Interprocedural Data Flow Analysis

Uday Khedker

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

October 2017

Copyright

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


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


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

Outline

- Issues in interprocedural analysis
- Functional approach
- Classical call strings approach
- Value context based approach
Part 2

Issues in Interprocedural Analysis

Interprocedural Analysis: Overview

- Extends the scope of data flow analysis across procedure boundaries
  - Incorporates the effects of
    - procedure calls in the caller procedures, and
    - calling contexts in the callee procedures
- Approaches:
  - Generic: Call strings approach, functional approach
  - Problem specific: Alias analysis, Points-to analysis, Partial redundancy elimination, Constant propagation

Why Interprocedural Analysis?

- 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?
- 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

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

### Bottom-up approach
- Traverses the call graph bottom up
- Computes a parameterized summary of each callee
- Can be viewed as procedure inlining
  - Summary is inlined at the all site, not the entire procedure body

### Top-down approach
- Traverses the call graph top down
- Needs to visit a procedure separately for every calling context
- Can be viewed as procedure inlining
Top-down Vs. Bottom-up Interprocedural Analysis

Top-down Analysis for Available Expressions Analysis

- Expression $b + c$ is available in procedure $p$
- Expression $a \times b$ is not available in procedure $p$

Bottom-Up Analysis for Available Expressions Analysis

- Call is replaced by procedure summary
- Using procedure summary of $g$ at call sites

Expressions $b + c$ and $c \times d$ are available in procedure $r$
Top-down Vs. Bottom-up Interprocedural Analysis

Bottom-Up Analysis for Available Expressions Analysis

Call is replaced by procedure summary

Expressions b + c and c * d are available in procedure r

Issues in Top-down Vs. Bottom-up Interprocedural Analysis

- Bottom-up approach
  - Compact representation
  - Information may depend on the calling context
- Top-down approach
  - Exponentially large number of calling contexts
  - Many contexts may have no effect on the procedure

Validity of Interprocedural Control Flow Paths
Validity of Interprocedural Control Flow Paths

Interprocedurally invalid control flow path

Interprocedurally valid control flow path

Soundness, Precision, and Efficiency of Data Flow Analysis

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

A path which yields information that affects the summary information

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

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

- Context sensitivity is all about returning the right value from a callee to the right caller, and
- not about passing the right value from a caller to the right callee.

Increasing Precision in Data Flow Analysis

- Flow insensitive
  - intraprocedural
  - context insensitive
- Flow sensitive
  - intraprocedural
  - context sensitive
  - context insensitive
- Context sensitive
  - flow sensitive
  - flow insensitive

Actually, only caller sensitive.

Functional Approach

Part 3

Classical Functional Approach
Functional Approach

- Bottom-up Approach
- Compute summary flow functions for each procedure
- Use summary flow functions as the flow function for a call block
- Main challenge: Appropriate representation for summary flow functions

Notation for Summary Flow Function

For simplicity forward flow is assumed

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

Equations for Constructing Summary Flow Functions

For simplicity forward flow is assumed. $I_n$ is Entry of $n$, $O_n$ is Exit of $n$

$$
\Phi_r(I_n) = \begin{cases} 
\phi_{id} & \text{if } n \text{ is } S_r \\
\prod_{p \in \text{pred}(n)} (\Phi_r(O_p)) & \text{otherwise}
\end{cases}
$$

$$
\Phi_r(O_n) = \begin{cases} 
\Phi_s(u) \circ \Phi_r(I_n) & \text{if } n \text{ calls procedure } s \\
f_n \circ \Phi_r(I_n) & \text{otherwise}
\end{cases}
$$

The summary flow function of a given procedure $r$

- is influenced by summary flow functions of the callees of $r$
- is not influenced by summary flow functions of the callers of $r$

Fixed point computation may be required in the presence of loops or recursion

Constructing Summary Flow Functions Iteratively

Iteration #1

$$
\Phi_r(u_1) = \phi_{id} \\
\Phi_r(u_2) = f_1 \\
\Phi_r(u_3) = f_1 \\
\Phi_r(u_4) = f_2 \circ f_1 \\
\Phi_r(u_5) = f_1 \\
\Phi_r(u_6) = f_3 \circ f_1 \\
\Phi_r(u_7) = f_2 \circ f_1 \cap f_3 \circ f_1 \\
\Phi_r(u_8) = f_4 \circ (f_2 \circ f_1 \cap f_3 \circ f_1)
$$
Constructing Summary Flow Functions Iteratively

\[ r \]

Iteration #2

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

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

Termination is possible only if all function compositions and confluenes 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}, \hat{\bot}, \hat{\top}_n )</td>
<td>( \hat{\top} )</td>
</tr>
<tr>
<td>( \hat{\bot} = {a} )</td>
<td>( \hat{\bot}_n(x), \forall x \in {\hat{\top}, \hat{\bot}} )</td>
<td>( \hat{\bot}_n )</td>
</tr>
</tbody>
</table>

Reducing Component Flow Functions for Live Variables Analysis

Let \( \hat{\phi} \in \{\hat{\top}, \hat{\bot}, \hat{\top}_n\} \) and \( x \in \{1, 0\} \). Then,

