
Value-Based Approach to Interprocedural Data Flow Analysis

In this chapter, we present the other paradigm of context and flow sensitive whole program analysis. This approach does not involve precomputation of summary flow functions. Instead, it directly computes the data flow information and propagates the inherited data flow information from callers to callees and the synthesized data flow information from callees to callers.

We first present the program model and some basic concepts underlying this approach. Then we present a method for precise flow and context sensitive interprocedural data flow analysis for bit vector frameworks. The subsequent section generalizes this method to general frameworks.

9.1 Program Model for Value-Based Approaches to Interprocedural Data Flow Analysis

A value-based approach of interprocedural data flow analysis views a program as a single large procedure with different kinds of paths rather than as a collection of independent procedures. With this view of programs, interprocedural data flow analysis reduces to identifying the origins of *ifps* and traversing them; the only difference is that these *ifps* are interprocedural rather than intraprocedural and hence must be sensitive to the calling contexts. This is required to distinguish between inherited data flow information propagated from different callers. This enables propagation of synthesized information to appropriate callers.

A value-based approach uses a supergraph which has been explained in Section 7.2. Let a given call site c_i in procedure r call procedure s . Then, logically the program points $Entry(C_i)$ and $Exit(R_i)$ belong to the caller procedure r in that the data flow information associated with these program points holds for procedure r . The program points $Exit(C_i)$ and $Entry(R_i)$ belong to the callee procedure s as the data flow information associated with these points holds for procedure s .

The roles of call and return nodes in a supergraph cannot be abstracted out into a single kind of node; they must be explicated for value-based interprocedural data flow analysis. Hence unlike intraprocedural data flow analysis, a general formulation that is uniformly applicable to both forward and backward data flow frameworks does

not seem natural; the flow function of the proposed abstract node representing call and return nodes will have to be predicated on whether the formulation is being used for forward flows or backward flows. Hence we restrict our formulations to forward data flow problems for simplicity of exposition,

As observed in [Chapter 7](#), traversing all paths in a supergraph results in context insensitive analysis. Context sensitivity requires that the propagation of data flow information must be restricted to interprocedurally valid paths.

DEFINITION 9.1 *A path from $\text{Start}_{\text{main}}$ to a block n in a supergraph is an interprocedurally valid path if*

1. *for every edge $\text{End}_r \rightarrow R_i$ in the path, there is a matching edge $C_i \rightarrow \text{Start}_r$ in the path, and*
2. *if the subpath from C_i to R_i does not contain any other call or return node, then after replacing this subpath by a single (fictitious) edge, the reduced path is interprocedurally valid.*

At the base level, a path consisting of only intraprocedural edges is a valid interprocedural path. Similarly, a path in which there is no return edge is also a valid interprocedural path; the validity constraint arises only when a return edge is encountered. This is because a return edge that appears in a path must correspond to the last call edge in the path. This constraint facilitates ensuring that the data flow information from a callee procedure is propagated back to the correct caller procedure.

Let call site c_i call procedure r . In an interprocedurally valid path, this procedure call is represented by a path segment starting with the call edge $C_i \rightarrow \text{Start}_r$ and ending in the corresponding return edge $\text{End}_r \rightarrow R_i$. Every such call appearing in an interprocedurally valid path can be abstracted out by a basic block making the call; a path containing this basic block remains an interprocedurally valid path.

We view call and return nodes as being *significant* nodes because they define the structure of an interprocedural path. Often we will restrict a path to the significant nodes appearing in it. For interprocedural validity, the structure of a path in terms of significant nodes should be derivable from the following context free grammar with *IPVP* as its start symbol:

$$\begin{aligned}
 \text{IPVP} &\rightarrow \text{finishedCalls } \text{unFinishedCalls} \\
 \text{finishedCalls} &\rightarrow C_i \text{ finishedCalls } R_i \\
 &\quad | \text{ finishedCalls } \text{finishedCalls} \\
 &\quad | \epsilon \\
 \text{unFinishedCalls} &\rightarrow C_i \text{ finishedCalls } \text{unFinishedCalls} \\
 &\quad | \epsilon
 \end{aligned}$$

where C_i and R_i are placeholders for terminal symbols representing corresponding call and return nodes in a supergraph.

DEFINITION 9.2 *An ifp ρ from a program point u to a program point v is an interprocedurally valid ifp if it is a suffix of some interprocedurally valid path.*

An important requirement of traversing interprocedurally valid ifps is discovering matching C_i for every R_i encountered in a path in the supergraph in order to establish interprocedural validity of the ifp. Value-based interprocedural analyses achieve this by embedding the information about contexts within the data flow values being computed. This information represents the call nodes C_i encountered in the paths traversed for computing the data flow value. In the presence of recursion, precise embedding of context information becomes an important issue in value-based interprocedural data flow analysis. The methods presented in this chapter handle recursive program without compromising on precision.

DEFINITION 9.3 *A calling context of procedure r is defined as a sequence of callers of r starting from the main procedure.*

A calling context σ is denoted by a string $c_1 \cdots c_k$ of call site names. This string represents a call sequence r_1, \dots, r_k starting from the main procedure, such that $c_i \in \text{CallsIn}_{r_i}$ and $c_i \in \text{CallsTo}_{r_{i+1}}$. Note that the call sites in a call string or the called procedures in a call chain need not be distinct.

Value-based interprocedural data flow analysis is defined in terms of data flow values that are pairs of the form $\langle \sigma, x \rangle$ where σ represents the context and $x \in L$ is the actual data flow value. We call a pair $\langle \sigma, x \rangle$ a *qualified data flow value* and denote it by X to distinguish it from x . In some cases, X may be a set of pairs $\langle \sigma, \alpha \rangle$ where $\alpha \in \Sigma$ is an entity. Where the context of usage is sufficient to distinguish between the two, we drop the adjective “qualified” and refer to both X and x as data flow values.

DEFINITION 9.4 *A path ρ in a supergraph is:*

- *An intraprocedural segment if ρ contains intraprocedural nodes only.*
- *A call segment if ρ contains intraprocedural nodes and at least one call node but no return node.*
- *A return segment if ρ contains intraprocedural nodes and at least one return node but no call node.*
- *A symmetric segment if ρ is an interprocedurally valid path from a call node C_i in procedure r to a return node R_j also in procedure r .*

An intraprocedural segment does not alter the context in a qualified data flow value whereas call and return segments do. A symmetric segment represents a sequence of finished calls—it alters the context within the segment but restores it at the end of the segment.

Apart from handling context, an interprocedural analysis must also handle scope rules and parameter passing mechanisms. These issues are handled as explained in Section 8.2. For simplicity, we assume that our programs have only global variables.

9.2 Interprocedural Analysis Using Restricted Contexts

Bit vector frameworks have special properties that make it possible to perform interprocedural analysis by remembering a restricted amount of context. The two key insights that this algorithm uses are:

- For bit vector frameworks, the default value of an entity α at a program point u , denoted \widehat{x}_u^α , can be considered as $\widehat{\top}$. If it becomes $\widehat{\perp}$, then it is sufficient to make $\widehat{x}_v^\alpha = \widehat{\perp}$ for all $v \in \text{neighbours}(u)$ such that $\widehat{f}_\alpha^{u \rightarrow v}$ is $\widehat{\phi}_{id}$. This effect needs to be propagated transitively.
- This propagation can be done independently of any other *ifp*. Thus there is no need to consider any other *ifp* of α or any *ifp* of some other entity β .

This allows fully context sensitive analysis by restricting the length of calling context σ to 1 at each call point in a sequence of calls. Reconstructing the calling contexts transitively along a call chain does not introduce any imprecision—it is possible to propagate different synthesized data flow information from a procedure to different callers of the procedure.

Let the qualified data flow value X_u at a program point u in procedure r be a set of tuples $\langle \psi, \alpha \rangle$ where α is the entity whose data flow value at u is $\widehat{\perp}$ and ψ is the context information which is either a call site $c_i \in \text{CallsTo}_r$ or “*”. When ψ is c_i , the data flow value $\widehat{x}_u^\alpha = \widehat{\perp}$ is inherited by r from the call at c_i . When ψ is *, the data flow value $\widehat{x}_u^\alpha = \widehat{\perp}$ is synthesized in r or in some procedure called from within r . The main difference between the two is that a data flow value qualified by c_i can only be propagated to the caller containing the call site c_i whereas the data flow value qualified by * should be propagated to all callers of r .

The exact criteria of propagation of $\widehat{\perp}$ values in a supergraph for a forward data flow framework is as described below. For backward data flows, the roles of C_i and R_i should be interchanged.

- When a pair $\langle \psi, \alpha \rangle$ reaches an intraprocedural node n ,
 - If $\widehat{f}_n^\alpha = \widehat{\phi}_\top$, $\langle \psi, \alpha \rangle$ should not be propagated any further.
 - If $\widehat{f}_n^\alpha = \widehat{\phi}_\perp$, it indicates generation of synthesized data flow information. Hence the pair $\langle \psi, \alpha \rangle$ must be replaced by the pair $\langle *, \alpha \rangle$.
 - If $\widehat{f}_n^\alpha = \widehat{\phi}_{id}$, the pair $\langle \psi, \alpha \rangle$ must be propagated further.

