Efficient Call Strings Method for Flow and Context Sensitive Interprocedural Data Flow Analysis

Uday Khedker

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

September 2012
Part 1

About These Slides
Copyright

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


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


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

Value Based Termination of Call String Construction
An Overview

- Value based termination of call string construction (VBTCC)
  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

Important Disclaimer

- These slides are aimed at
  - teaching rather than making a short technical presentation,
  - It is assumed that the people going through these slides do not have the benefit of attending the associated lectures
- Hence these slides are verbose with plenty of additional comments
  (usually not found in other slides)
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

**VBTCC: A Motivating Example**

- $S_p$
- $C_i$
- $R_i$
- $E_p$
- $S_q$
- $E_q$

Call q
WE WILL

• first work out the conventional call strings method on the example program,
• make useful observations about how it works, and
• convert it to VBTCC based call strings method
VBTCC: A Motivating Example

$\sigma_0/x_0 \quad \sigma_1/x_1 \quad \sigma_2/x_1 \quad \sigma_3/x_2 \quad \sigma_4/x_3$

$S_p \quad C_i \quad R_i \quad E_p \quad S_q \quad E_q$

Call q

Sep 2012 IIT Bombay
VBTC: A Motivating Example

- Call strings remain same
- Associated data flow values may change
- Since the data flow values of $\sigma_1$ and $\sigma_2$ were identical at $S_p$, they must remain identical at $C_i$ too
- In this case, the data flow value of $\sigma_3$ has become same as that of $\sigma_1$ and $\sigma_2$
VBTC: A Motivating Example

- Call strings have been updated by suffixing the call site
- Associated data flow values remain same

\[ \frac{\sigma_0}{x_0}, \frac{\sigma_1}{x_1}, \frac{\sigma_2}{x_2}, \frac{\sigma_3}{x_3}, \frac{\sigma_4}{x_4} \]

\[ \frac{\sigma_0 c_i}{x_0}, \frac{\sigma_1 c_i}{x_1}, \frac{\sigma_2 c_i}{x_1}, \frac{\sigma_3 c_i}{x_1}, \frac{\sigma_4 c_i}{x_3} \]
VBTC: A Motivating Example

- Call strings remain same
- Associated data flow values may change
- Since the data flow values of $\sigma_1$, $\sigma_2$, and $\sigma_3$ were identical at $S_q$, they must remain identical at $E_q$ too
VBTTCC: A Motivating Example

- Call strings have been updated by removing the last call site
- Associated data flow values remain same
VBTC: A Motivating Example
**VBTCC: A Motivating Example**

- Call strings remain same
- Associated data flow values could change
- All call strings that had identical values at $S_p$, have identical values at $E_p$ also
VBTCC: A Motivating Example

Call q

$S_p$ → $C_i$ → $R_i$ → $E_p$

$S_q$
Incorporating VBTCC in the Conventional Method

- Data flow value invariant: If $\sigma_1$ and $\sigma_2$ have equal values at $S_p$, then
Incorporating VBTCC in the Conventional Method

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

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

- We can reduce the amount of effort by
  - Partitioning the call strings at $S_p$ for each procedure $p$
  - Replacing all call strings in an equivalence class by its id
  - Regenerating call strings at $E_p$ by replacing equivalence class ids by the call strings in them
Incorporating VBTCC in the Conventional Method

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

- We can reduce the amount of effort by
  - Partitioning the call strings at $S_p$ for each procedure $p$
  - Replacing all call strings in an equivalence class by its id
  - Regenerating call strings at $E_p$ by replacing equivalence class ids by the call strings in them

- Can the partitions change?
Incorporating VBTCC in the Conventional Method

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

• We can reduce the amount of effort by
  ▶ Partitioning the call strings at $S_p$ for each procedure $p$
  ▶ Replacing all call strings in an equivalence class by its id
  ▶ Regenerating call strings at $E_p$ by replacing equivalence class ids by the call strings in them

• Can the partitions change?
  ▶ On a subsequent visit to $S_p$, the partition may be different
  ▶ The data flow values at $E_p$ would also change in a similar manner
  ▶ The data flow value invariant still holds
Understanding VBTCC: Motivating Example Revisited

\[ \sigma_0 \xrightarrow{x_0} \sigma_1 \xrightarrow{x_1} \sigma_2 \xrightarrow{x_1} \sigma_3 \xrightarrow{x_2} \sigma_4 \xrightarrow{x_3} \]

\[ S_p \rightarrow C_i \rightarrow R_i \rightarrow E_p \rightarrow S_q \]

Call q
Understanding VBTCC: Motivating Example Revisited

- Call strings are partitioned using data flow values
- A unique id is assigned to each equivalence class
Understanding VBTCC: Motivating Example Revisited

- Call strings are replaced by class ids
- Data flow values remain same
Understanding VBTCC: Motivating Example Revisited

- Different class ids may get the same data flow value but a given class id has a single data flow value.
- The data flow values could change but the class id names remain same.

