Interprocedural Data Flow Analysis

Uday P. Khedker

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

October 2011

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
- 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

- 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

S\_main
\[ a + b \]

Call p

E\_main

S\_p

n\_1 \ d = a + b

Call q

n\_3

E\_p

S\_q

\[ a = 1 \]

n\_2

\[ a = 1 \]

n\_4

\[ a = 1 \]

n\_2

\[ a = 1 \]

n\_4

\[ a = 1 \]

n\_2

\[ a = 1 \]

n\_4

Supergraphs of procedures

Call multi-graph

Oct 2011

IIT Bombay

Program Representation for Interprocedural Data Flow Analysis: Supergraph

S\_main
\[ a + b \]

Call p

C\_1

Call q

C\_2

E\_main

R\_1

R\_2

n\_3

n\_4

R\_3

R\_4

E\_p

n\_3

n\_4

S\_q

\[ d = a + b \]

n\_1

\[ a = 1 \]

n\_2

\[ a = 1 \]

n\_4

\[ a = 1 \]

n\_2

\[ a = 1 \]

n\_4

Oct 2011

IIT Bombay

Oct 2011

IIT Bombay

Oct 2011

IIT Bombay

Oct 2011

IIT Bombay
Validity of Interprocedural Control Flow Paths

Interprocedurally valid control flow path

Interprocedurally invalid control flow path
Validity of Interprocedural Control Flow Paths

Interprocedurally valid control flow path

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.

Context Sensitivity in Interprocedural 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

MFP computation restricted to valid paths only

x' = f_r(x) y' = f_r(y)
Staircase Diagrams of Interprocedurally Valid Paths

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

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

Increasing Precision in Data Flow Analysis

Flow insensitive
intraprocedural

Flow sensitive
intraprocedural

Context insensitive
flow insensitive

Context insensitive
flow sensitive

Context sensitive
flow sensitive

Context sensitive
flow sensitive

actually, only
caller sensitive

Classical 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
    $$\text{IN}_n = \bigcap_{(\sigma, x) \in \text{IN}_n} x$$
    $$\text{OUT}_n = \bigcap_{(\sigma, x) \in \text{OUT}_n} x$$

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

Interprocedural Validity and Calling Contexts

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

Overall Data Flow Equations

$$\text{IN}_n = \left\{ \begin{array}{ll}
\langle \lambda, BI \rangle & n \text{ is a } S_{\text{main}} \\
\bigcup_{p \in \text{pred}(n)} \text{OUT}_p & \text{otherwise}
\end{array} \right.$$  

$$\text{OUT}_n = \text{DepGEN}_n$$

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

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

(We merge underlying data flow values only if the contexts are same.)
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.

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

Available Expressions Analysis Using Call Strings Approach

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

Yes!

Available Expressions Analysis Using Call Strings Approach
Available Expressions Analysis Using Call Strings Approach

The diagram illustrates the process of available expressions analysis using call strings approach. It shows the flow of information from the main function, through the call to function $p$, and the subsequent return values and operations.

The main function $S_{\text{main}}$ reads $a, b$ and assigns $t := a * b$. Function $C_1$ calls function $p$ with argument $\lambda/1$. Function $C_2$ calls function $p$ with argument $\langle c_1|1 \rangle$. The diagram also includes the processing of expressions $n_1$, $n_2$, and $n_3$, involving $a, b, t$, and operations such as assignment and print.

Oct 2011

IIT Bombay
Available Expressions Analysis Using Call Strings Approach

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

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

The Need for Multiple Occurrences of a Call Site

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

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

The Need for Multiple Occurrences of a Call Site

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

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

The Need for Multiple Occurrences of a Call Site

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

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

The Need for Multiple Occurrences of a Call Site

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

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

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

In terms of staircase diagram

- Interprocedurally valid IFP
  \[ S_m, n_1, C_1, S_p, C_2, S_p, C_2, S_p, E_p, R_2, n_2 \]

- 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
- 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
  \[ \Rightarrow \text{Not a bound but prescribed necessary length} \]
  \[ \Rightarrow \text{Large number of long call strings} \]