- \( \hat{\top} \cap \hat{\phi} = \hat{\phi} \) (because \( 0 + x = x \))
- \( \hat{\bot} \cap \hat{\phi} = \hat{\bot} \) (because \( 1 + x = 1 \))
- \( \hat{\top} \circ \hat{\phi} = \hat{\top} \) (because \( \hat{\top} \) is a constant function)
- \( \hat{\bot} \circ \hat{\phi} = \hat{\bot} \) (because \( \hat{\bot} \) is a constant function)
- \( \hat{\bot}_n \circ \hat{\phi} = \hat{\phi} \) (because \( \hat{\bot}_n \) is the identity function)
Reducing Function Compositions in Bit Vector Frameworks

Kill\(_n\) denoted by \(K_n\) and Gen\(_n\) denoted by \(G_n\)

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

Hence,

\[
K_3 = K_1 \cup K_2 \\
G_3 = (G_1 - K_2) \cup G_2
\]

Reducing Bit Vector Flow Function Confluences (1)

Kill\(_n\) denoted by \(K_n\) and Gen\(_n\) denoted by \(G_n\)

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

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

Hence,

\[
K_3 = K_1 \cap K_2 \\
G_3 = G_1 \cup G_2
\]

Reducing Bit Vector Flow Function Confluences (2)

Kill\(_n\) denoted by \(K_n\) and Gen\(_n\) denoted by \(G_n\)

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

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

Hence,

\[
K_3 = K_1 \cup K_2 \\
G_3 = G_1 \cap G_2
\]

Lattice of Flow Functions for Live Variables Analysis

Flow functions for two variables

- Product of lattices for independent variables (because of separability)

<table>
<thead>
<tr>
<th>Lattice of data flow values</th>
<th>All possible functions</th>
<th>Lattice of flow functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\top = 0)</td>
<td>({})</td>
<td>({})</td>
</tr>
<tr>
<td>({a})</td>
<td>({b})</td>
<td>(\phi_I)</td>
</tr>
<tr>
<td>(\bot = {a, b})</td>
<td>({a, b})</td>
<td>(\phi_{II})</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{Gen}_n & : \{\}\, \{b\} \\
\text{Kill}_n & : \{b\} \\
f_n & : \{a\} \\
\text{Gen}_n & : \{a\} \\
\text{Kill}_n & : \{a\} \\
f_n & : \{a, b\}
\end{align*}
\]
An Example of Interprocedural Liveness Analysis

\[ S_{main} \]
\[
\begin{array}{c}
a = 5; b = 3 \\
c = 7; \text{read } d
\end{array}
\]
\[ S_p \]
\[
\begin{array}{c}
b = 2 \\
\text{if } (b < d)
\end{array}
\]
\[ c_1 \]
\[ \text{Call } p \]
\[ n_3 \]
\[
\begin{array}{c}
c = a + b
\end{array}
\]
\[ c_4 \]
\[ \text{Call } q \]
\[ E_p \]
\[
\begin{array}{c}
\text{print } c + d
\end{array}
\]
\[ n_2 \]
\[ d = a * b \]
\[ c_2 \]
\[ \text{Call } q \]
\[ E_{main} \]
\[
\begin{array}{c}
\text{print } a + c + e
\end{array}
\]

Computed Summary Flow Functions

\[ S_p \]
\[
\begin{array}{c}
b = 2 \\
\text{if } (b < d)
\end{array}
\]
\[ n_3 \]
\[
\begin{array}{c}
c = a + b
\end{array}
\]
\[ c_4 \]
\[ \text{Call } q \]
\[ E_p \]
\[
\begin{array}{c}
\text{print } c + d
\end{array}
\]
\[ S_q \]
\[
\begin{array}{c}
a = 1
\end{array}
\]
\[ c_3 \]
\[ \text{Call } p \]
\[ E_q \]
\[
\begin{array}{c}
a = a * b
\end{array}
\]