\[
\begin{align*}
\sigma_0 & \xrightarrow{x_0} s_0 \\
\sigma_1 & \xrightarrow{x_1} s_1 \\
\sigma_2 & \xrightarrow{x_1} s_2 \\
\sigma_3 & \xrightarrow{x_2} s_3 \\
\sigma_4 & \xrightarrow{x_3} s_4
\end{align*}
\]
Understanding VBTTCC: Motivating Example Revisited

- Call strings reaching $S_q$ have the call site $c_i$ suffixed to class ids created in the caller $p$
- Data flow values remain same
Understanding VBTCC: Motivating Example Revisited

- Call strings are partitioned at $S_q$
- A unique id is assigned to each equivalence class
Understanding VBTCC: Motivating Example Revisited

- Call strings are replaced by class ids

\[ \sigma_0/x_0 \quad \sigma_1/x_1 \quad \sigma_2/x_2 \quad \sigma_3/x_3 \]

\[ \sigma_1/x_1 \quad \sigma_2/x_1 \quad \sigma_3/x_2 \quad \sigma_4/x_3 \]

\[ s_0 \quad s_1 \quad s_2 \quad s_3 \]

\[ s_0/c_i \quad s_1/c_i \quad s_2/c_i \quad s_3/c_i \]

\[ s_4 \quad s_5 \quad s_6 \]

\[ E_q \]

- Sep 2012 IIT Bombay
Understanding VBTCC: Motivating Example Revisited

The data flow values could change but the class id names remain same.
Understanding VBTCC: Motivating Example Revisited

- Call strings are regenerated by replacing the class ids by the call strings in the equivalence class
- Data flow values remain same
Understanding VBTCC: Motivating Example Revisited

- The last call site is removed from the call strings to recover the class ids of $p$
- Data flow values remain same
Understanding VBTCC: Motivating Example Revisited
Call strings are regenerated by replacing the class ids by the call strings in the equivalence class
Understanding VBTCC: Motivating Example Revisited

\[
\begin{align*}
S_p & \xrightarrow{\sigma_0} s_0, s_1, s_2, s_3 \\
S_q & \xrightarrow{\sigma_0} s_4, s_5, s_6 \\
C_i & \xrightarrow{\sigma_0} s_0, s_1, s_2, s_3 \\
R_i & \xrightarrow{\sigma_0} s_0, s_1, s_2, s_3 \\
E_p & \xrightarrow{\sigma_0} s_0, s_1, s_2, s_3 \\
\end{align*}
\]
The Role of Partitions of Call Strings

- An equivalence class $s_i$ for a procedure $p$ means that
The Role of Partitions of Call Strings

- An equivalence class \( s_i \) for a procedure \( p \) means that

  \[
  \text{All call strings in } s_i \text{ would have identical data flow values at } E_p
  \]
The Role of Partitions of Call Strings

- An equivalence class $s_i$ for a procedure $p$ means that
  
  *All call strings in $s_i$ would have identical data flow values at $E_p$*

  (If two call strings are clubbed together in an equivalence class, they remain clubbed together until $E_p$ is reached)
The Role of Partitions of Call Strings

- An equivalence class \( s_i \) for a procedure \( p \) means that
  
  "All call strings in \( s_i \) would have identical data flow values at \( E_p \)"

  (If two call strings are clubbed together in an equivalence class, they remain clubbed together until \( E_p \) is reached)

- We start creating equivalence classes at \( S_p \)
The Role of Partitions of Call Strings

- An equivalence class $s_i$ for a procedure $p$ means that
  
  All call strings in $s_i$ would have identical data flow values at $E_p$

  (If two call strings are clubbed together in an equivalence class, they remain clubbed together until $E_p$ is reached)

- We start creating equivalence classes at $S_p$

- Every visit to $S_p$ adjusts the equivalence classes
The Role of Partitions of Call Strings

- An equivalence class $s_i$ for a procedure $p$ means that
  
  *All call strings in $s_i$ would have identical data flow values at $E_p$*

  (If two call strings are clubbed together in an equivalence class, they remain clubbed together until $E_p$ is reached)

- We start creating equivalence classes at $S_p$

- Every visit to $S_p$ adjusts the equivalence classes
  
  ▶ If a new data flow value is discovered, a new equivalence class is created
  
  ▶ If a new call string is discovered, it will be included in an equivalence class based on its data flow value
The Role of Partitions of Call Strings

- An equivalence class $s_i$ for a procedure $p$ means that

  All call strings in $s_i$ would have identical data flow values at $E_p$

  (If two call strings are clubbed together in an equivalence class, they remain clubbed together until $E_p$ is reached)

- We start creating equivalence classes at $S_p$

- Every visit to $S_p$ adjusts the equivalence classes

  - If a new data flow value is discovered, a new equivalence class is created
  - If a new call string is discovered, it will be included in an equivalence class based on its data flow value

- It is possible to adjust equivalence classes at internal nodes too

  However, doing it at each statement may be inefficient
Additional Requirements for VBTCC

- Work list management as described later
- Correctness requirement:
  - Whenever representation is performed at $S_p$, $E_p$ must be added to the work list
  - In case $E_p$ is to be processed but its predecessors have not been put on the work list ($E_p$ may have been added due to representation), discard $E_p$ from the work list (has the effect of generating $\top$ value).
- Efficiency consideration: Process “inner calls” first
Work List Organization for Forward Analyses