- When a pair $\langle \psi, \alpha \rangle$ reaches a call node C_i in procedure r , the $\widehat{\perp}$ value of α must be propagated to the called procedure with c_i as the calling context. Thus the pair $\langle c_i, \alpha \rangle$ must be propagated further.
- When a pair $\langle \psi, \alpha \rangle$ reaches R_i in procedure r ,
 - If ψ is $*$, the pair $\langle *, \alpha \rangle$ must be propagated further in r .
 - If ψ is c_i , the value $\widehat{\perp}$ of α has been inherited by r through the call site C_i so the $\widehat{\perp}$ value of α must be propagated further in the rest of r . However, the context from where its $\widehat{\perp}$ value reached C_i must be recovered. This is easily done by examining the pairs reaching C_i —if a pair $\langle \psi', \alpha \rangle$ reached C_i , then the required context is ψ' . Observe that ψ' could be another call site or could be $*$.
 - If ψ is some c_j other than c_i , it indicates traversal of an interprocedurally invalid path and the data flow value must be discarded. This is because the context information c_j represents the fact that C_j was the last call node traversed in the path so this qualified data flow value cannot reach any other return node; it must reach R_j where the calling context will be reconstructed.

We use IN_n and OUT_n to compute the qualified data flow values X ; the conventional variables In_n and Out_n continue to contain the underlying data flow values $x \in L$. The data flow equations are:

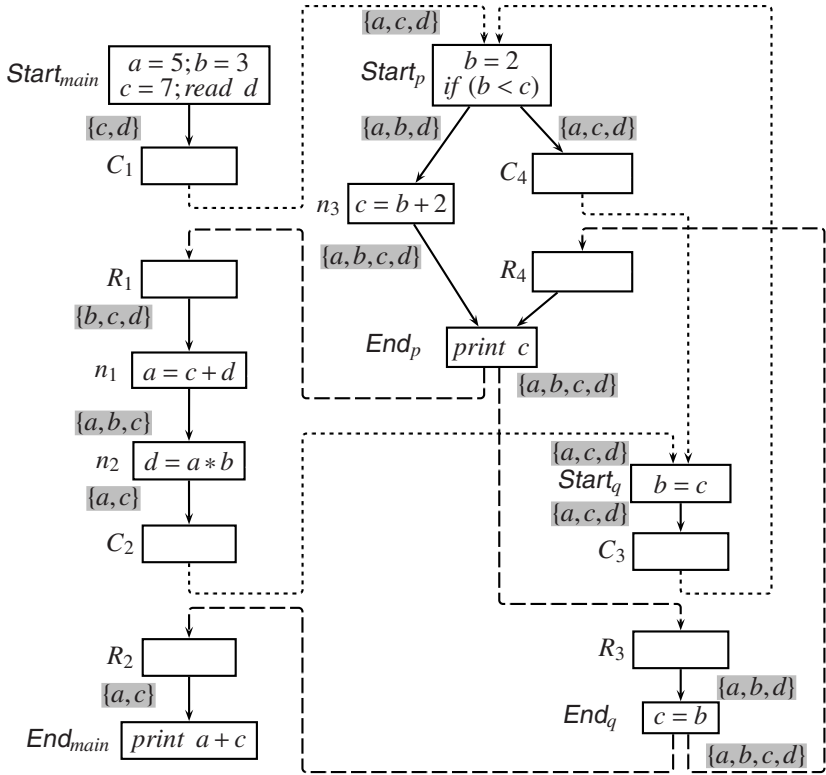
$$IN_n = \begin{cases} \{ \langle *, \alpha \rangle \mid \alpha \text{ is } \widehat{\perp} \text{ in } BI \} & n \text{ is } \text{Start}_{main} \\ \bigcup_{p \in \text{pred}(n)} OUT_p & \text{otherwise} \end{cases} \quad (9.1)$$

$$OUT_n = \text{ConstGEN}_n \cup \text{DepGEN}_n(IN_n) - (X - (\text{ConstKILL}_n - \text{DepKILL}_n(IN_n))) \quad (9.2)$$

where the constant and dependent components are defined as follows:

$$\begin{aligned} \text{ConstGEN}_n &= \begin{cases} \emptyset & n \text{ is } C_i \text{ or } R_i \\ \{ \langle *, \alpha \rangle \mid \widehat{f}_n^\alpha = \widehat{\phi}_\perp \} & \text{otherwise} \end{cases} \\ \text{DepGEN}_n(X) &= \begin{cases} \{ \langle c_i, \alpha \rangle \mid \langle \psi, \alpha \rangle \in X \} & n \text{ is } C_i \\ \{ \langle *, \alpha \rangle \mid \langle *, \alpha \rangle \in X \} \cup \{ \langle \psi, \alpha \rangle \mid \langle c_i, \alpha \rangle \in X, \langle \psi, \alpha \rangle \in IN_{C_i} \} & n \text{ is } R_i \\ \emptyset & \text{otherwise} \end{cases} \\ \text{ConstKILL}_n &= \{ \langle \psi, \alpha \rangle \mid \alpha \in \text{Gen}_n \text{ or } \alpha \in \text{Kill}_n \} \\ \text{DepKILL}_n(X) &= \emptyset \end{aligned}$$

Observe the use of the component function \widehat{f}_n^α for ConstGEN_n . For data flow frameworks that use \cup as \sqcap , \widehat{f}_n^α is $\widehat{\phi}_\perp$ if $\alpha \in \text{Gen}_n$ whereas for data flow frameworks that

**FIGURE 9.1**

An example program for interprocedural live variables analysis.

use \cap as \sqcap , it is $\widehat{\phi}_{\perp}$ if $\alpha \in Kill_n$ and $\alpha \notin Gen_n$. Also observe the use of IN_{C_i} in the definition of $DepGEN_n(X)$.

The final set of entities whose data flow values are $\widehat{\perp}$ are extracted from

$$In_n = \left\{ \alpha \mid \langle \psi, \alpha \rangle \in IN_n, \psi \in CallsTo_r \cup \{*\} \right\} \quad (9.3)$$

$$Out_n = \left\{ \alpha \mid \langle \psi, \alpha \rangle \in OUT_n, \psi \in CallsTo_r \cup \{*\} \right\} \quad (9.4)$$

Example 9.1

Consider the program in Figure 9.1 for interprocedural liveness analysis. All variables are global variables. Variables that are live at a program point are shown in graph boxes. Observe that variable d is live in procedure q at $Start_q$ but not before its call in the *main* procedure. The use of d in block n_1 makes it live in procedure p . Since q is called from p and neither p nor q modify d ,

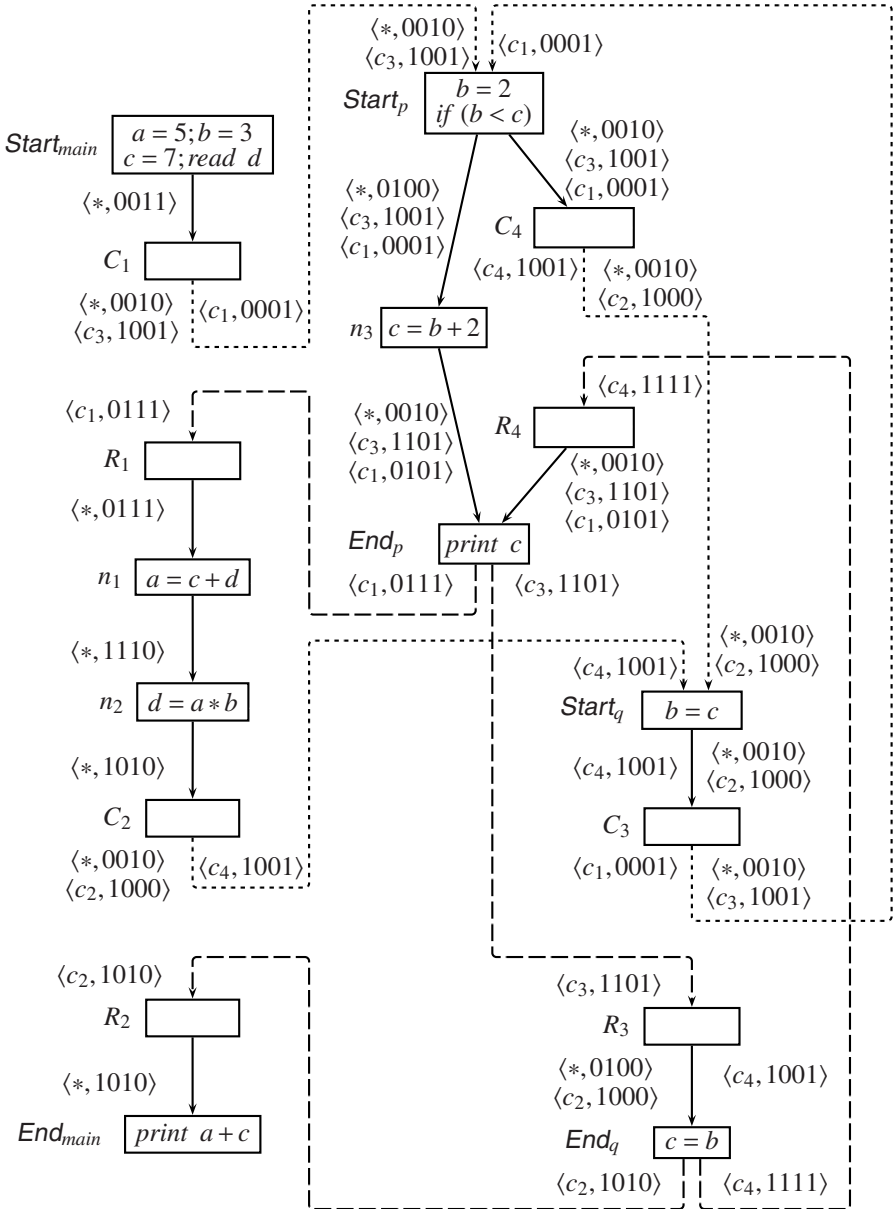
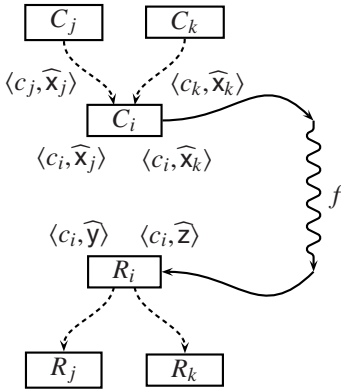


FIGURE 9.2

Result of interprocedural liveness analysis for the program in Figure 9.1 on the facing page.