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
  \[ CS(\rho) > M, \]
  \[ \exists \rho' = IVP(S_{main}, m) \text{ such that } CS(\rho') \leq M, \text{ and } f_\rho(BI) = f_{\rho'}(BI). \]
  \[ \Rightarrow \rho, \text{ the longer path, is redundant for data flow analysis} \]
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 $\Omega$:

\[
\begin{align*}
M & \quad \alpha_i \quad \beta_i \quad \beta_i \\
\rho & \quad \rho \\
\end{align*}
\]

When $\beta_i$ is $\Omega$:

\[
\begin{align*}
M & \quad \alpha_i \quad \beta_i \\
\rho & \quad \rho \\
\end{align*}
\]
Part 5

Value Based Termination of Call String Construction

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:
  - 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 \( u \).

\[
\begin{align*}
\text{represent}(⟨\sigma, x⟩, S_p) &= ⟨\text{shortest}(\sigma, S_p), x⟩ \\
\text{regenerate}(⟨\sigma, y⟩, E_p) &= \{⟨\sigma', y⟩ | \sigma \text{ and } \sigma' \text{ have the same value at } S_p\}
\end{align*}
\]

- 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 \( \rightarrow \) call nodes \( \rightarrow \) return nodes
These values are identical to the values computed by the full call strings method.
Safety and Precision of Representation and Regeneration

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

Represent

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

Stop regeneration after the values converge

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

Regenerate

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

\[
\langle \sigma \cdot \sigma_c^\omega | z_{m-i} \rangle \quad \langle \sigma \cdot \sigma_c^{\omega+1} | z_{m-i} \rangle
\]

Tutorial Problem for Value Based Termination of Call Strings

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

\[
S_{main} \quad C_1 \quad E_{main}
\]

\[
\langle c_1, c_2, 1 \rangle \quad \langle c_1, 1 \rangle \quad \langle c_2, c_2, 1 \rangle
\]

\[
n_1 \quad C_2 \quad R_2 \quad n_2 \quad a = a * b \quad E_p
\]

Work List Organization for Forward Analyses

- Maintain a stack of functions being processed
- Order the nodes in the work list in reverse post order
- Remove the head of work list for the procedure on top (say \(p\))
  - If the selected node is \(S_p\)
    - Perform representation
    - Insert \(E_p\) in the list for \(p\)
  - If the selected node is \(C_i\) calling procedure \(q\) then
    - Insert \(R_i\) in the list for \(p\)
    - Bring \(q\) on the top of stack
    - Insert \(S_q\) as the head of the list of \(q\)
  - If the selected node is \(E_p\)
    - Pop \(p\) from the stack
    - Perform regeneration
- The call strings and data flow values are manipulated normally

Work List Organization for Backward Analyses

- Swap the roles of \(S_p\) and \(E_p\)
- Swap the roles of \(C_i\) and \(R_i\)
- Replace reverse post order by post order
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
- $\sigma_c \equiv c_j c_k c_p c_i c_q$
- $\sigma_r \equiv r_q r_r f_k f_i f_j$

Assume that we allow up to $m$ occurrences of $\sigma_c$

Computing Data Flow Values along Recursive Paths

Fixed Bound Closure Bound of Flow Function

- $n > 0$ is the fixed point closure bound of $h : L \rightarrow L$ 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

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

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

Bounding the Call String Length Using Data Flow Values

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

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

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

Inductive step: \( g(z_{m-k}) \subseteq g(z_{m-\eta}) \quad \eta \leq j \leq (m-\omega) \)

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 \( \sigma \).

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 $f$

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

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

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

$x_i = \begin{cases} f^i(x_0) & i < \omega \\ f^\omega(x_0) & \text{otherwise} \end{cases}$ for a depth $j$ of recursion.

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

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 $\sigma_j = \ldots (C_1)^1 \ldots (C_1)^2 \ldots (C_1)^3 \ldots (C_1)^j \ldots$ $\sigma_{j+1} = \ldots (C_1)^1 \ldots (C_1)^2 \ldots (C_1)^3 \ldots (C_1)^j \ldots (C_1)^{j+1} \ldots$
  - Represent $\langle \sigma_j | x_j \rangle$ and $\langle \sigma_{j+1} | x_{j+1} \rangle$
    by $\langle \sigma_j | x_j \cap x_{j+1} \rangle$
- Context sensitive for a depth $j$ of recursion.
- Context insensitive beyond that.
- Assumption: Height of the lattice is finite.