- Maintain a stack of work lists for the procedures being analyzed (At most one entry per procedure on the stack)
- Order the nodes in each work list in reverse post order
- Remove the head of work list for the procedure on top (say $p$)
  - If the selected node is $S_p$
    - Adjust the call string partition based on the data flow values
    - Replace call strings by class ids
    - Insert $E_p$ in the list for $p$
  - If the selected node is $C_i$ calling procedure $q$ then
    - 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 and add its successor return nodes to appropriate work lists
    - Regenerate the call strings by replacing class ids by the call strings in the class
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
VBTPCC for Recursion

- We first make important observations about the role of the length of a call string in recursive contexts in the classical call strings method.
- Then we intuitively see how VBTPCC serves the same role without actually constructing redundant call strings.
- Finally we formally argue that the two methods are equivalent.
The Role of Call Strings Length in Recursion (1)

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

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

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

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

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

- We consider self recursion for simplicity; the principles are general and are also applicable to indirect recursion.
- Context sensitivity requires processing each recursive path independently.
- For a given recursive path
  - The *recursive call sequence* (RCS) refers to the subpath that builds recursion.
  - The *recursive return sequence* (RRS) refers to the subpath that unfolds recursion.
  - The *recursion terminating path* (RTP) refers to the subpath from RCS to RRS.

(We assume that a static analysis can assign arbitrary data flow values to the unreachable parts of the program in the absence of an RTP.)
The Role of Call Strings Length in Recursion (2)

- Data flow value reaching from outside

\[ \frac{\sigma}{x_0} \]

Diagram:
- \( S_p \)
- \( C_i \)
- \( R_i \)
- \( E_p \)
- Call \( p \)
The data flow value $\langle \sigma, x_0 \rangle$ is propagated over RCS.
The data flow value $\langle\sigma, x_0\rangle$ is propagated over RCS.
The data flow value \( \langle \sigma, x_0 \rangle \) is propagated over RCS.

We get the new context \( \sigma c_i \) and the new data flow value \( x_1 \).
The Role of Call Strings Length in Recursion (2)

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

The new pair \( \sigma \sigma C_i, x_1 \) is propagated over RCS.
The Role of Call Strings Length in Recursion (2)

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

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

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

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

- Even if the data flow values do not change any further, we still need to propagate them in RCS in order to build call strings that are sufficiently large.
- We need large call strings to allow for all changes in RRS (as will be clear soon).
The Role of Call Strings Length in Recursion (2)

- Even if the data flow values do not change any further, we still need to propagate them in RCS in order to build call strings that are sufficiently large.
- We need large call strings to allow for all changes in RRS (as will be clear soon).
The Role of Call Strings Length in Recursion (2)

- Even if the data flow values do not change any further, we still need to propagate them in RCS in order to build call strings that are sufficiently large.
- We need large call strings to allow for all changes in RRS (as will be clear soon).
The Role of Call Strings Length in Recursion (2)

- Assume that we construct all call strings as required by the conventional call strings method.

- Many of these call strings are redundant in that they do not correspond to any new data flow value.

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

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

\[
\begin{array}{c|c|c|c|c|c}
\sigma & \sigma C_i & \sigma C_i C_i & \sigma C_i C_i C_i & \sigma C_i C_i C_i C_i & \ldots \\
X_0 & X_1 & X_2 & X_2 & X_2 & \\
\end{array}
\]
• Now we traverse the recursion terminating path
The Role of Call Strings Length in Recursion (2)

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

- The call strings and data flow values reach the exit of $E_p$ unchanged.
Now we start processing RRS

The call strings ending with $c_i$ and their data flow values reach the entry of $R_i$ unchanged
Now we start processing RRS

The call strings ending with $c_i$ and their data flow values reach the entry of $R_i$ unchanged

The last occurrence of $c_i$ is removed and the call strings reach the entry of $E_p$ with new data flow values.
The Role of Call Strings Length in Recursion (2)

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

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

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

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

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

- We merge the data values at the entry of $E_p$ again and
- give new names to the resulting data flow values

\[
\begin{align*}
\frac{\sigma}{X_0} & \quad \frac{\sigma C_i}{X_1} & \quad \frac{\sigma C_i C_i}{X_2} & \quad \frac{\sigma C_i C_i C_i}{X_2} & \quad \frac{\sigma C_i C_i C_i C_i \ldots}{X_2} \\
\frac{\sigma}{Z_0} & \quad \frac{\sigma C_i}{Z_1} & \quad \frac{\sigma C_i C_i}{Z_2} & \quad \frac{\sigma C_i C_i C_i}{Z_2} \\
& \quad \frac{\sigma C_i C_i C_i C_i}{Z_2} & \quad \frac{\sigma C_i C_i C_i C_i}{Z_2} \\
\frac{\sigma}{Y_0} & \quad \frac{\sigma C_i}{Y_1} & \quad \frac{\sigma C_i C_i}{Y_2} & \quad \frac{\sigma C_i C_i C_i}{Y_2} & \quad \frac{\sigma C_i C_i C_i C_i}{Y_2} & \quad \frac{\sigma C_i C_i C_i C_i C_i \ldots}{Y_2}
\end{align*}
\]
The Role of Call Strings Length in Recursion (2)

- The call strings and their new data flow values reach the exit of $E_p$ unchanged.
We process RRS yet another time.