- At return node R_i , we wish to reconstruct the values $\langle c_j, \widehat{f}(\widehat{x}_j) \rangle$ and $\langle c_k, \widehat{f}(\widehat{x}_k) \rangle$.
- If we merge the values of a at C_i and propagate $\langle c_i, \widehat{x}_j \sqcap \widehat{x}_k \rangle$, we will not get the values $\widehat{f}(\widehat{x}_j)$ and $\widehat{f}(\widehat{x}_k)$.
- If we do not merge the values but propagate them separately, how can we know if \widehat{y} should be propagated to R_j or R_k ? (Similarly for \widehat{z})?

FIGURE 9.3

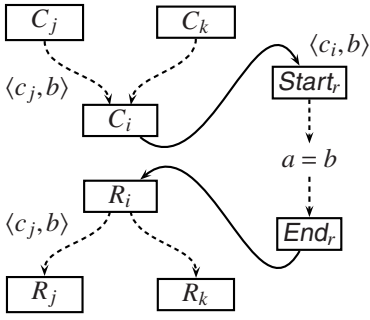
Difficulty in handling propagation of multiple values for the same entity.

it remains live at all program points in both the procedures. Similarly, a is live in procedure p because of the use of a in End_{main} but it is not live before the call to p in procedure main .

The result of interprocedural liveness analysis using this method has been shown in Figure 9.2 on the preceding page. Since this is a backward data flow problem, reconstruction of contexts happens at the call nodes rather than return nodes. Observe that at C_1 , $\langle c_3, 1001 \rangle$ contained in OUT_{C_1} is ignored, $\langle *, 0010 \rangle$ is allowed to pass through, and the context of $\langle c_1, 0001 \rangle$ is reconstructed to $*$ by examining OUT_{R_1} . At R_3 , the data flow values $\langle *, 0100 \rangle$, $\langle c_2, 1000 \rangle$, and $\langle c_4, 1001 \rangle$ are also propagated with the new context c_3 . Similar actions are taken at other call and return nodes. Blocks n_3 , End_p and End_q exhibit generation of synthesized data flow information. This causes a transfer of context from a call site c_i to $*$ for the entities that are contained in Gen set of these blocks; the component functions for these entities compute \perp for these entities. \square

To see why this method chooses to propagate a single value, consider Figure 9.3. Assume that instead of propagating a single value, we wish to propagate two different values \widehat{x}_i and \widehat{x}_k . In general, these values could be incomparable. For context sensitive analysis, we wish to get the values $\widehat{f}(\widehat{x}_i)$ at R_j and $\widehat{f}(\widehat{x}_k)$ at R_k . If we merge \widehat{x}_i and \widehat{x}_k at C_i , we cannot get independent values $\widehat{f}(\widehat{x}_i)$ and $\widehat{f}(\widehat{x}_k)$. If we keep \widehat{x}_i and \widehat{x}_k separate at C_i and propagate them separately, then we can get two distinct values \widehat{y} and \widehat{z} but we will not be able to map them to \widehat{x}_j and \widehat{x}_k . Thus we will not know which one of \widehat{y} and \widehat{z} should be propagated to R_j and which one to R_k . Thus only one value can be propagated and it should be \perp rather than $\widehat{\top}$.

The other interesting question that needs to be answered is: Can this method be used for non-separable frameworks in which the component lattice is $\{\widehat{\top}, \perp\}$? To



- From the calling context c_j , variable a is initialized but b is not.
- From the calling context c_k , both a and b are initialized.
- At R_i , data flow values of only the following forms are valid: $\langle *, \alpha \rangle$ and $\langle c_i, \alpha \rangle$.
- Our method can construct $\langle c_j, b \rangle$ at R_i but not $\langle c_j, a \rangle$.

FIGURE 9.4

Difficulty in reconstructing contexts for possibly uninitialized variables analysis.

see why propagating $\widehat{\perp}$ values using this method is not sufficient in the presence of non-separability, consider the problem of performing possibly uninitialized variables analysis as illustrated in Figure 9.4. A pair $\langle \psi, x \rangle$ indicates that variable x is possibly uninitialized and this fact has been discovered along the context ψ . The assignment $a = b$ dictates that a must be considered possibly uninitialized in all contexts in which b has been discovered to be possibly uninitialized. Even if we generate the pair $\langle c_i, a \rangle$ from the pair $\langle c_i, b \rangle$ after encountering the assignment statement $a = b$, there is no context information about a at C_i . Further, this method cannot handle general constraints that copy the context of b into the context of a whenever a context of b is reconstructed.

For the qualified data flow value X , the \top is \emptyset . In order to establish that this method computes *MFP* assignment in terms of X , we only need to argue about the termination. It follows from the fact that at each call node, the incoming context information is overwritten by the call site. During analysis, the number of tuples representing the synthesized data flow information at any node in procedure r can at most be $|\mathbb{G}\text{var}|$ and the number of tuples representing the inherited data flow information is bounded by $|\text{CallsTo}_r| \times |\mathbb{G}\text{var}|$.

Since Equations (9.1) and (9.2) cover all paths, they cover all interprocedurally valid *ifps* also. This ensures safety of data flow analysis. The precision follows from the fact that data flow analysis is restricted to interprocedurally valid paths only.

9.3 Interprocedural Analysis Using Unrestricted Contexts

The main limitation of interprocedural data flow analysis using a restricted context is that it requires reconstruction of context. This restricts the method to bit vector frameworks only. In this section we generalize the method to use unrestricted con-

text. This not only eliminates the need of reconstruction of contexts but also of the special context $*$ to represent synthesized data flow information. Further it allows propagation of any data flow value.

In the presence of recursion, unrestricted context could result in an infinite number of unbounded length call strings. Thus the main issue in unrestricted context approach is how to bound the length and the number of contexts. We first present the method without any concern for bounding the contexts. Then we present a general method of bounding contexts based on data flow values for data flow frameworks with finite lattices.

9.3.1 Using Call Strings to Represent Unrestricted Contexts

The call strings method uses $X = \langle \sigma, x \rangle$ as a qualified data flow value where σ is a call string representing a calling context. Special symbol λ denotes the empty call string. Concatenation $\lambda \cdot c_i$ results in the call string c_i .

The computation and propagation of qualified data flow value X is simpler in this method than in the previous method:

- If a pair $\langle \sigma, x \rangle$ reaches C_i , the context σ is extended and the pair $\langle \sigma \cdot c_i, x \rangle$ is propagated further.
- If a pair $\langle \sigma, x \rangle$ reaches R_i , there are two possibilities:
 - If the last call site in σ is c_i , i.e. $\sigma = \sigma' \cdot c_i$, it indicates a matching C_i and R_i and thus represents an interprocedurally valid path. In such a situation, the pair $\langle \sigma', x \rangle$ is propagated further. Note that σ' could be λ .
 - If the last call site in σ is not c_i , or σ is λ , it indicates an interprocedurally invalid path and the pair $\langle \sigma, x \rangle$ should not be propagated further.
- If a pair $\langle \sigma, x \rangle$ reaches an intraprocedural node n , the context does not change, only the data flow value changes. Let the flow function for block n be f_n . Then the pair $\langle \sigma, f_n(x) \rangle$ should be propagated further.

There is no need of the special context $*$ because a call string remembers the call sites corresponding to all unfinished calls. This makes it possible to propagate synthesized data flow information to appropriate callers without the need of reconstructing contexts. Note that the above description does not guarantee termination of call strings in recursive programs; we address this issue independently.

Since this method propagates all values in L rather than only $\widehat{\perp}$, multiple qualified data flow values reaching a node cannot be combined by plain set union. Instead, the data flow values associated with the same context must be merged. Thus the confluence of qualified data flow values is defined as follows:

$$\begin{aligned}
 X \uplus Y = & \{ \langle \sigma, x \sqcap y \rangle \mid \langle \sigma, x \rangle \in X, \langle \sigma, y \rangle \in Y \} \cup \\
 & \{ \langle \sigma, x \rangle \mid \langle \sigma, x \rangle \in X, \forall z \in L, \langle \sigma, z \rangle \notin Y \} \cup \\
 & \{ \langle \sigma, y \rangle \mid \langle \sigma, y \rangle \in Y, \forall z \in L, \langle \sigma, z \rangle \notin X \}
 \end{aligned}$$

The resulting data flow equations computing the qualified data flow values IN_n and OUT_n are as defined below:

$$IN_n = \begin{cases} \langle \lambda, BI \rangle & n \text{ is a } Start_{main} \\ \bigcup_{p \in pred(n)} OUT_p & \text{otherwise} \end{cases}$$

$$OUT_n = ConstGEN_n \cup DepGEN_n(IN_n) - (X - (ConstKILL_n - DepKILL_n(IN_n)))$$

where $ConstGEN_n = ConstKILL_n = DepKILL_n(X) = \emptyset$ and

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

The above data flow equations should be taken as a specification of the computation to be performed. In practice, we use a work list based iterative algorithm for computed IN_n and OUT_n rather than a round-robin iterative algorithm. This is because the effect of a change does not affect the entire supergraph directly.