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>Phi_p(E_p)</td>
<td>f_{E_p}</td>
<td>{c, d}</td>
<td>()</td>
</tr>
<tr>
<td></td>
<td>Phi_p(n_3)</td>
<td>f_{n_3} \cap 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_{c_4} \cap Phi_p(E_p) = empty</td>
<td>0</td>
<td>{a, b, c, d, e}</td>
</tr>
<tr>
<td></td>
<td>Phi_p(S_p)</td>
<td>f_{S_p} \cap Phi_p(n_3)</td>
<td>{a, d}</td>
<td>{b, c}</td>
</tr>
<tr>
<td>f_p</td>
<td>Phi_p(S_p)</td>
<td>{a, d}</td>
<td>{b, c}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Phi_q(c_3)</td>
<td>f_{c_3} \cap Phi_q(E_q)</td>
<td>{a, d}</td>
<td>{a, b, c}</td>
</tr>
<tr>
<td></td>
<td>Phi_q(c_3)</td>
<td>f_{c_3} \cap 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} \cap Phi_q(c_3)</td>
<td>{d}</td>
<td>{a, b, c}</td>
</tr>
<tr>
<td>f_q</td>
<td>Phi_q(S_q)</td>
<td>{d}</td>
<td>{a, b, c}</td>
<td></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>(L_{E_n})</td>
<td>(\Phi_m(E_m))</td>
<td>(B_{I_m} \cup {a, c, e})</td>
</tr>
<tr>
<td>(L_{c_2})</td>
<td>(\Phi_m(c_2))</td>
<td>(B_{I_m} \setminus {a, b, c} \cup {d, e})</td>
</tr>
<tr>
<td>(L_{n_2})</td>
<td>(\Phi_m(n_2))</td>
<td>(B_{I_m} \setminus {a, b, c, d} \cup {a, b, e})</td>
</tr>
<tr>
<td>(L_{n_1})</td>
<td>(\Phi_m(n_1))</td>
<td>(B_{I_m} \setminus {a, b, c, d, e} \cup {a, b, c, d})</td>
</tr>
<tr>
<td>(L_{c_1})</td>
<td>(\Phi_m(c_1))</td>
<td>(B_{I_m} \setminus {a, b, c, d, e} \cup {a, d})</td>
</tr>
<tr>
<td>(L_{S_m})</td>
<td>(\Phi_m(S_m))</td>
<td>(B_{I_m} \setminus {a, b, c, d, e})</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>InEn_1</td>
<td>φ_p(E_p)</td>
<td>{a, b, c, d, e}</td>
</tr>
<tr>
<td>InEn_2</td>
<td>(Bl_p - {c}) U {a, b, d}</td>
<td>{a, b, d, e}</td>
</tr>
<tr>
<td>InEn_3</td>
<td>(Bl_p - {a, b, c}) U {d}</td>
<td>{d, e}</td>
</tr>
<tr>
<td>InEn_4</td>
<td>(Bl_p - {a, b, c}) U {a, d}</td>
<td>{a, b, d, e}</td>
</tr>
</tbody>
</table>

Procedure p, Bl = {a, b, c, d, e}

Context Sensitivity of Interprocedural Liveness Analysis

Context Sensitivity of Interprocedural Liveness Analysis

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

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

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

\[
S_{n_1} : a = a + 2; e = c + d
\]

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

Explaining Context Sensitivity

- Flow function of procedure p is identity with respect to variable e
- Is e live in the body of procedure p?
  - During the analysis: Depends on the calling context
  - After the analysis: Yes (static approximation across all executions)
- Distinction between caller’s effect on callee and callee’s effect on caller
Tutorial Problem #1

Perform interprocedural live variables analysis for the following program

```c
main()
{
p();
}
p()
{ while (c < 10)
{ p();
a = a*b;
}
}
```

Tutorial Problem #2: Summary Flow Function for Constant Propagation

```
b = c + d

S_p

n1 a = a + b

C1 Call p

n2 a = a - b

Ep

<table>
<thead>
<tr>
<th>Iter. #1</th>
<th>Iter. #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Phi_p(S_p))((\langle v_a, v_b \rangle ))</td>
<td>(\langle v_a, v_b \rangle )</td>
</tr>
<tr>
<td>(\Phi_p(n_1))((\langle v_a, v_b \rangle ))</td>
<td>(\langle v_a + v_b, v_b \rangle )</td>
</tr>
<tr>
<td>(\Phi_p(C_1))((\langle v_a, v_b \rangle ))</td>
<td>(\langle \hat{\top}, \hat{\top} \rangle )</td>
</tr>
<tr>
<td>(\Phi_p(n_2))((\langle v_a, v_b \rangle ))</td>
<td>(\langle \hat{\top}, \hat{\top} \rangle )</td>
</tr>
<tr>
<td>(\Phi_p(E_p))((\langle v_a, v_b \rangle ))</td>
<td>(\langle v_a, v_b \rangle )</td>
</tr>
</tbody>
</table>
```

Will this work always?

Tutorial Problem #3

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

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

Limitations of Functional Approach to Interprocedural Data Flow Analysis

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

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

Separable

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

General Non-Separable

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

Entity Functions in Points-to Analysis

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

Entity Functions in Constant Propagation

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

Enumeration Based Functional Approach

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

Requires the number of values to be finite
Part 4

**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

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

Calling context is represented by the remaining descending steps
Interprocedural Data Flow Analysis Using Call Strings

- Augmented data flow information
  - $IN_n$ and $OUT_n$ are partial maps from call strings to $L$
  - The final data flow information at a program point is
    \[
    \begin{align*}
    IN_n &= \bigcap_{(\sigma, x) \in IN_n} x \\
    OUT_n &= \bigcap_{(\sigma, x) \in OUT_n} x
    \end{align*}
    \]
    (glb of data flow values for all call strings)

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

Augmented Data Flow Equations: Computing $IN_n$

\[
IN_n = \begin{cases} 
\langle \lambda, BI \rangle & n \text{ is a } S_{main} \\
\bigoplus_{p \in \text{pred}(n)} OUT_p & \text{otherwise}
\end{cases}
\]

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

\[
\Gamma_1 \sqcup \Gamma_2 = \{ (\sigma, z) | (\sigma, x) \in \Gamma_1 \wedge (\sigma, y) \in \Gamma_2 \Rightarrow z = x \cap y, \\
(\sigma, x) \in \Gamma_1 \wedge (\sigma, y) \not\in \Gamma_2 \Rightarrow z = x, \\
(\sigma, x) \not\in \Gamma_1 \wedge (\sigma, y) \in \Gamma_2 \Rightarrow z = y \}
\]

Augmented Data Flow Equations: Computing $OUT_n$

- Call node $C_i$
  - Append $c_i$ to every $\sigma$
  - Propagate the data flow values unchanged
  
- Return node $R_i$
  - If the last call site is $c_i$, remove it and propagate the data flow value unchanged
  - Block other data flow values

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

Available Expressions Analysis Using Call Strings Approach

- $S_{main}$
  - $t := a * b$

- $S_p$
  - if $a == 0$

- $S_{p}$
  - $a = a - 1$

- $S_{p}$
  - $t = a * b$

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

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

Yes!