The call strings ending with \( c_i \) and their new data flow values reach the entry of \( R_i \) unchanged.
The Role of Call Strings Length in Recursion (2)

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

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

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

Since the last $c_i$ is removed every time $R_i$ is visited, we can visit it at most as many times as the number of $c_i$ in a call string.

We build call strings while processing RCS, and since at that time we do not know the number of times RRS may have to be processed, we build large call strings.
The Role of Call Strings Length in Recursion (2)

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

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

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

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

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

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

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

*The main reason behind building long call strings is to allow an adequate number of traversals over RCS.*
We have a single data flow value reaching $S_p$ from the outside
We have a single data flow value reaching $S_p$ from the outside
Thus we have a single equivalence class in the partition
We have a single data flow value reaching $S_p$ from the outside.

Thus we have a single equivalence class in the partition.

We replace the call string $\sigma$ by the class id $s_0$ and
We have a single data flow value reaching $S_p$ from the outside.

Thus we have a single equivalence class in the partition.

We replace the call string $\sigma$ by the class id $s_0$ and get the new call string $s_0c_i$ at the exit of $C_i$ with new data flow value $x_1$. 
The new call string and its data flow value reaches $S_p$. 

- **VBTCC in Recursion: Motivating Example Revisited**

- **$S_0$**
  - $\sigma \frac{x_0}{S_0}$
  - $S_0 \frac{c_i}{x_1}$

- **Call $p$**
  - $C_i$
  - $R_i$

- **$S_p$**
VBTCC in Recursion: Motivating Example Revisited

- The new call string and its data flow value reaches \( S_p \)
- We adjust the partitioning to include the new call string
- New partition names are introduced only for new data flow values; the association between a partition and a data flow value in a procedure must remain invariant during analysis
The new call string and its data flow value reaches $S_p$

We adjust the partitioning to include the new call string

New partition names are introduced only for new data flow values; the association between a partition and a data flow value in a procedure must remain invariant during analysis

We get $\langle s_1 c_i, x_2 \rangle$ at the exit of $C_i$
We continue to process RCS and
We continue to process RCS and \( \langle s_1 c_i, x_2 \rangle \) reaches \( S_p \)
VBTCC in Recursion: Motivating Example Revisited

- We continue to process RCS and $\langle s_1 c_i, x_2 \rangle$ reaches $S_p$
- We adjust the partition to create a new equivalence class for the new data flow value $x_2$
VBTCC in Recursion: Motivating Example Revisited

- We continue to process RCS and \( \langle s_1 c_i, x_2 \rangle \) reaches \( S_p \).
- We adjust the partition to create a new equivalence class for the new data flow value \( x_2 \).
- For our example, the data flow values cease to change after computing \( x_2 \), i.e. \( \langle s_2, x_2 \rangle \) at the exit of \( S_p \) reaches as \( \langle s_2 c_i, x_2 \rangle \) at the exit of \( C_i \).
VBTCC in Recursion: Motivating Example Revisited

- \( \langle s_2 c_i, x_2 \rangle \) reaches \( S_p \)
VBTCC in Recursion: Motivating Example Revisited

- $\langle s_2 c_i, x_2 \rangle$ reaches $S_p$
- We need to adjust the partitions again
• \(\langle s_2c_i, x_2 \rangle\) reaches \(S_p\)
• We need to adjust the partitions again
• Since we already have a partition for \(x_2\), \(s_2c_i\) gets included in it
• Since no new partition has been created (because no new data flow value is discovered), we stop processing RCS
VBTCC in Recursion: Motivating Example Revisited

- The call strings are replaced by partition ids and are propagated to $E_p$
- Their data flow values may change along the way
VBTCC in Recursion: Motivating Example Revisited

- At $E_p$, the class ids are replaced by call strings contained in equivalence classes regenerating the call strings with their new values.

- Observe the effect of de-partitioning: The data flow value of $s_2$ has been copied to $s_1c_i$ as well as $s_2c_i$.

This has the effect of pushing the data flow values into “deeper” recursion without constructing the call strings because all these call strings would have identical data flow values.
• We now begin processing RRS
• The call strings ending with $c_i$ reach the entry of $R_i$ without any change in their data flow values
We now begin processing RRS.

The call strings ending with $c_i$ reach the entry of $R_i$ without any change in their data flow values.

The last occurrence of $c_i$ is removed and the call strings reach the entry of $E_p$ with new values.
VBTCC in Recursion: Motivating Example Revisited

- The values reaching $E_p$ along the two paths are merged
VBTCC in Recursion: Motivating Example Revisited

- The values reaching $E_p$ along the two paths are merged.
- For simplicity, we give new names to the resulting data flow values.