The final data flow values at a node n are:

$$In_n = \bigcap_{\langle \sigma, x \rangle \in IN_n} x \quad (9.5)$$

$$Out_n = \bigcap_{\langle \sigma, x \rangle \in OUT_n} x \quad (9.6)$$

Example 9.2

Consider the program in Figure 9.1 on page 298 for call strings based interprocedural liveness analysis. A partial result of this analysis is shown in Figure 9.5 on the next page. It is complete in the sense that it includes all live variables at all program points. However, it is partial in the sense that it does not enumerate all call strings. For example, $\langle c_2c_3c_4, 1010 \rangle$ and $\langle c_1c_4, 0011 \rangle$ contained in IN_{Start_q} could be propagated to OUT_{C_2} , only to be ignored at C_2 because these call strings do not end with c_2 . However, some pairs that will not be ignored by the algorithm are $\langle c_2c_3c_4c_3, 1110 \rangle$ and $\langle c_1c_4c_3, 0111 \rangle$ that should be propagated from IN_{End_p} to OUT_{R_4} where new pairs $\langle c_2c_3c_4c_3c_4, 1110 \rangle$ and $\langle c_1c_4c_3c_4, 0111 \rangle$ would be created. This will further result in call strings $c_2c_3c_4c_3c_4c_3$ and $c_1c_4c_3c_4c_3$ and the construction of call strings will not terminate in spite of the fact that no new data flow information is generated.

Observe that in OUT_{Start_p} , the data flow information has been shown as 1100 + 1010 and 0101 + 0011 to highlight the fact that it is a merge of the data flow information propagated from the two successors of $Start_p$. \square

| Block | OUT_n | IN_n |
|-----------|---|--|
| End_m | $\langle \lambda, 0000 \rangle$ | $\langle \lambda, 1010 \rangle$ |
| R_2 | $\langle \lambda, 1010 \rangle$ | $\langle c_2, 1010 \rangle$ |
| C_2 | $\langle c_2, 1010 \rangle$ | $\langle \lambda, 1010 \rangle$ |
| n_2 | $\langle \lambda, 1010 \rangle$ | $\langle \lambda, 1110 \rangle$ |
| n_1 | $\langle \lambda, 1110 \rangle$ | $\langle \lambda, 0111 \rangle$ |
| R_1 | $\langle \lambda, 0111 \rangle$ | $\langle c_1, 0111 \rangle$ |
| C_1 | $\langle c_2 c_3, 1010 \rangle, \langle c_1, 0011 \rangle$ | $\langle \lambda, 0011 \rangle$ |
| $Start_m$ | $\langle \lambda, 0011 \rangle$ | $\langle \lambda, 1111 \rangle$ |
| End_q | $\langle c_2, 1010 \rangle, \langle c_2 c_3 c_4, 1110 \rangle, \langle c_1 c_4, 0111 \rangle$ | $\langle c_2, 1100 \rangle, \langle c_2 c_3 c_4, 1100 \rangle, \langle c_1 c_4, 0101 \rangle$ |
| R_3 | $\langle c_2, 1100 \rangle, \langle c_2 c_3 c_4, 1100 \rangle, \langle c_1 c_4, 0101 \rangle$ | $\langle c_2 c_3, 1100 \rangle, \langle c_2 c_3 c_4 c_3, 1100 \rangle, \langle c_1 c_4 c_3, 0101 \rangle$ |
| C_3 | $\langle c_2 c_3, 1010 \rangle, \langle c_1, 0011 \rangle, \langle c_2 c_3 c_4 c_3, 1010 \rangle, \langle c_1 c_4 c_3, 0011 \rangle$ | $\langle c_2, 1010 \rangle, \langle c_2 c_3 c_4, 1010 \rangle, \langle c_1 c_4, 0011 \rangle$ |
| $Start_q$ | $\langle c_2, 1010 \rangle, \langle c_2 c_3 c_4, 1010 \rangle, \langle c_1 c_4, 0011 \rangle$ | $\langle c_2, 1010 \rangle, \langle c_2 c_3 c_4, 1010 \rangle, \langle c_1 c_4, 0011 \rangle$ |
| End_p | $\langle c_2 c_3, 1100 \rangle, \langle c_1, 0111 \rangle, \langle c_2 c_3 c_4 c_3, 1100 \rangle, \langle c_1 c_4 c_3, 0101 \rangle$ | $\langle c_2 c_3, 1110 \rangle, \langle c_1, 0111 \rangle, \langle c_2 c_3 c_4 c_3, 1110 \rangle, \langle c_1 c_4 c_3, 0111 \rangle$ |
| n_3 | $\langle c_2 c_3, 1110 \rangle, \langle c_1, 0111 \rangle, \langle c_2 c_3 c_4 c_3, 1110 \rangle, \langle c_1 c_4 c_3, 0111 \rangle$ | $\langle c_2 c_3, 1100 \rangle, \langle c_1, 0101 \rangle, \langle c_2 c_3 c_4 c_3, 1100 \rangle, \langle c_1 c_4 c_3, 0101 \rangle$ |
| R_4 | $\langle c_2 c_3, 1110 \rangle, \langle c_1, 0111 \rangle$ | $\langle c_2 c_3 c_4, 1110 \rangle, \langle c_1 c_4, 0111 \rangle$ |
| C_4 | $\langle c_2, 1010 \rangle, \langle c_2 c_3 c_4 c_3, 1110 \rangle, \langle c_1 c_4 c_3, 0111 \rangle, \langle c_2 c_3 c_4, 1010 \rangle, \langle c_1 c_4, 0011 \rangle$ | $\langle c_2 c_3, 1010 \rangle, \langle c_1, 0011 \rangle$ |
| $Start_p$ | $\langle c_2 c_3, 1100 + 1010 \rangle, \langle c_1, 0101 + 0011 \rangle, \langle c_2 c_3 c_4 c_3, 1100 \rangle, \langle c_1 c_4 c_3, 0101 \rangle$ | $\langle c_2 c_3, 1010 \rangle, \langle c_1, 0011 \rangle, \langle c_2 c_3 c_4 c_3, 1010 \rangle, \langle c_1 c_4 c_3, 0011 \rangle$ |

FIGURE 9.5

Some call strings and associated values for interprocedural live variables analysis of our example program [Figure 9.1](#) on page 298.

Example 9.3

[Figure 9.6](#) on the facing page provides a recursive procedure r that reverses a linked list pointed to by the *head* pointer. Every call to r reverses the pointer of the *head* node by assigning the *previous* pointer and then moves the three pointers forward. As the recursion unwinds, the same operations are repeated nullifying the effect of the operations carried out before a recursive call was made. Thus, at the end of every call to r , regardless of the depth of recursion, the list is identical to what it was before the call.

[Figure 9.7](#) on page 306 shows the points-to graphs for some contexts discovered by the call strings method. Call site c_1 represents a call from *main*, whereas c_2 represents the recursive call. The points-to graph associated with

```

0. void r()
1. { /* Reverse the list */
2.   n = *h;
3.   *h = p;
4.   p = h;
5.   if (n != NULL)
6.   { h = n;
7.     r();
8.   }
9.   else
10.  { /* Reversed */
11.    { p = NULL;
12.      n = NULL;
13.      /* Process it */
14.    }
15.    /* Reverse it again */
16.    n = *h;
17.    *h = p;
18.    p = h;
19.    h = n;
20.  }

```

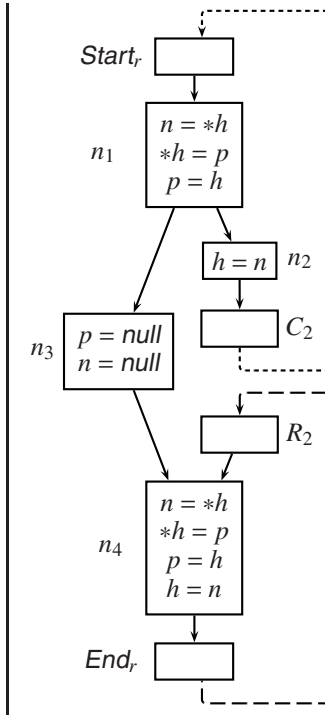


FIGURE 9.6

Example program for interprocedural points-to analysis. Pointer *h* is the *head* pointer, *p* is the *previous* pointer, and *n* is the *next* pointer.