Worst Case Length Bound

- Consider a call string $\sigma = \ldots (C_1)^1 \ldots (C_1)^2 \ldots (C_1)^3 \ldots (C_1)^j \ldots$ 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 $C_i$ 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 $= K \times (|L| + 1)^2$

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>hanoi</td>
<td>33</td>
<td>2</td>
<td>4</td>
<td>100000+ 99922</td>
<td>3973 x 10^1</td>
</tr>
<tr>
<td>bit_gray</td>
<td>53</td>
<td>11</td>
<td>17</td>
<td>100000+ 31374</td>
<td>2705 x 10^1</td>
</tr>
<tr>
<td>analyzer</td>
<td>288</td>
<td>14</td>
<td>20</td>
<td>21</td>
<td>4</td>
</tr>
<tr>
<td>distray</td>
<td>331</td>
<td>9</td>
<td>21</td>
<td>96</td>
<td>28</td>
</tr>
<tr>
<td>mason</td>
<td>350</td>
<td>9</td>
<td>13</td>
<td>100000+ 22143</td>
<td>432 x 10^1</td>
</tr>
<tr>
<td>fourinarow</td>
<td>676</td>
<td>17</td>
<td>45</td>
<td>510</td>
<td>158</td>
</tr>
<tr>
<td>sim</td>
<td>1146</td>
<td>13</td>
<td>45</td>
<td>100000+ 33546</td>
<td>1427 x 10^1</td>
</tr>
<tr>
<td>mcf</td>
<td>1299</td>
<td>17</td>
<td>24</td>
<td>32789</td>
<td>32767</td>
</tr>
<tr>
<td>256_courses</td>
<td>3320</td>
<td>63</td>
<td>198</td>
<td>492</td>
<td>63</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.

Oct 2011

Tutorial Problem on Interprocedural Liveness Analysis

main()
{
    a = 5; b = 3;
    c = 7; d = 2;
    /*{a,d}*/
    p(); /*{a,d,e}*/
    b = 2;
    /*{a,b,c,d}*/
    if (b<d)
        /*{a,b,d,e}*/
        c = a+b;
    else
        /*{a,d}*/
        a = a + 2;
    e = c+d;
    /*{a,b,e}*/
    d = a*b;
    /*{a,b,c,d}*/
    q(); /*{a,b,c,d,e}*/
    q(); /*{d,e}*/
    /*{a,d,e}*/
    print a+c+e;
}

p()
{
    b = 2;
    p();
    a = a + 2;
    e = c+d;
    d = a*b;
    q();
    print a+c+e;
}

q()
{
    a = 1;
    p();
    a = a*b;
}

Context sensitivity: e is live on entry to p but not before its call in main

Result of Interprocedural Liveness Analysis

main()
{
    a = 5; b = 3;
    c = 7; d = 2;
    /*{a,d}*/
    p(); /*{a,d,e}*/
    b = 2;
    /*{a,b,c,d}*/
    if (b<d)
        /*{a,b,d,e}*/
        c = a+b;
    else
        /*{a,d}*/
        a = a + 2;
    e = c+d;
    /*{a,b,e}*/
    d = a*b;
    /*{a,b,c,d}*/
    q(); /*{a,b,c,d,e}*/
    q(); /*{d,e}*/
    /*{a,d,e}*/
    print a+c+e;
}

Tutorial Problem on Interprocedural Points-to Analysis

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

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

q()
{
    x = &y;
    z = &x;
    p(); /* C1 */
}

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

Number of distinct call sites in a call chain
K = 2.

Number of variables: 3

Number of distinct points-to pairs: 3 × 3 = 9

L is powerset of all points-to pairs

| L | = 2^9

Length of the longest call string in Sharir-Pnueli method
2 × (|L| + 1)^2 = 2^{10} + 2^{10} + 1 = 5, 25, 313

All call strings upto this length must be constructed by the Sharir-Pnueli method!
Tutorial Problem on Interprocedural Points-to Analysis

main()
{ x = &y;
  z = &x;
  y = &z;
  p(); /* C1 */
}
p()
{ if (....)
  { p(); /* C2 */
    x = *x;
  }
}