Sep 2012 IIT Bombay
VBTCC in Recursion: Motivating Example Revisited

- At $E_p$, the class ids are replaced by call strings contained in equivalence classes.
- This regenerates all call strings with their new values.
VBTC in Recursion: Motivating Example Revisited

- We now process RRS for the second time.
- The call strings ending with $c_i$ reach the entry of $R_i$ without any change in their data flow values.
VBTCC in Recursion: Motivating Example Revisited

- We now process RRS for the second time.
- The call strings ending with $c_i$ reach the entry of $R_i$ without any change in their data flow values.
- The last occurrence of $c_i$ is removed and the call strings reach the entry of $E_p$ with new values.
VBTTCC in Recursion: Motivating Example Revisited

- The values reaching $E_p$ along the two paths are merged.

\[
\begin{align*}
\sigma &/ s_0 \\
S_0 &/ x_0 \\
S_1 &/ x_1 \\
S_2 &/ x_2 \\
R_i &/ y_1 \\
R_i &/ y_2 \\
E_p &/ y_0 \sqcap y_1 \\
E_p &/ y_0 \sqcap y_2 \\
E_p &/ y_1 \sqcap y_2 \\
E_p &/ y_1 \\
E_p &/ y_2 \\
E_p &/ y_2 \\
\end{align*}
\]
The values reaching $E_p$ along the two paths are merged.

For simplicity, we give new names to the resulting data flow values.
At $E_p$, the class ids are replaced by call strings contained in equivalence classes.

This regenerates all call strings with their new values.
We now process RRS for the third time.

The call strings ending with \( c_i \) reach the entry of \( R_i \) without any change in their data flow values.
VBTCC in Recursion: Motivating Example Revisited

- We now process RRS for the third time.
- The call strings ending with $c_i$ reach the entry of $R_i$ without any change in their data flow values.
- The last occurrence of $c_i$ is removed and the call strings reach the entry of $E_p$ with new values.
The values reaching $E_p$ along the two paths are merged.
• The values reaching $E_p$ along the two paths are merged
• For simplicity, we give new names to the resulting data flow values
VBTCC in Recursion: Motivating Example Revisited

- At $E_p$, the class ids are replaced by call strings contained in equivalence classes.
- This regenerates all call strings with their new values.
• We now process RRS for the fourth time
• The call strings ending with $c_i$ reach the entry of $R_i$ without any change in their data flow values
• We now process RRS for the fourth time

• The call strings ending with $c_i$ reach the entry of $R_i$ without any change in their data flow values

• The last occurrence of $c_i$ is removed and the call strings reach the entry of $E_p$ with new values

• Note that for our example, these are the final values
• The values reaching $E_p$ along the two paths are merged
• The values reaching $E_p$ along the two paths are merged.
• For simplicity, we give new names to the resulting data flow values.
• For our example, these are the final values.
VBTCC in Recursion: Motivating Example Revisited

- At $E_p$, the class ids are replaced by call strings contained in equivalence classes.
- This regenerates call strings with their final values at the exit of $E_p$.
- The method needs to traverse RRS once more to ensure that there is no further change.
• We now process RRS for the fifth time
• The call strings ending with $c_i$ reach the entry of $R_i$ without any change in their data flow values
We now process RRS for the fifth time.

The call strings ending with $c_i$ reach the entry of $R_i$ without any change in their data flow values.

When the call strings reach the entry of $E_p$, their values do not change.
• Observe that we have traversed RCS only three times whereas RRS was traversed five times

• We had terminated call string construction using data flow values
VBTCC in Recursion: Motivating Example Revisited

- Observe that we have traversed RCS only three times whereas RRS was traversed five times.
- We had terminated call string construction using data flow values.
- The process of regeneration copies the data flow values to not only call string representing recursion depth \( i \), but also to the call string representing recursion depth \( i + 1 \).
VBTCC in Recursion: Motivating Example Revisited

- Observe that we have traversed RCS only three times whereas RRS was traversed five times.
- We had terminated call string construction using data flow values.
- The process of regeneration copies the data flow values to not only call string representing recursion depth $i$, but also to the call string representing recursion depth $i + 1$.
- Since this happens repeatedly, we end up propagating values to call strings for all depths of recursion without having to construct them.
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 $\sigma_c$ and $\sigma_r$ be the call site sequences for RCS and RRS

- $\sigma_c \equiv c_j c_r c_k c_p c_i c_q$
  (flow function $f$)

- $\sigma_r \equiv r_q r_i r_p r_k r_r r_j$
  (flow function $g$)

Assume that we allow up to $m$ occurrences of $\sigma_c$
Staircase Diagram of Computation along Recursive Paths

Traversing RCS $m$ times

$x_1 = f(x_0)$

Data flow value at $C_q$
Staircase Diagram of Computation along Recursive Paths

Traversing RCS $m$ times

$x_2 = f^2(x_0)$

Data flow value at $C_q$
Staircase Diagram of Computation along Recursive Paths

Traversing RCS \(m\) times

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

Data flow value at \(C_q\)
Staircase Diagram of Computation along Recursive Paths

Traversing RCS \( m \) times

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

Data flow value at \( C_q \)
Staircase Diagram of Computation along Recursive Paths