End_r for the call string *c₁* represents the data flow information returned to the *main* procedure confirming that two reversals of the list have restored the list to its original structure. This is possible because the call strings method remembers the history of calls. This ensures that the method traverses interprocedurally valid paths only: In every path reaching the *main* procedure, for every occurrence of *Start_r*, there is a matching *End_r*, and vice-versa. Thus the number of times the flow function representing a single step of list reversal is applied remains equal for the control flow path entering the recursion and the control flow path leaving the recursion. \square

9.3.2 Issues in Termination of Call String Construction

In non-recursive programs, only a finite number of call strings can be constructed and the termination of the method is governed solely by the convergence of data flow values associated with the call strings. In recursive programs, termination of call string construction needs to be ensured explicitly. Once the termination of call strings is ensured, the usual fixed point criterion of data flow values can be applied

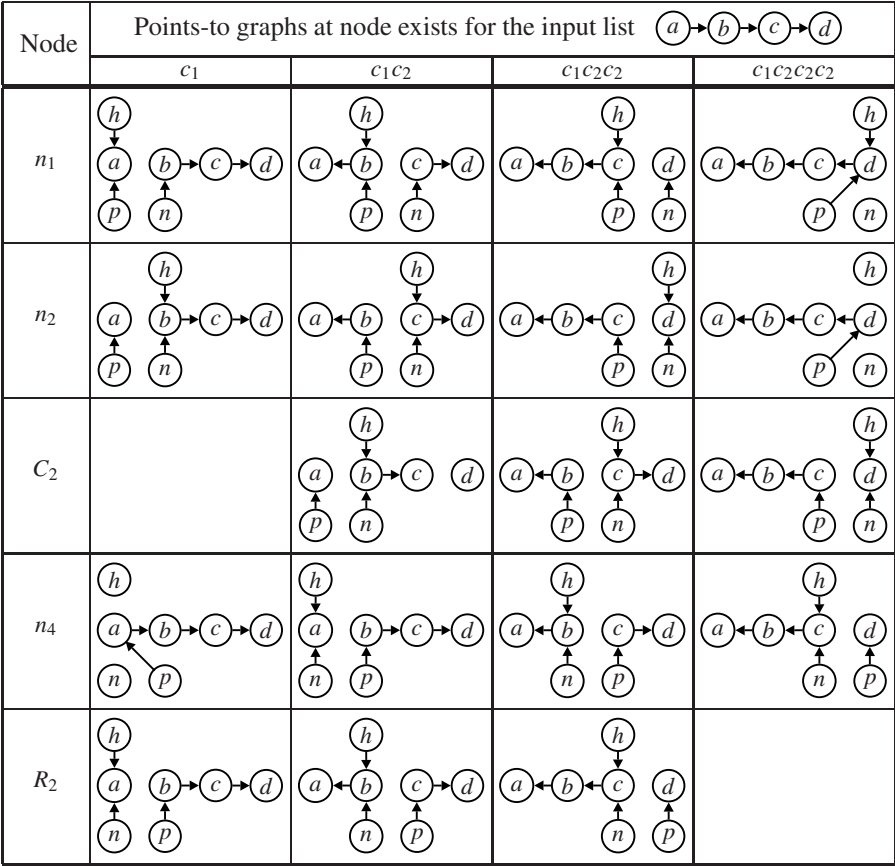


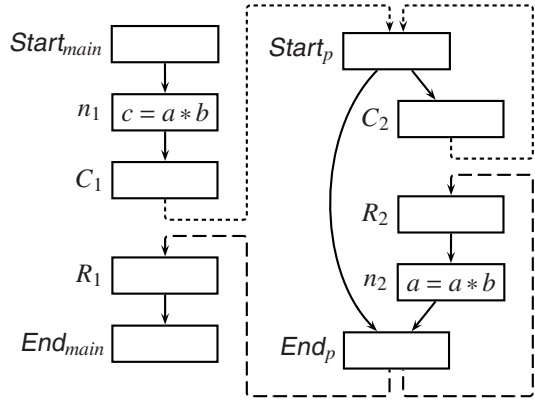
FIGURE 9.7 Points-to graphs in selected calling contexts. The *head* pointer points to variable *a* in the linked list.

to ensure the termination of analysis exactly as in iterative intraprocedural analysis.