Oct 2017 IIT Bombay
Perform available expressions analysis for the following program

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

\[ n \]
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;
   p();
4. }
5. }
6. void p()
7. {
   if (...)
8. {
   p();
9. /*Is a*b available?*/
10. a = a * b;
11. }
12. }

Recursive calls: 1

Path 1: 3 Gen 4 : 7 : 8 : 7 : 8 : 7 : 5
Path 2: 3 Gen 4 : 7 : 8 : 7 : 8 : 7 : 5

Recursive calls: 2

• Interprocedurally valid IFP

\[ S_{main}, n_1, C_1, S_p, C_2, S_p, C_2, S_p, E_p, R_2, n_2, E_p, R_2, n_2 \]

Path 1:
- C2
- R2
- R1

Path 2:
- C2
- R2
- R1

• 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

Oct 2017

Tutorial Problem #2

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

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

Oct 2017
Terminating Call String Construction

- For non-recursive programs: Number of call strings is finite
- For recursive programs: Number of call strings could be infinite
  Fortunately, the problem is decidable for finite lattices
  - All call strings up to the following length must be constructed
    - $K \cdot (|L| + 1)^2$ for general bounded frameworks
      ($L$ is the overall lattice of data flow values)
    - $K \cdot (|\hat{L}| + 1)^2$ for separable bounded frameworks
      ($\hat{L}$ is the component lattice for an entity)
    - $K \cdot 3$ for bit vector frameworks
    - 3 occurrences of any call site in a call string for bit vector frameworks
  - Not a bound but prescribed necessary length
  - Large number of long call strings

Classical Call String Length

Sharir-Pnueli [1981]
- Consider the smallest prefix $\rho_0$ of $\rho$ such that $CS(\rho_0) > M$
- Consider a triple $\langle c_i, \alpha_i, \beta_i \rangle$ where
  - $\alpha_i$ is the data flow value reaching call node $C_i$ along $\rho$ and
  - $\beta_i$ is the data flow value reaching the corresponding return node $R_i$ along $\rho$
  - If $R_i$ is not in $\rho$, then $\beta_i = \Omega$ (undefined)

Classical Call String Length

- Notation
  - $IVP(n, m)$: Interprocedurally valid path from block $n$ to block $m$
  - $CS(\rho)$: Number of call nodes in $\rho$ that do not have the matching return node in $\rho$
    (length of the call string representing $IVP(n, m)$)
- Claim
  Let $M = K \cdot (|L| + 1)^2$ where $K$ is the number of distinct call sites in any call chain
  Then, for any $\rho = IVP(S_{main}, m)$ such that $CS(\rho) > M$
  $\exists \rho' = IVP(S_{main}, m)$ such that $CS(\rho') \leq M$, and $f_\rho(Bl) = f_{\rho'}(Bl)$
  $\Rightarrow \rho$, the longer path, is redundant for data flow analysis
Number of distinct triples \( \langle c_i, \alpha_i, \beta_i \rangle \) is \( M = K \cdot (|L| + 1)^2 \).

There are at least two calls from the same call site that have the same effect on data flow values.
Classical Call String Length

When $\beta_i$ is not $\Omega$

\[ M \]

$\alpha_i$ $\beta_i$

$\rho'$

Classical Call String Length

When $\beta_i$ is $\Omega$

\[ M \]

$\alpha_i$

$\rho_0$ $\rho$

Tighter Bound for Bit Vector Frameworks

- $\hat{L}$ is $\{0, 1, \ldots\}$, $L$ is $\{0, 1\}^m$
- $\otimes$ is either boolean AND or boolean OR
- $\hat{\top}$ and $\hat{\bot}$ are 0 or 1 depending on $\otimes$
- $\hat{h}$ is a bit function and could be one of the following:

<table>
<thead>
<tr>
<th>Raise</th>
<th>Lower</th>
<th>Propagate</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{\top}$</td>
<td>$\hat{\bot}$</td>
<td>$\hat{\top}$</td>
</tr>
<tr>
<td>$\hat{\bot}$</td>
<td>$\hat{\top}$</td>
<td>$\hat{\bot}$</td>
</tr>
</tbody>
</table>
Tighter Bound for Bit Vector Frameworks

Karkare Khedker 2007

- Validity constraints are imposed by the presence of return nodes
- For every cyclic path consisting on Propagate functions, there exists an acyclic path consisting of Propagate functions
- Source of information is a Raise or Lower function
- Target is a point reachable by a series of Propagate functions
- Identifies interesting path segments that we need to consider for determining a sufficient set of call strings