Traversing RCS $m$ times

$$x_i = f^i(x_0)$$

Data flow value at $C_q$
Staircase Diagram of Computation along Recursive Paths

Traversing RCS $m$ times

$$x_i = f^i(x_0)$$

Data flow value at $C_q$
Staircase Diagram of Computation along Recursive Paths

Traversing RCS \( m \) times

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

Data flow value at \( C_q \)
Staircase Diagram of Computation along Recursive Paths

Traversing RCS \( m \) times

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

Data flow value at \( C_q \)

\[
z_m = h(x_m)
\]

Data flow value at \( R_q \)

Sep 2012
Staircase Diagram of Computation along Recursive Paths

**Traversing RCS m times**

\[
x_i = f^i(x_0) \quad \text{Data flow value at } C_q
\]

**Traversing RRS m times**

\[
z_{m-1} = h(x_{m-1}) \cap g(z_m) \quad \text{Data flow value at } R_q
\]
Staircase Diagram of Computation along Recursive Paths

Traversing RCS $m$ times

$\sigma_c \xrightarrow{f} x_m \xrightarrow{h} z_m \xrightarrow{g} z_{m-1} \xrightarrow{g} \sigma_r$

Traversing RRS $m$ times

$\sigma_c \xrightarrow{f} x_{m-1} \xrightarrow{h} z_{m-1} \xrightarrow{g} \sigma_r$

$x_i = f^i(x_0)$

Data flow value at $C_q$

$z_{m-2} = h(x_{m-2}) \cap g(z_{m-1})$

Data flow value at $R_q$
Staircase Diagram of Computation along Recursive Paths

Traversing RCS $m$ times

Traversing RRS $m$ times

$x_i = f^i(x_0)$

Data flow value at $C_q$

$z_{m-j} = h(x_{m-j}) \cap g(z_{m-j+1})$

Data flow value at $R_q$
Staircase Diagram of Computation along Recursive Paths

Traversing RCS \( m \) times

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

Data flow value at \( C_q \)

Traversing RRS \( m \) times

\[ z_{m-j} = h(x_{m-j}) \cap g(z_{m-j+1}) \]

Data flow value at \( R_q \)
Staircase Diagram of Computation along Recursive Paths

Traversing RCS \( m \) times

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

Data flow value at \( C_q \)

Traversing RRS \( m \) times

\[ z_{m-j} = h(x_{m-j}) \cap g(z_{m-j+1}) \]

Data flow value at \( R_q \)
Staircase Diagram of Computation along Recursive Paths

Traversing RCS \( m \) times

\[
\begin{align*}
\sigma_c & \xrightarrow{f} x_1 \\
& \xrightarrow{f} x_2 \\
& \xrightarrow{f} x_3 \\
& \vdots \\
& \xrightarrow{f} x_{m-3} \\
& \xrightarrow{f} x_{m-2} \\
& \xrightarrow{f} x_{m-1} \\
& \xrightarrow{f} x_m \\
& \xrightarrow{h} z_m \\
\end{align*}
\]

Traversing RRS \( m \) times

\[
\begin{align*}
\sigma_r & \xleftarrow{g} z_{m-1} \\
& \xleftarrow{g} z_{m-2} \\
& \xleftarrow{g} z_{m-3} \\
& \vdots \\
& \xleftarrow{g} z_2 \\
& \xleftarrow{g} z_1 \\
\end{align*}
\]

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

Data flow value at \( C_q \)

\[z_{m-j} = h(x_{m-j}) \cap g(z_{m-j+1})\]

Data flow value at \( R_q \)

Sep 2012 IIT Bombay
Staircase Diagram of Computation along Recursive Paths

Traversing RCS $m$ times

$\sigma_c \xrightarrow{f} x_m \xrightarrow{h} z_m \xrightarrow{\sigma_r}$

Traversing RRS $m$ times

$\sigma_c \xrightarrow{f} x_{m-1} \xrightarrow{h} z_{m-1} \xrightarrow{\sigma_r}$

$\sigma_c \xrightarrow{f} x_{m-2} \xrightarrow{h} z_{m-2} \xrightarrow{\sigma_r}$

$\sigma_c \xrightarrow{f} x_{m-3} \xrightarrow{h} z_{m-3} \xrightarrow{\sigma_r}$

$\sigma_c \xrightarrow{f} x_2 \xrightarrow{h} z_2 \xrightarrow{\sigma_r}$

$\sigma_c \xrightarrow{f} x_1 \xrightarrow{h} z_1 \xrightarrow{\sigma_r}$

$\sigma_c \xrightarrow{f} x_0 \xrightarrow{h} z_0 \xrightarrow{\sigma_r}$

$x_i = f^i(x_0)$

Data flow value at $C_q$

$z_{m-j} = h(x_{m-j}) \cap g(z_{m-j+1})$

Data flow value at $R_q$
Fixed Bound Closure Bound of Flow Function

- $n > 0$ is the fixed point closure bound of $h : L \mapsto L$ if it is the smallest number such that

$$\forall x \in L, \ h^{n+1}(x) = h^n(x)$$
Computation of Data Flow Values along Recursive Paths