A natural question that needs to be answered is whether call string construction can be terminated based on convergence of data flow values. In particular, we need to ascertain whether we need to continue constructing new call strings even when no new data flow information is generated for the new call strings. We have observed that in the case of intraprocedural analysis it is possible to compute the data flow values by a fixed point computation of the data flow variables associated with the nodes in a loop. The main difference between recursive contexts and loops is separation of data space. However, if we restrict ourselves to global variables only, is it possible to perform a fixed point computation of call strings? The next example shows that if call strings construction is stopped when data flow values reach a fixed point, it may result in an unsafe solution.

```

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



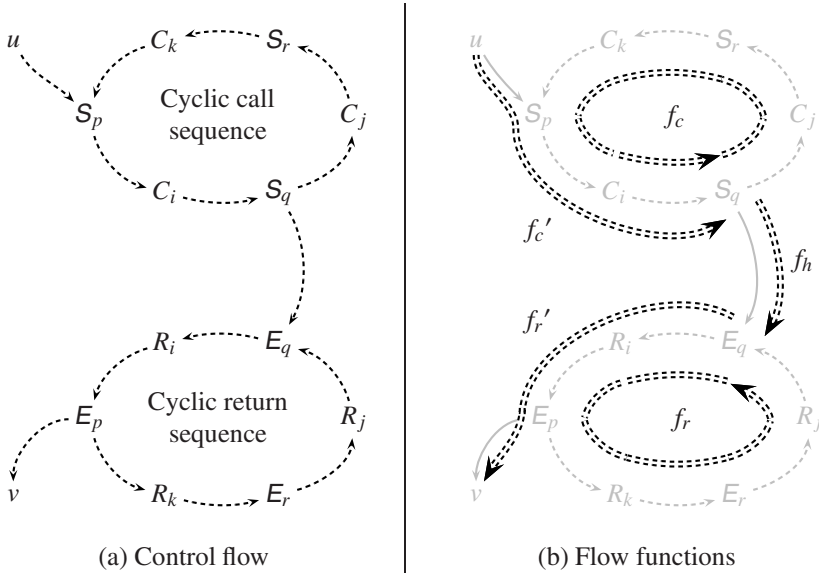
| Constructed call strings | Block | Iteration #1 | | Iteration #2 | |
|--|---------|--|--|--|--|
| | | IN_n | OUT_n | IN_n | OUT_n |
| $\langle c_1, 1 \rangle,$ $\langle c_1 c_2, 1 \rangle$ | R_2 | $\langle c_1, 1 \rangle,$ $\langle c_1 c_2, 1 \rangle$ | $\langle c_1, 1 \rangle$ | $\langle c_1, 0 \rangle,$ $\langle c_1 c_2, 1 \rangle$ | $\langle c_1, 1 \rangle$ |
| | n_2 | $\langle c_1, 1 \rangle$ | $\langle c_1, 0 \rangle$ | $\langle c_1, 1 \rangle$ | $\langle c_1, 0 \rangle$ |
| | End_p | $\langle c_1, 0 \rangle,$ $\langle c_1 c_2, 1 \rangle$ | $\langle c_1, 0 \rangle,$ $\langle c_1 c_2, 1 \rangle$ | $\langle c_1, 0 \rangle,$ $\langle c_1 c_2, 1 \rangle$ | $\langle c_1, 0 \rangle,$ $\langle c_1 c_2, 0 \rangle$ |
| $\langle c_1, 1 \rangle,$ $\langle c_1 c_2, 1 \rangle,$ $\langle c_1 c_2 c_2, 1 \rangle$ | R_2 | $\langle c_1, 1 \rangle,$ $\langle c_1 c_2, 1 \rangle,$ $\langle c_1 c_2 c_2, 1 \rangle$ | $\langle c_1, 1 \rangle,$ $\langle c_1 c_2, 1 \rangle$ | $\langle c_1, 0 \rangle,$ $\langle c_1 c_2, 0 \rangle,$ $\langle c_1 c_2 c_2, 1 \rangle$ | $\langle c_1, 0 \rangle,$ $\langle c_1 c_2, 1 \rangle$ |
| | n_2 | $\langle c_1, 1 \rangle,$ $\langle c_1 c_2, 1 \rangle$ | $\langle c_1, 0 \rangle,$ $\langle c_1 c_2, 0 \rangle$ | $\langle c_1, 0 \rangle,$ $\langle c_1 c_2, 1 \rangle$ | $\langle c_1, 0 \rangle,$ $\langle c_1 c_2, 0 \rangle$ |
| | End_p | $\langle c_1, 0 \rangle,$ $\langle c_1 c_2, 0 \rangle,$ $\langle c_1 c_2 c_2, 1 \rangle$ | $\langle c_1, 0 \rangle,$ $\langle c_1 c_2, 0 \rangle,$ $\langle c_1 c_2 c_2, 1 \rangle$ | $\langle c_1, 0 \rangle,$ $\langle c_1 c_2, 0 \rangle,$ $\langle c_1 c_2 c_2, 1 \rangle$ | $\langle c_1, 0 \rangle,$ $\langle c_1 c_2, 0 \rangle,$ $\langle c_1 c_2 c_2, 1 \rangle$ |

FIGURE 9.8

Available expressions analysis using call strings approach. Unless call string $c_1 c_2 c_2$ is constructed, it is not possible to find out that $a * b$ is not available at $Entry(n_2)$.

Example 9.4

Consider the program in Figure 9.8. Since variable a is modified in n_2 and is a global variable, the expression $a * b$ is not available at the entry of n_2 in any call of procedure p except for the most deeply nested call from which the recursion starts unwinding. When the call strings based method constructs call string $c_1 c_2$, the expression is available. When the pair $\langle c_1 c_2, 1 \rangle$ reaches R_2 , call site c_2 is removed and the data flow value is passed on through the pair $\langle c_1, 1 \rangle$. At the exit of n_2 , the qualified data flow value becomes $\langle c_1, 0 \rangle$ and is propagated to both R_1 and R_2 . However since the last call site c_1 does not

**FIGURE 9.9**

Modelling recursion. $Start_r$ and End_r for procedure r are abbreviated by S_r and E_r .

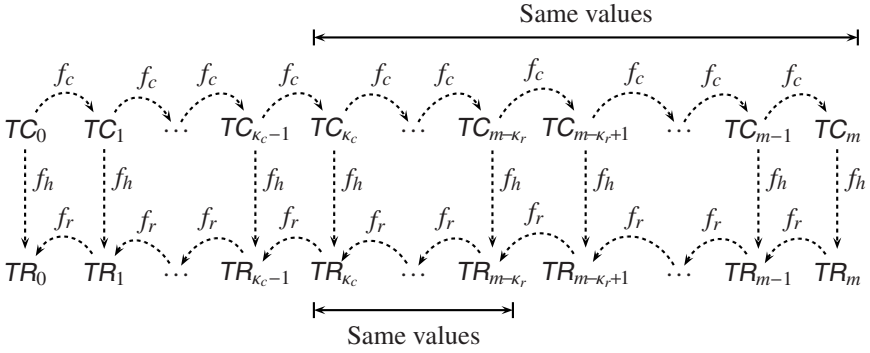
correspond to R_2 , this qualified value is ignored at R_2 . Thus the only way we can get the data flow value 0 at $Entry(n_2)$ is by ensuring that the cycle (R_2, n_2, End_p, R_2) is traversed at least once more. This is not possible unless call string $c_1c_2c_2$ is constructed in the cycle $(C_2, Start_p, C_2)$.

In terms of information flow paths, our analysis must cover the following *ifp*: $O_{n_2} \rightarrow l_{End_p} \rightarrow O_{End_p} \rightarrow l_{R_2} \rightarrow O_{R_2} \rightarrow l_{n_2}$ where l_{n_i} and O_{n_i} denote $Entry(n_i)$ and $Exit(n_i)$ respectively. Observe that node n_2 can be reached only via R_2 and the two occurrence of R_2 require at least two occurrences of C_2 . The shortest interprocedurally valid path that covers this *ifp* is:

$$(Start_m, n_1, C_1, Start_p, C_2, Start_p, C_2, Start_p, End_p, R_2, n_2, End_p, R_2, n_2)$$

The call string corresponding to this *ifp* is $c_1c_2c_2$. \square

This situation arises because a recursive call sequence in a program consists of two loops rather than one as illustrated in Figure 9.9. The first loop represents the control flow entering the recursive call while the other loop represents the control flow leaving the recursive calls. We call them as *cyclic call sequence* and *cyclic return sequence* respectively. We denote the flow functions associated with them by f_c and f_r respectively. The dashed line from $Start_q$ to End_q represents the recursion ending control flow path and the flow function associated with it is denoted by f_h . Since we do not require the call sites along a cyclic call sequence to be distinct, this figure models a general recursive path. In the most general case, there could be a path from the cyclic return sequence to the cyclic call sequence if there exists a recursive

**FIGURE 9.10**

Computation of data flow values along recursive paths. Dashed arrows indicate function applications.

call within a loop. Since we do not require f_c , f_r , and f_h to be independent, this does not affect our Modelling.

In a valid interprocedural path from u to any program point in the recursive procedures, the cyclic call sequence must be traversed at least as many times as the cyclic return sequence. For a valid interprocedural path from u to v , the cyclic call sequence must be traversed exactly as least as many times as the cyclic return sequence.

For forward data flow problems, call strings are constructed when the cyclic call sequence is traversed. Let the sequence of call sites $(\dots c_i \dots c_j \dots c_k \dots)$ along the cyclic call sequence from $Start_q$ back to $Start_q$ be represented by σ_c . Each application of f_c suffixes σ_c to every call string reaching $Start_q$. These call strings are consumed when the corresponding cyclic return sequence is traversed. Each application of f_r requires traversing the cyclic return sequence once. In the process, the last occurrence of σ_c is removed from every call string. Thus, f_r can be applied only as many times as the maximum number of σ_c in any call string reaching the entry of End_p . Note that the application of f_c does not have such a requirement because the call strings are constructed rather than consumed while applying f_c .

In order to guarantee safety of interprocedural data flow analysis, the call strings should be long enough to allow computation of all possible data flow values in both cyclic call and return sequences. We quantify this length in terms of a *fixed point closure bound*. A fixed point closure bound of a function h is the smallest number $n > 0$ such that $\forall x, h^{n+1}(x) = h^n(x)$.

Let the fixed point closure bound of f_c be κ_c and that of f_r be κ_r . Let the number of occurrences of σ_c in the longest call string be $m > \kappa_c$. Let the qualified data flow value reaching $Start_q$ in Figure 9.9 on the facing page be $\langle \sigma, x \rangle$. Let the sequence of the qualified data flow values computed at $Start_q$ be denoted by $\langle \sigma \cdot \sigma_c^i, TC_i \rangle$. We

know that

$$TC_i = \begin{cases} \mathbf{x} & i = 0 \\ f_c(TC_{i-1}) & 1 \leq i \leq m \end{cases}$$

This recurrence trivially reduces to:

$$TC_i = \begin{cases} f_c^i(\mathbf{x}) & 0 \leq i < \kappa_c \\ f_c^{\kappa_c}(\mathbf{x}) & \kappa_c \leq i \leq m \end{cases}$$

Let the sequence of the qualified data flow values computed at End_q be denoted by $\langle \sigma \cdot \sigma_c^i, TR_i \rangle$. Then,

$$TR_i = \begin{cases} f_h(TC_i) \sqcap f_r(TR_{i+1}) & 0 \leq i < m \\ f_h(TC_i) & i = m \end{cases}$$

The first term of \sqcap represents the data flow value along the path from $Start_q$ to End_q whereas the second term represents the data flow value computed along the cyclic return sequence. On substituting the values of TC_i , we get

$$TR_i = \begin{cases} TR_i \sqcap f_r(TR_{i+1}) & 0 \leq i < \kappa_c \\ TR_m \sqcap f_r(TR_{i+1}) & \kappa_c \leq i < m \\ f_h(f_c^{\kappa_c}(\mathbf{x})) & i = m \end{cases} \quad (9.7)$$

Since TR_i depends on TR_{i+1} , the final computation in cyclic return sequence starts from the last call string as illustrated in [Figure 9.10](#) on the previous page. Clearly, m should be at least $\kappa_c + \kappa_r$. If $m < \kappa_c + \kappa_r$, then some data flow values corresponding to unbounded recursion may not be computed. However, the values of κ_c and κ_r are not known a priori, and there should be some way of terminating the construction of call strings.

Example 9.5

