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

Is a*b available?
Yes!
```

Diagram:

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

- **C_{1}**
  - call p

- **R_{1}**
  - Is a * b available?
    - Yes!

- **n_{1}**
  - print a * b

- **E_{main}**

- **S_{p}**
  - if a == 0

- **n_{2}**
  - a = a - 1

- **C_{2}**
  - call p

- **R_{2}**
  -

- **n_{3}**
  - t = a * b

- **E_{p}**
Available Expressions Analysis Using Call Strings Approach

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

\[ C_1 \]
call \( p \)

\[ R_1 \]

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

\[ E_{main} \]

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

\[ n_2 \]
a = a - 1

\[ C_2 \]
call \( p \)

\[ n_3 \]
t = a \times b

\[ E_p \]

Kill

May 2011
Uday Khedker
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 $p$

$R_1$

$n_1$

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

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

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

\[
R_1
\]

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

\[
E_{main}
\]

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

\[
n_2 : \text{a = a} - 1
\]

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

\[
R_2
\]

\[
n_3 : \text{t = a} \ast \text{b}
\]

\[
E_p
\]

May 2011 Uday Khedker
Available Expressions Analysis Using Call Strings Approach

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

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

\[ R_1 \]
- \( R_1 \) does nothing

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

\[ E_{\text{main}} \]
- \( E_{\text{main}} \) does nothing

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

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

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

\[ R_2 \]
- \( R_2 \) does nothing

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

\[ E_p \]
- \( E_p \) does nothing

\[ \text{Kill} \]
- \( \text{Kill} \)

\[ \text{Gen} \]
- \( \text{Gen} \)

May 2011
Uday Khedker
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

\( S_{main} \)

\( C_1 \) call \( p \)

\( R_1 \)

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

\( E_{main} \)

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

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

\( C_2 \) call \( p \)

\( R_2 \)

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

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

Maintain a worklist of nodes to be processed

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

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

\[ R_1 \]

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

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

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

\[ n_2 \]
\[
\text{a = a - 1}
\]

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

\[ R_2 \]

\[ n_3 \]
\[
\text{t = a * b}
\]

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

Maintain a worklist of nodes to be processed

\[ S_{main} \]
\[ \text{read } a, b \]
\[ t := a \times b \]
\[ \langle \lambda | 1 \rangle \]
\[ \langle c_1 | 1 \rangle \]
\[ C_1 \]
\[ \text{call } p \]

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

\[ E_{main} \]

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

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

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

\[ E_p \]
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 $p$

$S_p$

if $a == 0$

$n_2$

$a = a - 1$

$C_2$

call $p$

$R_1$

$n_1$

print $a \times b$

$E_{\text{main}}$

$\langle c_1|1 \rangle$

$\langle \lambda|1 \rangle$

$\langle c_1|1 \rangle$

$E_p$

$\langle c_1|1 \rangle$

$R_2$

$n_3$

$t = a \times b$
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

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

\[ C_1 \]
call \( p \)

\[ R_1 \]

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

\[ E_{main} \]

\[ \langle c_1|1 \rangle \]

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

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

\[ \langle c_1|0 \rangle \]

\[ C_2 \]
call \( p \)

\[ R_2 \]

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

\[ E_p \]

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

Maintain a worklist of nodes to be processed

\[ S_{main} \]

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

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

\[ C_1 \]

call \( p \)

\[ \langle c_1 | 1 \rangle \]

\[ S_p \]

if \( a == 0 \)

\[ n_2 \]

\[ a = a - 1 \]

\[ \langle c_1 | 0 \rangle \]

\[ C_2 \]

call \( p \)

\[ \langle c_1 | 1 \rangle \]

\[ R_1 \]

\[ n_1 \]

print \( a \times b \)

\[ \langle c_1 | 1 \rangle \]

\[ E_{main} \]

\[ n_3 \]

\[ t = a \times b \]

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

\[ E_p \]

May 2011

Uday Khedker
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 \( p \)

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

\[ R_1 \]

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

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

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

\[ \langle c_1|0 \rangle \]

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

\[ C_2 \] call \( p \)

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

\[ R_2 \]

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

\[ E_p \]

\[ \langle c_1|1 \rangle \]
\[ \langle c_1 c_2|0 \rangle \]
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

\[ S_{main} \]

\[
\begin{align*}
    \text{read } a, b \\
    t := a \times b
\end{align*}
\]

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

\[ C_1 \]

\begin{align*}
    \text{call } p
\end{align*}

\[ R_1 \]

\[ n_1 \]

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

\[ E_{main} \]

\[ \langle c_1 | 1 \rangle \]

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

\[ S_p \]

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

\[ n_2 \]

\[ a = a - 1 \]