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

FP closure bound of $f$

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

FP closure bound of $f$

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

FP closure bound of $f$

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

FP closure bound of $f$

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

FP closure bound of $f$

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

FP closure bound of $f$

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

FP closure bound of $f$

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

\[ z_m = h(x_\omega) \]
Computation of Data Flow Values along Recursive Paths

FP closure bound of $f$

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

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

FP closure bound of $f$

FP closure bound of $g$

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

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

FP closure bound of $f$

FP closure bound of $g$

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

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

Identical data flow values

FP closure bound of $f$

FP closure bound of $g$

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

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

FP closure bound of \( f \)

FP closure bound of \( g \)

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

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

FP closure bound of \( f \)

FP closure bound of \( g \)

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

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

**FP closure bound of \( f \)**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

### FP closure bound of $f$

$$
\begin{align*}
  x_i &= \begin{cases} 
    f^i(x_0) & i < \omega \\
    f^\omega(x_0) & \text{otherwise}
  \end{cases} \\
  z_{m-j} &= \begin{cases} 
    h(x_\omega) \sqcap g(z_{m-j+1}) & 0 \leq j \leq \eta \\
    h(x_\omega) \sqcap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\
    h(x_j) \sqcap g(z_{m-j+1}) & \text{otherwise}
  \end{cases}
\end{align*}
$$
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

\[ x_i = \begin{cases} 
  f^i(x_0) & i < \omega \\
  f^\omega(x_0) & \text{otherwise}
\end{cases} \quad \text{and} \quad z_{m-j} = \begin{cases} 
  h(x_\omega) \sqcap g(z_{m-j+1}) & 0 \leq j \leq \eta \\
  h(x_\omega) \sqcap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\
  h(x_j) \sqcap g(z_{m-j+1}) & \text{otherwise}
\end{cases} \]
Bounding the Call String Length Using Data Flow Values

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

- This amounts to simulating all call strings that would have otherwise been constructed while traversing RCS.
- It can also be seen as virtually climbing up the steps in RRS as much as needed and then climbing down.
- This is possible only because
  - all these call strings would have the same data flow value associated with them, and
  - the data flow value computation begins from the last call strings

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

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

- This amounts to simulating all call strings that would have otherwise been constructed while traversing RCS.
- It can also be seen as virtually climbing up the steps in RRS as much as needed and then climbing down.
- This is possible only because
  - all these call strings would have the same data flow value associated with them, and
  - the data flow value computation begins from the last call strings

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

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

- This amounts to simulating all call strings that would have otherwise been constructed while traversing RCS.
- It can also be seen as virtually climbing up the steps in RRS as much as needed and then climbing down.
- This is possible only because
  - all these call strings would have the same data flow value associated with them, and
  - the data flow value computation begins from the last call strings

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

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

- This amounts to simulating all call strings that would have otherwise been constructed while traversing RCS.
- It can also be seen as virtually climbing up the steps in RRS as much as needed and then climbing down.
- This is possible only because
  - all these call strings would have the same data flow value associated with them, and
  - the data flow value computation begins from the last call strings

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

\[
z_{m-j} = \begin{cases} h(x_0) \cap g(z_{m-j+1}) & 0 \leq j \leq \eta \\ h(x_0) \cap g(z_{m-\eta}) & \eta < j \leq (m-\omega) \\ h(x_j) \cap g(z_{m-j+1}) & \text{otherwise} \end{cases}
\]
Bounding the Call String Length Using Data Flow Values

- This amounts to simulating all call strings that would have otherwise been constructed while traversing RCS.

- It can also be seen as virtually climbing up the steps in RRS as much as needed and then climbing down.

- This is possible only because
  - all these call strings would have the same data flow value associated with them, and
  - the data flow value computation begins from the last call strings.

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

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

• Consider a call string $\sigma = \ldots (c_i)_1 \ldots (c_i)_2 \ldots (c_i)_3 \ldots (c_i)_j \ldots$
  where $(c_i)_j$ denotes the $j^{th}$ occurrence of $c_i$
  Let $j \geq |L| + 1$
  Let $C_i$ call procedure $p$
Worst Case Length Bound

- Consider a call string $\sigma = \ldots (c_i)_1 \ldots (c_i)_2 \ldots (c_i)_3 \ldots (c_i)_j \ldots$
  where $(c_i)_j$ denotes the $j^{th}$ occurrence of $c_i$
  Let $j \geq |L| + 1$
  Let $C_i$ call procedure $p$
- All call string ending with $C_i$ reach entry $S_p$
Worst Case Length Bound

- Consider a call string \( \sigma = \ldots (c_i)_1 \ldots (c_i)_2 \ldots (c_i)_3 \ldots (c_i)_j \ldots \)
  where \((c_i)_j\) denotes the \(j^{th}\) occurrence of \(c_i\)
- Let \(j \geq |L| + 1\)
- Let \(C_i\) call procedure \(p\)
- All call string ending with \(C_i\) reach entry \(S_p\)
- Since only \(|L|\) distinct values are possible, by the pigeon hole principle, at least two prefixes ending with \(C_i\) will carry the same data flow value to \(S_p\).
Worst Case Length Bound