Consider the program of [Figure 9.6](#) on page 305. Flow function f_c is the composition of the flow functions for n_1 and n_2 , f_h is the composition of the flow functions for n_1 , n_3 , and n_4 , whereas f_r is the flow function for block n_4 . If we ignore the *head* pointer which is conditionally advanced in the cyclic call sequence, $f_r^i(f_h^i(\mathbf{x})) = \mathbf{x}$. Further, for the given input value \mathbf{x} (consisting of a linked list of 4 elements), $\kappa_c = \kappa_r = 4$. Because of these reasons, it is sufficient to use $m = 4$ in this special case and stop call string construction when $c_1c_2c_2c_2c_2$ is created. This can be readily verified from [Figure 9.7](#) on page 306. In the general case, m should be larger than $\kappa_c + \kappa_r$ for all values of κ_c and κ_r . \square

9.4 Bounding Unrestricted Contexts Using Data Flow Values

A simple approach of allowing unrestricted call strings and yet bounding the overall set of call strings is to maintain in each procedure r , a single representative call string for each possible value in the lattice. This technique is deceptively simple and requires elaborate explanation. We outline the basis of this simple idea in terms of the following fundamental invariants of the call strings method:

- We observe that the same set of call strings reaches all program points in a procedure although they may have different values associated with them. As a consequence, if a mechanism is devised to ignore some call strings in a procedure (e.g., to represent them by other call strings), it would be possible to reconstruct them wherever they are required.
- If the call strings reaching a procedure are partitioned on the basis of data flow values, the equivalence classes remain unchanged in the procedure (the data flow value associated with an equivalence class may be different at different program point). More call strings may be included in an equivalence class across procedure calls because of construction of additional call strings.
- Finally, if there is a way of computing the correct value of $\sigma \cdot \sigma_c^{\kappa_c}$ at End_p , call strings $\sigma \cdot \sigma^i$, $\kappa_c < i \leq m$ need not be constructed. Further, there is not need to regenerate them explicitly; their implicit regeneration can be simulated by iterative computation of data flow values.

9.4.1 Call String Invariants

This section proves the call string invariants; the actual details of the method are presented in Section 9.4.2.

DEFINITION 9.5 *A context defining path from program point u to program point v is a valid interprocedural path from u to v that consists of only intraprocedural segments, call segments, or symmetric segments.*

If a context defining path contains return segments, they are suffixes of symmetric segments. For the purpose of our discussion, we restrict a context defining path to the significant nodes appearing in it. Thus each adjacent pair of nodes in a context defining path may correspond to many distinct intraprocedural segments.

DEFINITION 9.6 *A program point v is context dependent on program point u , denoted $v \in Cd(u)$, if there is a context defining path from u to v .*

Given procedure r , $\mathbf{Cd}(\text{Start}_r)$ contains all program points within r and all program points within all callees in every call chain starting in r . For $v \in \mathbf{Cd}(u)$, we use $\mathbf{Cdp}(u, v)$ to denote the set of context defining paths from u to v and $\mathbf{Cs}(u, v)$ to denote the set of call strings corresponding to paths in $\mathbf{Cdp}(u, v)$.

Let $\mathbf{dfVal}(\sigma, u)$ denote the value associated with call string σ at program point u .

DEFINITION 9.7 *Call strings σ_1 and σ_2 are equivalent at program point u , denoted $\sigma_1 \stackrel{u}{=} \sigma_2$, if $\{\sigma_1, \sigma_2\} \subseteq \mathbf{Cs}(\text{Start}_{\text{main}}, u)$ and $\mathbf{dfVal}(\sigma_1, u) = \mathbf{dfVal}(\sigma_2, u)$.*

We assume that the work list based interprocedural analysis traverses interprocedural paths such that all intraprocedural segments are processed completely before propagating data flow information from a significant node to another significant node. This can be achieved by maintaining two separate work lists: One for intraprocedural nodes and the other for significant nodes. A significant node is selected for processing only after ensuring that the work list of intraprocedural nodes is empty. We call such an interprocedural analysis algorithm as being *intraprocedurally eager*.

LEMMA 9.1

The calling contexts of all intraprocedural program points in a procedure are identical.

PROOF Obvious. ■

Calling contexts of a procedure depend on the callers so they cannot be different for different program points within the procedure. For a given call site $c_i \in \mathbf{CallsIn}_r$, $\mathbf{Exit}(C_i)$ and $\mathbf{Entry}(R_i)$ are assumed to logically belong to the callee procedure rather than r .

The following lemma shows that if σ_1 and σ_2 are transformed in the same manner by following the same set of paths, the values associated with them will also be transformed in the same manner and will continue to remain equal.

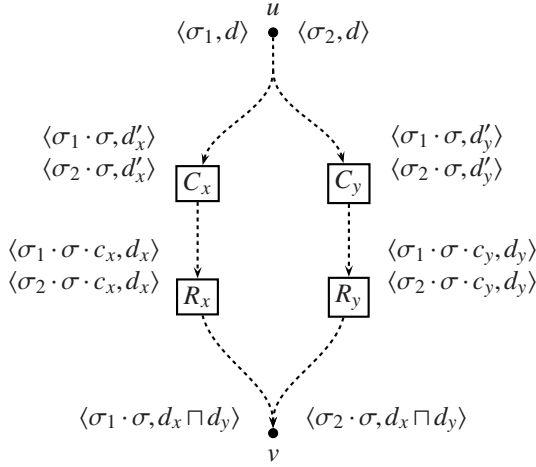
LEMMA 9.2

Consider a program point $v \in \mathbf{Cd}(u)$. Assume that the recursive paths in $\mathbf{Cdp}(u, v)$ are unbounded. When the work list of intraprocedural nodes is empty in an intraprocedurally eager call strings based method,

$$\sigma_1 \stackrel{u}{=} \sigma_2 \Rightarrow \forall \sigma \in \mathbf{Cs}(u, v), (\sigma_1 \cdot \sigma) \stackrel{v}{=} (\sigma_2 \cdot \sigma)$$

PROOF There are two cases to consider:

1. There is only one context defining path in $\mathbf{Cdp}(u, v)$ leading to a single sequence σ of call nodes that can be suffixed to both σ_1 and σ_2 .

**FIGURE 9.11**

Case 2 for Lemma 9.2.

In this case, the eager interprocedural analysis algorithm traverses exactly the same set of paths from u to v for computing the data flow information associated with the call strings $\sigma_1 \cdot \sigma$ and $\sigma_2 \cdot \sigma$. Thus the data flow values along the call strings $\sigma_1 \cdot \sigma$ and $\sigma_2 \cdot \sigma$ undergo the same change. Clearly,

$$dfVal(\sigma_1, u) = dfVal(\sigma_2, u) \Rightarrow dfVal(\sigma_1 \cdot \sigma, v) = dfVal(\sigma_2 \cdot \sigma, v)$$

2. $Cdp(u, v)$ contains multiple context defining paths corresponding to σ .

We prove this case by induction on the length of the maximal common suffix of all paths in $Cdp(u, v)$ which correspond to σ .

- *Basis.* The basis is the case when there is no common suffix.

For simplicity, assume that we have only two paths corresponding to σ as illustrated in Figure 9.11. Without any loss of generality, assume that R_x and R_y are the last nodes which are different.* Since both the paths from u to v correspond to a common call string σ , $Cs(u, Entry(R_x))$ contains a call string $\sigma \cdot c_x$ and $Cs(u, Entry(R_y))$ contains a call string $\sigma \cdot c_y$.

Let $dfVal(\sigma_1, u) = dfVal(\sigma_2, u) = d$. Assume that the path segment from u to $Entry(R_x)$ changes this value to d_x and the path segment from u to $Entry(R_y)$ changes this value to d_y .

*If two context defining paths differ in call nodes which are not followed by matching return nodes, then the two paths would not correspond to the same call string.

Since σ_1 and σ_2 reach u , $\sigma_1 \cdot \sigma \cdot c_x$ and $\sigma_2 \cdot \sigma \cdot c_x$ reach $\text{Entry}(R_x)$. Thus,

$$\text{dfVal}(\sigma_1 \cdot \sigma \cdot c_x, \text{Entry}(R_x)) = \text{dfVal}(\sigma_2 \cdot \sigma \cdot c_x, \text{Entry}(R_x)) = d_x$$

Similarly,

$$\text{dfVal}(\sigma_1 \cdot \sigma \cdot c_y, \text{Entry}(R_y)) = \text{dfVal}(\sigma_2 \cdot \sigma \cdot c_y, \text{Entry}(R_y)) = d_y$$

At the exit of the return nodes, the two call sites are removed. Hence at v , we get the pairs $\langle \sigma_1 \cdot \sigma, d_x \rangle$ and $\langle \sigma_2 \cdot \sigma, d_x \rangle$ along one path whereas along the other path we get the pairs $\langle \sigma_1 \cdot \sigma, d_y \rangle$ and $\langle \sigma_2 \cdot \sigma, d_y \rangle$. The data flow values for same call strings from different paths are merged and hence

$$\text{dfVal}(\sigma_1 \cdot \sigma, v) = \text{dfVal}(\sigma_2 \cdot \sigma, v) = d_x \sqcap d_y$$

This proves the basis case for two paths. Extending it to more than two paths is easy due to the finiteness of L . If there is a recursive call in a path from u to v , there will be infinitely many context defining paths corresponding to σ , each with a different number of matchings of some call and return nodes. However, since L is finite, these paths can be partitioned based on the data flow values corresponding to the call strings $\sigma_1 \cdot \sigma$ and $\sigma_2 \cdot \sigma$. Thus we will have a finite merge and inducting on the number of values (or number of partitions of paths from u to v) serves the purpose.

- **Inductive step.** Assume that all paths in $\text{Cdp}(u, v)$ which correspond to σ have a non-empty common suffix. Assume further that the lemma holds for a maximal common suffix consisting of k nodes. To show that it holds for a common suffix of $k+1$ nodes, observe that since all call strings traverse essentially the same path segment from node k to node $k+1$, the data flow values associated with the call string will be modified in the same way. Since the data flow values are equal after k nodes, they remain equal after $k+1$ nodes.

■