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

\[ C_2 \]

\begin{align*}
    \text{call } p
\end{align*}

\[ R_2 \]

\[ n_3 \]

\[ t = a \times b \]

\[ E_p \]
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 $p$

$R_1$

print $a \times b$

$E_{main}$

$S_p$
if $a == 0$

$n_2$
$a = a - 1$

$C_2$
call $p$

$R_2$

$t = a \times b$

$E_p$

$\langle c_1 | 1 \rangle$
$\langle c_1 c_2 | 0 \rangle, \langle c_1 c_2 c_2 | 0 \rangle, \ldots$

$\langle \lambda | 1 \rangle$

$\langle c_1 | 1 \rangle$
$\langle c_1 c_2 | 0 \rangle$
$\langle c_1 c_2 c_2 | 0 \rangle$

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

May 2011

Uday Khedker
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

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

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

\[ C_1 \]
\[ \text{call } p \]
\[ \langle c_1 | 1 \rangle \]
\[ \langle c_1 c_2 | 0 \rangle, \langle c_1 c_2 c_2 | 0 \rangle, \ldots \]

\[ S_p \]
\[ \text{if } a == 0 \]
\[ n_2 \]
\[ a = a - 1 \]
\[ \langle c_1 | 0 \rangle, \langle c_1 c_2 | 0 \rangle, \ldots \]

\[ C_2 \]
\[ \text{call } p \]
\[ \langle c_1 c_2 | 0 \rangle, \langle c_1 c_2 c_2 | 0 \rangle, \ldots \]

\[ R_1 \]
\[ n_1 \]
\[ \text{print } a \times b \]
\[ \langle c_1 | 1 \rangle \]
\[ \langle c_1 c_2 | 0 \rangle \]
\[ \langle c_1 c_2 c_2 | 0 \rangle \]
\[ \ldots \]

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

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

\[ n_3 \]
\[ t = a \times b \]
\[ \langle c_1 c_2 | 0 \rangle, \langle c_1 c_2 c_2 | 0 \rangle, \ldots \]
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

\( S^\text{main} \)
- read \( a, b \)
  - \( t := a \ast b \)
- call \( p \)

\( C_1 \)

\( R_1 \)
- \( n_1 \)
  - print \( a \ast b \)

\( E^\text{main} \)

\( S_p \)
- if \( a == 0 \)
  - \( n_2 \)
    - \( a = a - 1 \)
  - \( C_2 \)
    - call \( p \)

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

\( E_p \)
- \( \langle c_1|1 \rangle \)
  - \( \langle c_1 c_2|0 \rangle, \langle c_1 c_2 c_2|0 \rangle, \ldots \)

\( n_2 \)
- \( \langle c_1|0 \rangle, \langle c_1 c_2|0 \rangle, \ldots \)

\( n_3 \)
- \( \langle c_1|1 \rangle \)
  - \( \langle c_1 c_2|1 \rangle \)
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

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

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

\[ R_1 \]
- \( \langle c_1 | 1 \rangle \)

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

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

\[ S_p \]
- \( \text{if } a == 0 \)

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

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

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

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

\[ E_p \]
- \( \langle c_1 | 1 \rangle \)
- \( \langle c_1 c_2 | 1 \rangle \)

- Maintain a worklist of nodes to be processed
Available Expressions Analysis Using Call Strings Approach

Maintain a worklist of nodes to be processed

\[ S_{main} \]

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

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

\[ C_1 \]

\[ \text{call } p \]

\[ \langle c_1 | 1 \rangle \]

\[ R_1 \]

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

\[ n_1 \]

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

\[ E_{main} \]

\[ \langle c_1 | 1 \rangle \]

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

\[ S_p \]

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

\[ \langle c_1 | 0 \rangle \]

\[ n_2 \]

\[ a = a - 1 \]

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

\[ C_2 \]

\[ \text{call } p \]

\[ \langle c_1 | 1 \rangle \]

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

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

\[ R_2 \]

\[ \langle c_1 | 0 \rangle \]

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

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

\[ n_3 \]

\[ t = a \times b \]

\[ \langle c_1 | 1 \rangle \]

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

\[ E_p \]
## 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><strong>IN_n</strong></td>
<td><strong>OUT_n</strong></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

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.    a = a * b;
11.   }
12. }
The Need for Multiple Occurrences of a Call Site

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

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

Path 1

3: Gen
4
7
8
7
12
9
10: Kill
11
12
5
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.     p();
11. }
12. }

Path 1

Path 2

3 : Gen
4
7
8
7
8
9
10 : Kill
11
12
5
9
10 : Kill
10 : Kill

May 2011
Uday Khedker
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. {
4.  c = a*b;
5.  p();
6. }
7. void p()
8. {
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.   {
   9.   Is a*b available?
   10.   a = a*b;
11. }
12. }
The Need for Multiple Occurrences of a Call Site

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

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

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

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

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

- Interprocedurally valid IFP

(C2, S_p, E_p, R2, n2, E_p, R2, n2)
The Need for Multiple Occurrences of a Call Site

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

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

- Interprocedurally valid IFP

C₂, S_p, C₂, S_p, E_p, R₂, \text{Kill} n₂, E_p, R₂, n₂
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

$$S_{main}, n_1, C_1, S_p, C_2, S_p, C_2, S_p, E_p, R_2, \text{Kill } n_2, E_p, R_2, n_2$$
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 up to the following length must be constructed
Terminating Call String Construction

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

- For recursive programs: Number of call strings could be infinite
  Fortunately, the problem is decidable for finite lattices.