Relevant Path Segments for Tighter Bound for Bit Vector Frameworks

- All paths from $C_i$ to $R_i$ are abstracted away when a call node $C_j$ is reached after $R_i$
- Consider maximal interprocedurally valid paths in which there is no path from a return node to a call node
Relevant Path Segments for Tighter Bound for Bit Vector Frameworks

Consider all four combinations

- **Case A**: Source is a call node and target is a call node
- **Case B**: Source is a call node and target is a return node
- **Case C**: Source is a return node and target is also a return node
- **Case D**: Source is a return node and target is a call node: Not relevant

Tighter Length for Bit Vector Frameworks

Case A:

*Source* is a call node and *target* is also a call node $P(I \leadsto C \leadsto C_T)$

- No return node, no validity constraints
- Paths $P(I \leadsto C_S)$ and Paths $P(C_S \leadsto C_T)$ can be acyclic
- A call node may be common to both segments
- At most 2 occurrences of a call site

Case B:

*Source* is a call node $C_S$ and *target* is some return node $R_T$

- $P(I \leadsto C_T \leadsto C_S \leadsto R_S \leadsto R_T)$
  - Call strings are derived from the paths $P(I \leadsto C_S \leadsto C_T \leadsto C_L)$ where $C_L$ is the last call node
  - Thus there are three acyclic segments $P(I \leadsto C_S)$, $P(C_S \leadsto C_T)$, and $P(C_T \leadsto C_L)$
  - A call node may be shared in all three
    ⇒ At most 3 occurrences of a call site
- $P(I \leadsto C_T \leadsto C_S \leadsto R_S \leadsto R_T)$
  - $C_T$ is required because of validity constraints
  - Call strings are derived from the paths $P(I \leadsto C_T \leadsto C_S \leadsto C_L)$ where $C_L$ is the last call node
  - Again, there are three acyclic segments and at most 3 occurrences of a call site

Case C:

*Source* is a return node $R_S$ and *target* is also some return node $R_T$

- $P(I \leadsto C_T \leadsto C_S \leadsto R_S \leadsto R_T)$
- $C_T$ and $C_S$ are required because of validity constraints
- Call strings are derived from the paths $P(I \leadsto C_T \leadsto C_S \leadsto C_L)$ where $C_L$ is the last call node
- Again, there are three acyclic segments and at most 3 occurrences of a call site
Classical Approximate Call Strings Approach

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

Call string of length $m - 1$:

\[ \langle C_i \cdot C_{i-1} \cdots C_0 \mid x \rangle \]

Call string of length $m$:

\[ \langle C_i \cdot C_{i-1} \cdots C_0 \cdot C_a \mid x \rangle \]
\[ \langle C_i \cdot C_{i-1} \cdots C_0 \cdot C_a \mid y \rangle \]

\[ \langle C_i \cdot C_{i-1} \cdots C_0 \mid y \rangle \]

Approximate Call Strings in Presence of Recursion

• For simplicity, assume $m = 2$

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

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

- For simplicity, assume $m = 2$

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

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

$R_a$
Approximate Call Strings in Presence of Recursion

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

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

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

\[
\langle C_a \cdot C_a \mid x_4 \rangle
\]

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

- For simplicity, assume $m = 2$

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

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

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

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

Part 5

Value Contexts: Key Ideas

Consider call chains $\sigma_1$ and $\sigma_2$ reaching $S_p$

- Data flow value invariant:
  - If the data flow reaching $S_p$ along $\sigma_1$ and $\sigma_2$ are identical, then
    - the data flow values reaching $E_p$ for the two contexts will also be identical
  - We can reduce the amount of effort by using
    - Data flow values at $S_p$ as value contexts
    - Maintaining distinct data flow values in $p$ for each value context

Interprocedural Data Flow Analysis Using Value Contexts

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

Separate contexts are created for each unique data flow value.

New contexts are created for data flow values reaching \( q \).
Context transitions on call sites are recorded globally.

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

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

Interprocedural Data Flow Analysis Using Value Contexts

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

  | $⟨p, v⟩$ | Context for procedure $p$ with data flow value $v$ |
  | $X|m$ | Work list entry for context $X$ for node $m$ |
  | $X.v$ | Data flow value in context $X$ is $v$ |
  | $Out_m[X]$ | Data flow value of context $X$ in $Out_m$ |
  | $X \overset{<}{\rightarrow} Y$ | Transition from context $X$ to context $Y$ at call site $C_i$ |

Defining Value Contexts

Interprocedural Data Flow Analysis Using Value Contexts: An Overview

- Select $X|n$ from $WL$. Compute $In_n$.
  - If $n = C_i$ calling procedure $p$
    - Propagate $In_n$ to appropriate value context of the callee procedure $p$
  - If $n = E_p$
    - Propagate $In_n$ to appropriate value contexts of the callers of $p$
  - If $n$ is some other node
    - Compute $Out_n$
  - Update $WL$
- Repeat until $WL$ is empty
Interprocedural Data Flow Analysis Using Value Contexts (2)

Select \( X|n \) from WL. Compute \( In_n \). Let \( X.v \) be in \( In_n \)