- Consider a call string \( \sigma = \ldots (c_i)_1 \ldots (c_i)_2 \ldots (c_i)_3 \ldots (c_i)_j \ldots \)
  where \( (c_i)_j \) denotes the \( j^{th} \) occurrence of \( c_i \)
  Let \( j \geq |L| + 1 \)
  Let \( C_i \) call procedure \( p \)
- All call string ending with \( C_i \) reach entry \( S_p \)
- Since only \( |L| \) distinct values are possible, by the pigeon hole principle, at least two prefixes ending with \( C_i \) will carry the same data flow value to \( S_p \).
  - All longer call strings will belong to the same partition
Worst Case Length Bound

- Consider a call string $\sigma = \ldots (c_i)_1 \ldots (c_i)_2 \ldots (c_i)_3 \ldots (c_i)_j \ldots$
  where $(c_i)_j$ denotes the $j^{th}$ occurrence of $c_i$
  Let $j \geq |L| + 1$
  Let $C_i$ call procedure $p$

- All call string ending with $C_i$ reach entry $S_p$

- Since only $|L|$ distinct values are possible, by the pigeon hole principle, at least two prefixes ending with $C_i$ will carry the same data flow value to $S_p$.
  - All longer call strings will belong to the same partition
  - Since one more $C_i$ may be suffixed to discover fixed point, $j \leq |L| + 1$
Worst Case Length Bound

- Consider a call string \( \sigma = \ldots (c_i)_1 \ldots (c_i)_2 \ldots (c_i)_3 \ldots (c_i)_j \ldots \)
  where \((c_i)_j\) denotes the \(j^{th}\) occurrence of \(c_i\)
  
  Let \(j \geq |L| + 1\)

- Let \(C_i\) call procedure \(p\)

- All call string ending with \(C_i\) reach entry \(S_p\)

- Since only \(|L|\) distinct values are possible, by the pigeon hole principle, at least two prefixes ending with \(C_i\) will carry the same data flow value to \(S_p\).
  
  \[\blacktriangleright \text{All longer call strings will belong to the same partition}\]
  \[\blacktriangleright \text{Since one more } C_i \text{ may be suffixed to discover fixed point, } j \leq |L| + 1\]

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

- Consider a call string $\sigma = \ldots (c_i)_1 \ldots (c_i)_2 \ldots (c_i)_3 \ldots (c_i)_j \ldots$
  where $(c_i)_j$ denotes the $j^{th}$ occurrence of $c_i$
  Let $j \geq |L| + 1$
  Let $C_i$ call procedure $p$

- All call string ending with $C_i$ reach entry $S_p$

- Since only $|L|$ distinct values are possible, by the pigeon hole principle, at least two prefixes ending with $C_i$ will carry the same data flow value to $S_p$.
  - All longer call strings will belong to the same partition
  - 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 $= K \times (|L| + 1)^2$
Approximate Version

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

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

- Use a demand driven approach:
  - After a dynamically definable limit (say a number $j$),
  - Start merging the data flow values and create a single partition for the merged value (even if it keeps changing)
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 data flow values and create a single partition for the merged value (even if it keeps changing)
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 data flow values and create a single partition for the merged value (even if it keeps changing)

- Context sensitive for a depth $j$ of recursion.
  Context insensitive beyond that.
Approximate Version

- For framework with infinite lattices, a fixed point for cyclic call sequence may not exist.
- Use a demand driven approach:
  - After a dynamically definable limit (say a number $j$),
  - Start merging the data flow values and create a single partition for the merged value (even if it keeps changing)
- Context sensitive for a depth $j$ of recursion.
  Context insensitive beyond that.
Approximate Version

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

- Use a demand driven approach:
  - After a dynamically definable limit (say a number $j$),
  - Start merging the data flow values and create a single partition for the merged value (even if it keeps changing)

- Context sensitive for a depth $j$ of recursion.
  Context insensitive beyond that.

- Assumption: Height of the lattice is finite.
# Reaching Definitions Analysis in GCC 4.0

<table>
<thead>
<tr>
<th>Program</th>
<th>LoC</th>
<th>#F</th>
<th>#C</th>
<th>3K length bound</th>
<th>Proposed Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>63</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>258.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>406</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>34</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>200.19</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.
- **Time** is in milliseconds.

(Implementation was carried out by Seema Ravandale.)
Some Observations

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

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

\[ \# \text{ distinct tagged values} = \min (\# \text{ actual contexts}, \# \text{ actual data flow values}) \]
Tutorial Problem on Interprocedural Points-to Analysis

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

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

• Number of distinct call sites in a call chain $K = 2$.
• Number of variables: 3
• Number of distinct points-to pairs: $3 \times 3 = 9$
• $L$ is powerset of all points-to pairs
• $|L| = 2^9$
• Length of the longest call string in Sharir-Pnueli method:
  \[ 2 \times (|L| + 1)^2 = 2^{19} + 2^{10} + 1 = 5, 25, 313 \]
• All call strings upto this length must be constructed by the Sharir-Pnueli method!
main()
{
    x = &y;
    z = &x;
    y = &z;
    p(); /* C1 */
}

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