  - All call strings up to the following length must be constructed
    - $K \cdot (|L| + 1)^2$ for general bounded frameworks
      ($L$ is the overall lattice of data flow values)
Terminating Call String Construction

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

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

⇒ 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 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 $m$.

\[
\{ C_{i_1} \cdot C_{i_2} \ldots C_{i_{m-1}} \mid x \}
\]

\[
C_a
\]

\[
R_a
\]
Classical Approximate Approach

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

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

\[ \downarrow \]

$C_a$

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

\[ \downarrow \]

$R_a$
Classical Approximate Approach

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

\begin{align*}
\text{Call string of length } m - 1 & \quad \langle C_{i_1} \cdot C_{i_2} \ldots C_{i_{m-1}} \mid x \rangle \\
\text{Call string of length } m & \quad \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 & \quad \text{May 2011 Uday Khedker}
\end{align*}
Classical Approximate Approach

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

\[
\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_{m-1}} \mid x \rangle \\
\quad \downarrow \\
C_a \\
\quad \downarrow \\
\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_{m-1}} \cdot C_a \mid x \rangle \\
\quad \downarrow \quad \downarrow \\
\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_{m-1}} \cdot C_a \mid y \rangle \\
\quad \downarrow \\
R_a \\
\quad \downarrow \\
\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_{m-1}} \mid y \rangle
\]
Classical Approximate Approach

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

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

\[ C_a \]

\[ R_a \]
Classical Approximate Approach

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

Call string of length $m$:

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

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

$$\langle C_{i_2} \ldots C_{i_m} \cdot c_a \mid x \rangle$$

$C_a$

$R_a$
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 $m$.

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

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

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

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

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

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

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

$C_a$

$R_a$
Classical Approximate Approach

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

\[
\langle C_i \circ C_{i_2} \cdots C_{i_m} \mid x_1 \rangle \quad \langle C_{j_1} \circ C_{i_2} \cdots C_{i_m} \mid x_2 \rangle
\]
Classical Approximate Approach

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

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

$C_a$

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

$R_a$
Classical Approximate Approach

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

$$\langle C_{i_1} \cdot C_{i_2} \ldots C_{i_m} \mid x_1 \rangle$$  $$\langle C_{j_1} \cdot C_{i_2} \ldots C_{i_m} \mid x_2 \rangle$$

$$\langle C_{i_2} \cdot C_{i_3} \ldots C_{i_m} \cdot C_a \mid x_1 \sqcap x_2 \rangle$$

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

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

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

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

\[
\langle C_{i_2} \cdot C_{i_3} \ldots C_{i_m} \cdot C_a \mid y \rangle
\]

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

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

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

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

- For simplicity, assume $m = 2$

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

\[C_a\]

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

- For simplicity, assume $m = 2$

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

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

\[
C_a
\]

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

- For simplicity, assume \( m = 2 \)

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

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

- For simplicity, assume $m = 2$

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

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

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

- For simplicity, assume $m = 2$

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

$C_a$

$R_a$
Approximate Call Strings in Presence of Recursion

• For simplicity, assume $m = 2$

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

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

$C_a$

$R_a$
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

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

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

$R_a$
Approximate Call Strings in Presence of Recursion

- For simplicity, assume $m = 2$

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

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

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

- For simplicity, assume $m = 2$

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

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

\[
\langle C_b \cdot C_a \mid y_1 \rangle, \quad \langle C_a \cdot C_a \mid y_2 \rangle
\]

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

- For simplicity, assume \( m = 2 \)

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

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

\[
\langle C_b \cdot C_a \mid y_1 \rangle, \langle C_a \cdot C_a \mid y_2 \rangle
\]

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

- For simplicity, assume \( m = 2 \)

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

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

\[
\langle C_b \cdot C_a \mid y_1 \rangle, \langle C_a \cdot C_a \mid y_2 \rangle
\]

\[
\langle C_b \mid y_1 \rangle \quad \langle C_b \cdot C_a \mid y_2 \rangle, \langle C_a \cdot C_a \mid y_2 \rangle
\]
Value Based Termination of Call String Construction

- 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.
Value Based Termination of Call String Construction

- 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.
Value Based Termination of Call String Construction

- 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.
Value Based Termination of Call String Construction

- 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.
Value Based Termination of Call String Construction

- 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.
Value Based Termination of Call String Construction

- Clearly identifies the exact set of call strings required.
- Value based termination of call string construction. No need to construct call strings upto 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.
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.