- If \( n = C_i \) calling procedure \( p \)
  - If some context \( \langle p,v \rangle \) exists (say \( Y \)) /* \( p \) is the callee */
    - record the transition \( X \xrightarrow{C} Y \)
    - \( Out_{C_i}[X] = Out_{C_i}[X] \cap exitValue(Y) \)
    - if there is a change, add \( X|m, \forall m \in succ(C_i) \) to WL
  - If it does not exist
    - create a new context \( Y = \langle p,v \rangle \) /* \( p \) is the callee */
    - initialize \( exitValue(Y) = \top \)
    - record the transition \( X \xrightarrow{i} Y \)
    - initialize \( Out_m[Y] = \top \) for all nodes \( m \) of procedure \( p \)
    - add entries \( Y|m \) for all nodes \( m \) of procedure \( p \) to WL

Interprocedural Data Flow Analysis Using Value Contexts (3)

Select \( X|n \) from WL. Compute \( In_n \). Let \( X.v \) be in \( In_n \)

- If \( n = E_p \)
  - Set \( exitValue(X) = v \) /* \( E_p \) is an empty block */
  - Find out all transitions \( Z \xrightarrow{C} X \)
    - Set \( Out_{C}[Z] = Out_{C}[Z] \cap v \)
    - If there is a change, add \( Z|m, \forall m \in succ(C) \) to WL
- For all other nodes
  - Set \( Out_n[X] = f_i(v) \)
  - If there is a change, add \( X|m, \forall m \in succ(n) \) to WL

Available Expressions Analysis Using Value Contexts

\[
\begin{align*}
S_{main} \quad & \text{read } a, b, \\
& t := a \ast b \\
C_1 \quad & \text{call } p \\
& \text{if } a == 0 \\
n_1 \quad & \text{print } a \ast b \\
n_2 \quad & a = a - 1 \\
n_3 \quad & t = a \ast b \\
E_{main} \quad & \\
S_p \quad & \\
\end{align*}
\]

\[
\begin{align*}
S_{main} \quad & \text{read } a, b, \\
& t := a \ast b \\
C_1 \quad & \text{call } p \\
& \text{if } a == 0 \\
n_1 \quad & \text{print } a \ast b \\
n_2 \quad & a = a - 1 \\
n_3 \quad & t = a \ast b \\
E_{main} \quad & \\
S_p \quad & \\
\end{align*}
\]

Available Expressions Analysis Using Value Contexts

\[
\begin{align*}
S_{main} \quad & \text{read } a, b, \\
& t := a \ast b \\
C_1 \quad & \text{call } p \\
& \text{if } a == 0 \\
n_1 \quad & \text{print } a \ast b \\
n_2 \quad & a = a - 1 \\
n_3 \quad & t = a \ast b \\
E_{main} \quad & \\
S_p \quad & \\
\end{align*}
\]

\[
\begin{align*}
S_{main} \quad & \text{read } a, b, \\
& t := a \ast b \\
C_1 \quad & \text{call } p \\
& \text{if } a == 0 \\
n_1 \quad & \text{print } a \ast b \\
n_2 \quad & a = a - 1 \\
n_3 \quad & t = a \ast b \\
E_{main} \quad & \\
S_p \quad & \text{int } a, b, t; \\
& \text{void } p() \\
& \{ \text{if } (a == 0) \\
& \{ a = a-1; \} \\
& p(); \} \\
& t = a+b; \}
\end{align*}
\]
CS 618 Interprocedural DFA: IPDFA Using Value Contexts 77/98
Available Expressions Analysis Using Value Contexts

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

Create a new context \( X_0 \) with \( BI \) which is 0 for available expressions analysis.
Initialize \( exitValue(X_0) \) to \( T = 1 \)
Initialize the work list with all nodes in procedure main for \( X_0 \)
Initialize \( Out_n[X_0] \) for all \( n \) in main to \( T \)

Oct 2017 IIT Bombay

CS 618 Interprocedural DFA: IPDFA Using Value Contexts 77/98
Available Expressions Analysis Using Value Contexts

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

Create a new context \( X_1 \) with entry value 1
Record the transition to \( X_1 \)
Initialize \( exitValue(X_1) \) to \( T = 1 \)
Add all nodes of procedure \( p \) to the work list for \( X_1 \)
Initialize \( Out_n[X_1] \) for all \( n \) in \( p \) to \( T \)

Oct 2017 IIT Bombay
Available Expressions Analysis Using Value Contexts

WL = \[X_1|C_2, X_1|n_3, X_1|E_p, X_0|n_1, X_0|E_m]\n
<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>X_0 = \text{(main.0)}</td>
<td>1</td>
</tr>
<tr>
<td>X_1 = \text{(p.1)}</td>
<td>1</td>
</tr>
<tr>
<td>X_2 = \text{(p.0)}</td>
<td>1</td>
</tr>
</tbody>
</table>

Since there is no context for \(p\) with value 0, create context \(X_2\). Record the transition to \(X_2\). Initialize \(\text{exitValue}(X_2) = 1\). Add all nodes of procedure \(p\) to the work list for \(X_2\). Initialize \(\text{Out}_p[X_2]\) for all \(n\) in \(p\) to \(1\).

Available Expressions Analysis Using Value Contexts

WL = \[X_2|n_2, X_2|C_2, X_2|n_3, X_2|E_p, X_1|n_1, X_1|E_p, X_0|n_1, X_0|E_m]\n
<table>
<thead>
<tr>
<th>Context</th>
<th>exitValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>X_0 = \text{(main.0)}</td>
<td>1</td>
</tr>
<tr>
<td>X_1 = \text{(p.1)}</td>
<td>1</td>
</tr>
<tr>
<td>X_2 = \text{(p.0)}</td>
<td>1</td>
</tr>
</tbody>
</table>

Oct 2017

IIT Bombay
Available Expressions Analysis Using Value Contexts

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

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

**Available Expressions Analysis Using Value Contexts**

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

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

At \( E_p \) the values from \( S_p \) and \( n_3 \) are merged for context \( X_2 \), \( \text{exitValue}(X_2) \) is set to 0.

Since \( X_2 \) has transitions \( X_1 \rightarrow X_2 \) and \( X_2 \rightarrow X_3 \), \( \text{Out}_{C_2}[X_2] \) and \( \text{Out}_{C_3}[X_2] \) become 0.

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

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

Context | exitValue
--- | ---
\( X_0 = (\text{main.0}) \) | 1
\( X_1 = (p.1) \) | 1
\( X_2 = (p.0) \) | 0

Work list is empty and the analysis is over.
This example illustrates...

At the interprocedural level, we want to compute the glb across repeated calls at the same call site (perhaps in a loop).

Out

If some context

No change

Partially Available Expressions Analysis Using Value Contexts

This example illustrates non-termination of analysis if the exit value is not merged with the previous Out value.

We assume that main calls p with entry value 1.
Partially Available Expressions Analysis Using Value Contexts

Value 1 reaches \( q \) creating the context \( X_2 \).
Because of the kill in \( n_2 \), value 0 reaches \( p \).

Procedure \( q \) is analysed in the context \( X_4 \) (for entry value 0).
Value 0 reaches \( p \) at call site \( C_3 \).

Since we have context \( X_3 \) for \( p \) with entry value 0, we use its exitValue to compute the value in \( Out_{C_3} \) for context \( X_3 \).
Value 1 reaches the end of \( q \) for \( X_4 \) and is recorded as its exitValue.

Oct 2017
Because of the loop, the value of X₃ in Inₙ₂ changes to 1 which then reaches q at C₂.
We already have context X₂ for q that has entry value 1.

We remove the transition X₃ ⊆ X₄ and include the transition X₃ ⊆ X₂.
We use the exitValue of X₂ to compute Out_C₂ for X₃ which becomes 0.

We restore the transition X₃ ⊆ X₄ and use the exitValue of X₄ to compute Out_C₁ for X₃ which again becomes 1.
We are back to the same situation.
The overall data flow values \( \Gamma \) are sets of \( X.v \) where \( X \) is a context and \( v \in L \) is the underlying data flow value.

- We merge underlying data flow values only if the contexts are same
  
  \[
  \Gamma_1 \uplus \Gamma_2 = \{ X.w | X.u \in \Gamma_1 \land X.v \in \Gamma_2 \Rightarrow w = u \cap v, \ 
  X.u \in \Gamma_1 \land X.v \notin \Gamma_2 \Rightarrow w = u, \ 
  X.u \notin \Gamma_1 \land X.v \in \Gamma_2 \Rightarrow w = v \}
  \]

  Effectively, if a context does not exist in \( \Gamma \), its value is \( \top \in L \) in \( \Gamma \).

- Data flow variables for node \( n \) in procedure \( p \) are \( \text{In}(p, n) \) and \( \text{Out}(p, n) \)

- The flow function for node \( n \) in procedure \( p \) is \( f(p, n) \)

We assume the following auxiliary functions:

- Function \( \text{context}(p, v) \) returns the context of procedure \( p \) for entry value \( v \).
  
  If no such context exists, the function creates a new context and returns it.

- Function \( \text{exitValue}(X) \) returns the exit value of context \( X \).
  
  If context \( X \) does not exist, the function returns \( \top \in L \).

- Function \( \text{gpred} \) extends the predecessor relation \( \text{pred} \) (which is local to a procedure) to a global level across procedures.

  \[
  \text{gpred}(p, n) = \begin{cases} 
  \{ (q, m) | \text{call site } m \text{ in } q \text{ calls } p \} & \text{if } n \in S_p \\
  \{ (p, m) | m \in \text{pred}(n) \} & \text{otherwise}
  \end{cases}
  \]
Value Contexts and Interprocedurally Valid Paths

The role of value contexts in context sensitivity
- Value contexts preserve interprocedurally valid paths
- Value contexts consider only interprocedurally valid paths

We explain this with the help of an example by illustrating paths using a staircase diagram.

Value Contexts and Interprocedurally Valid Paths: Example

We explain the data flow value at the entry of \( C_2 \) by dividing the paths into the following two categories:

A. Paths in which the innermost recursion is along the call at \( C_2 \).
B. Paths in which the innermost recursion is along the call at \( C_3 \).

We draw the staircase diagrams of the example paths in the two categories.

Innermost Recursion Along the Call at \( C_2 \)

For this example, the innermost call determines the exitValue of contexts.
**Innermost Recursion Along the Call at C₃**

Again, the innermost call determines the exitValue of contexts. The final values at the entry of C₃ are 1 (union of 1 and 0).

```
(X₂.1) Sₚ → X₂ → Eₚ (X₂.1)
(X₁.1) Sₚ → C₃ → X₁ → n₂ → Eₚ (X₁.0)
(X₂.1) Sₚ → C₂ → X₂ → n₁ → Eₚ (X₂.1)
(X₁.1) Sₚ → C₃ → X₁ → n₂ → Eₚ (X₁.0)
(X₁.1) Sₚ → C₁ → X₁ → n₁ → Eₚ (X₁.0)
(X₀.1) Sₘ → C₁ → X₀ → Eₘ (X₀.0)
```

---

**Tutorial Problem #1 for Value Contexts**

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

```
(a ∗ b) Smain
c = a ∗ bn₁
Call pC₁
c = a ∗ bEₚ
(a ∗ b) Sp
Call pC₂
(a ∗ b) bn₂
Call p
```

---

**Tutorial Problem #2 for Value Contexts**

Perform interprocedural live variables analysis using value contexts.

```
main()
{
    while (...) {
        printf("%d\n", a);
        p();
    }
    p();
}
```

Observe the change in edges in the transition diagram.

---

**Tutorial Problem #3 for Value Contexts**

Perform interprocedural available expressions analysis using value contexts.

```
main()
{
    while (a > b) {
        c = a * b;
        p();
        a = a * b;
    }
}
```

Observe the change in edges in the transition diagram.
**Tutorial Problem #4 for Value Contexts**

Perform interprocedural available expressions analysis using value contexts

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

**Result of Tutorial #5**

```plaintext
main()
{  a = 5; b = 3;
   c = 7; d = 2;
   /*{a,d}*/
   p();
   /*{a,b,c,d}*/
   y = &z;
   /*{a,b,e}*/
   d = a*b;
   /*{d,e}*/
   q();
   /*{a,c,e}*/
   print a+c+e;
}
```

```plaintext
p()
{ /*{a,d,e}*/
  b = 2;
  if (b<d)
    /*{a,b,d,e}*/
    c = a+b;
  else
    /*{d,e}*/
    q();
    /*{a,b,c,d,e}*/
    a = a*b;
}
```

```plaintext
q()
{ /*{d,e}*/
  a = 1;
  /*{a,d,e}*/
  p(); /* C1 */
}
```

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

**Tutorial Problem #5 for Value Contexts**

Perform interprocedural live variables analysis using value contexts

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

**Tutorial Problem #6: Interprocedural Points-to Analysis**

Value contexts method requires three contexts as shown below in the transition diagram

```plaintext
main()
{  x = &y;
   z = &x;
   p(); /* C1 */
   y = &z;
   p(); /* C2 */
   x = *x;
   q(); /* C3 */
}
p()
{  if (...)
   {  p(); /* C2 */
      x = *x;
   }
}
```
### 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>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>hanoi</td>
<td>33</td>
<td>2</td>
<td>4</td>
<td>100000+</td>
<td>99922 3973 ×10²</td>
<td>8</td>
</tr>
<tr>
<td>bit_gray</td>
<td>53</td>
<td>5</td>
<td>11</td>
<td>100000+</td>
<td>31374 2705 ×10²</td>
<td>17</td>
</tr>
<tr>
<td>analyzer</td>
<td>288</td>
<td>14</td>
<td>20</td>
<td>21</td>
<td>4 20.33</td>
<td>21</td>
</tr>
<tr>
<td>distray</td>
<td>331</td>
<td>9</td>
<td>21</td>
<td>96</td>
<td>28 322.41</td>
<td>22</td>
</tr>
<tr>
<td>mason</td>
<td>350</td>
<td>9</td>
<td>13</td>
<td>100000+</td>
<td>22143 432 ×10²</td>
<td>14</td>
</tr>
<tr>
<td>fourinarow</td>
<td>676</td>
<td>17</td>
<td>45</td>
<td>510</td>
<td>158 397.76</td>
<td>46</td>
</tr>
<tr>
<td>sim</td>
<td>1146</td>
<td>13</td>
<td>45</td>
<td>8 100000+</td>
<td>33546 1427 ×10²</td>
<td>211</td>
</tr>
<tr>
<td>181.mcf</td>
<td>1299</td>
<td>17</td>
<td>24</td>
<td>6</td>
<td>32789 484 ×10²</td>
<td>41</td>
</tr>
<tr>
<td>256_bzip2</td>
<td>3320</td>
<td>63</td>
<td>198</td>
<td>7</td>
<td>402 63 258.33</td>
<td>406</td>
</tr>
</tbody>
</table>

- LoC is the number of lines of code,
- #F is the number of procedures,
- #C is the number of call sites,
- #CS is the number of call strings
- Max denotes the maximum number of call strings reaching any node.
- Analysis time is in milliseconds.

(Implementation was carried out by Seema Ravandale.)

### Some Observations

- Compromising on precision may not be necessary for efficiency.
- Separating the necessary information from redundant information is much more significant.
- Data flow propagation in real programs seems to involve only a small subset of all possible values.
- Much fewer changes than the theoretically possible worst case number of changes.
- A precise modelling of the process of analysis is often an eye opener.

\[
\text{# distinct tagged values = Min (# actual contexts, # actual data flow values)}
\]