Note that this lemma assumes unbounded recursion. If call string construction is terminated after some repetition of cyclic call sequence (say m), then as illustrated in Figure 9.10 on page 309, the values of TR_i for $(m - \kappa_r + 1) \leq i \leq m$ are likely to be different in spite of the fact that the values of TC_i for the same range of i are identical ($f_c^{\kappa_c}(x)$). The lemma holds for the values of TR_i for $\kappa_c \leq i < (m - \kappa_r + 1)$. However, this exception arising due to bounded call strings does not matter because the associated values follow a strictly descending chain and converge on the least value. Hence the result of the merge of $TR_i, \kappa_c \leq i \leq m$ is the same as the values in those ranges of i for which the above lemma holds.

Intuitively, the values of TR_i for $(m - \kappa_r + 1) \leq i \leq m$ follow a strictly descending chain because they are repeatedly computed using the same function and are merged with the same value ($f_c^{\kappa_c}(x)$ in our case) at each step. We prove this in the following lemma.

LEMMA 9.3

Assume that the call strings method constructs call strings long enough so that all call strings $\sigma \cdot \sigma_c^i$, $0 \leq i \leq m$ are constructed where $m \geq \kappa_c + \kappa_r$ for all possible values of κ_c and κ_r . Then,

$$\forall \kappa_r, \quad TR_{m-\kappa_r} \sqsubseteq TR_i, \quad m - \kappa_r \leq i \leq m$$

PROOF We prove this by inducting on the distance of i from m by rewriting TR_i , $m - \kappa_r \leq i \leq m$ as TR_{m-j} , $0 \leq j \leq \kappa_r$ and by showing that

$$TR_{m-(j+1)} \sqsubseteq TR_{m-j}, \quad 0 \leq j < \kappa_r$$

The basis of induction is $j = 0$. Since $TR_{m-1} = TR_m \sqcap f_r(TR_m)$ it trivially follows that $TR_{m-1} \sqsubseteq TR_m$. For the inductive hypothesis, assume that $TR_{m-(j+1)} \sqsubseteq TR_{m-j}$. We need to show that $TR_{m-(j+2)} \sqsubseteq TR_{m-(j+1)}$. From the definition of TR_i ,

$$TR_{m-(j+2)} = TR_m \sqcap f_r(TR_{m-(j+1)}) \quad 0 \leq j \leq \kappa_r, \quad m > \kappa_c + \kappa_r \quad (9.8)$$

$$TR_{m-(j+1)} = TR_m \sqcap f_r(TR_{m-j}) \quad 0 \leq j \leq \kappa_r, \quad m > \kappa_c + \kappa_r \quad (9.9)$$

From the inductive hypothesis and monotonicity of flow functions,

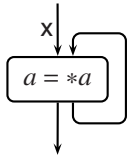
$$TR_{m-(j+1)} \sqsubseteq TR_{m-j} \Rightarrow f_r(TR_{m-(j+1)}) \sqsubseteq f_r(TR_{m-j})$$

The inductive step follows by comparing the right hand sides of Equations (9.8) and (9.9). ■

Observe the role of κ_c in the above proof. Since TR_{κ_c-1} does not have the first term as $f_h(f_c^{\kappa_c}(x))$ unlike TR_{κ_c} , a partial order relation between TR_{κ_c-1} and TR_{κ_c} cannot be established and lemma may not hold.

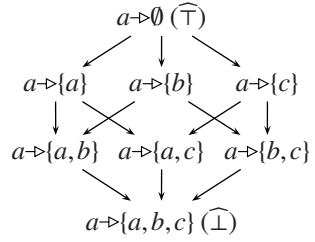
We have defined TC_i and TR_i for $Start_q$ and End_q in Figure 9.9 on page 308. In particular, the term TR_i for End_q involves a merge of the data flow values along the recursion ending path and the cyclic return sequence. For some other pair of program points, say $Start_r$ and End_r , the term TR_i may not be a merge of data flow values along two paths. However, the data flow values at all program point in the cyclic return sequence must converge. When the computation of a data flow value converges at a program point in a cycle, it must converge at each program point in the cycle. Further, the direction of convergence must be same for each program point.

This convergence immediately suggests that the data flow values associated with call strings $\sigma \cdot \sigma_c^i$, $\kappa_c \leq i \leq m$ are not required for the final data flow value in cyclic return sequences. This happens because when the data flow values that are being



$$\begin{aligned}
 \mathbf{x} &= \{a \mapsto \{b\}, b \mapsto \{c\}, c \mapsto \{b\}\} \\
 f^1(\mathbf{x}) &= \{a \mapsto \{c\}, b \mapsto \{c\}, c \mapsto \{b\}\} \\
 f^2(\mathbf{x}) &= \{a \mapsto \{b\}, b \mapsto \{c\}, c \mapsto \{b\}\} \\
 f^3(\mathbf{x}) &= \{a \mapsto \{c\}, b \mapsto \{c\}, c \mapsto \{b\}\}
 \end{aligned}$$

(a) \mathbf{x} is a periodic point with a period 2 for the flow function for pointer assignment $a = *a$.



(b) Lattice of *may* points-to information for variable a

FIGURE 9.12

Flow functions in Points-to analysis. Data flow value $v \mapsto S$ indicates that variable v may point to the variables contained in S .

merged follow a descending chain, only the last value in the chain matters in the overall merge and since our lattices are finite, all descending chains are finite and such a last value is guaranteed to exist.

THEOREM 9.1

Assume that the call strings method constructs call strings long enough so that all call strings $\sigma \cdot \sigma_c^i$, $0 \leq i \leq m$ constructed where $m \geq \kappa_c + \kappa_r$ for all possible values of κ_c and κ_r . Then for each program point v in a return sequence

$$\bigcap_{i=0}^m dfVal(\sigma \cdot \sigma_c^i, v) = \bigcap_{i=0}^{\max(\kappa_c)} dfVal(\sigma \cdot \sigma_c^i, v)$$

PROOF The values $dfVal(\sigma \cdot \sigma_c^i, v)$, $0 \leq i < \kappa_c$ may be different. However, due to the convergence of data flow values for subsequent call strings,

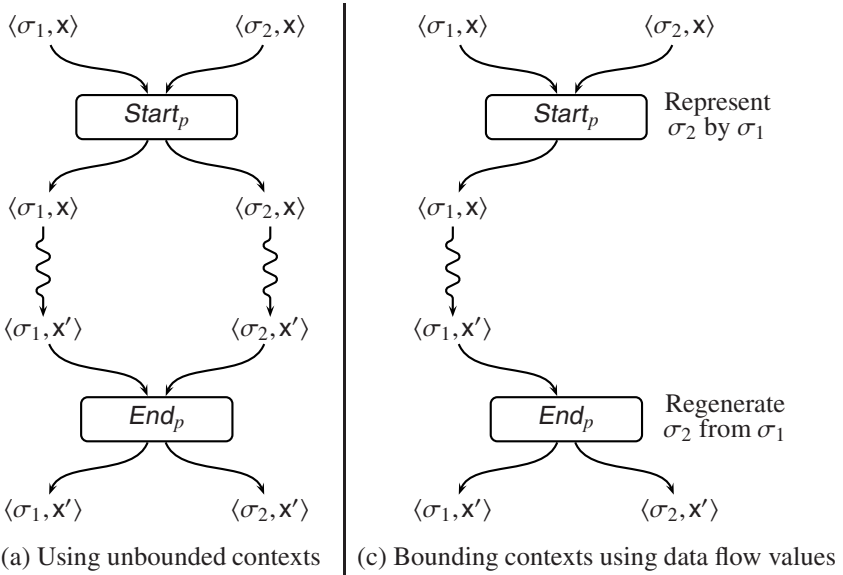
$$dfVal(\sigma \cdot \sigma_c^i, v) = dfVal(\sigma \cdot \sigma_c^{i+1}, v), \quad \kappa_c \leq i < m - \kappa_r$$

Thus $dfVal(\sigma \cdot \sigma_c^{\kappa_c}, v)$ is the least value for $\kappa_c \leq i \leq m$. Hence it is sufficient to merge the values of all call strings up to κ_c number of occurrence of σ_c . ■

An aside on flow function with periodic points

For a given function h and a value x , if $h^n(x) = x$ and $h^i(x) \neq x$, $0 < i < n$, then x is a *periodic point* of h with period n . A fixed point is a periodic point of period one. In general, flow functions can have periodic points of larger periods even if the functions are monotonic. This is possible only when functions compute incomparable values. Figure 9.12 shows an example of such a flow function from *may* Points-to analysis. \mathbf{x} is the data flow information reaching statement n from outside of the loop. Observe that f_n computes incomparable values in all successive applications.

We have restricted the discussion in this chapter to flow functions with period one. Extending the arguments to functions of larger periods is easy. Consider a flow

**FIGURE 9.13**

Representation and regeneration of equivalent call strings.

function which has period n for the incoming data flow value. Then, there are n periodic points instead of 1. In such a situation, instead of

$$dfVal(\sigma \cdot \sigma^{i+1}, Start_p) = dfVal(\sigma \cdot \sigma^i, Start_p), \quad \kappa_c \leq i \leq m$$

we have

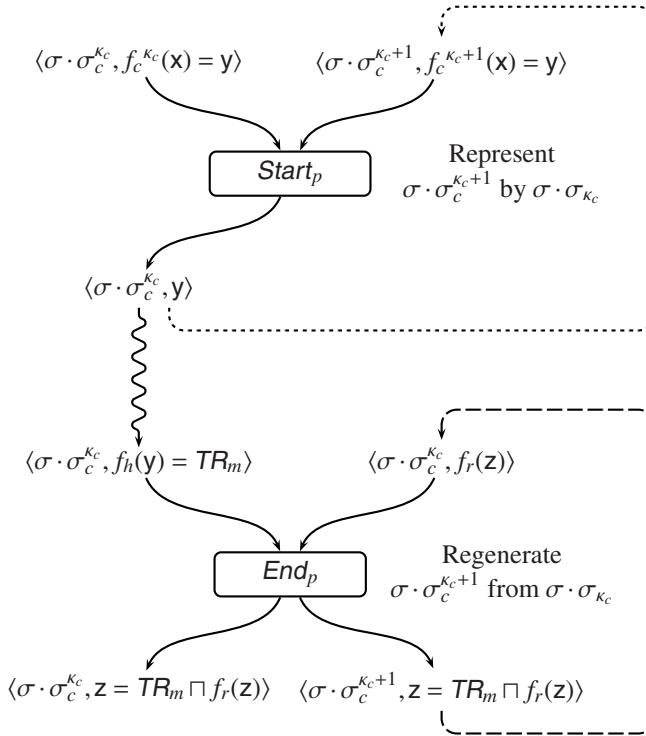
$$dfVal(\sigma \cdot \sigma^{i+n}, Start_p) = dfVal(\sigma \cdot \sigma^i, Start_p), \quad \kappa_c \leq i \leq m-n$$

The convergence holds for call strings corresponding to each periodic point independently. For periodic point i , Lemma 9.3 can be proved by inducting on the distance of the call string from the call string $\sigma \cdot \sigma^{m-i}$.

9.4.2 Value-Based Termination of Call String Construction

Given a set of cyclic call strings, Theorem 9.1 allows us to distinguish between two types of call strings:

- The call strings whose data flow values are relevant for the final result of data flow analysis. These call strings involve up to κ occurrences of any cyclic call sequence where κ is the largest possible value of κ_c .
- The call strings which facilitate a sufficient number of traversal over return segment to allow convergence of data flow values. These are the call strings

**FIGURE 9.14**

Representation and regeneration of cyclic call strings whose data flow values reach convergence in a cyclic call sequence. These call strings are used for convergence of data flow values in the corresponding cyclic return sequence.

that contain κ' additional occurrences of cyclic return sequences where κ' is the largest possible value of κ_r .

If there is some way of allowing traversal of a cyclic return sequence as many times as may be required, we may be able to terminate construction of redundant call strings in the corresponding cyclic call sequence. This is achieved as follows:

A single representative call string for an equivalence class within the scope of a maximal context dependent region is maintained and at the end of the region, all call strings belonging to each equivalence class are reconstructed. Some of them are constructed explicitly while some of them are constructed implicitly.

For procedure p , the decision of representation is taken at $Start_p$. This representation remains valid at all program points which are context dependent on $Start_p$.

End_p is the last such point and the call strings must be regenerated so that appropriate data flow values can be propagated to different callers of p . Similar to the scope of variables in a program, this representation may be “shadowed” by other context dependent regions created by procedure calls in the outer context dependent region.

Let $representative(x, Start_p)$ denote a uniquely selected call string which has value x at $Start_p$. The selection can be made based on some well defined criterion and the choice of this criterion is immaterial so long as it identifies a unique call string. One example of selecting a unique call string is to select the shortest call string from among the set of call strings that have the same data flow value. Another criterion could be to select the first call string that is listed in the associated data structure. The representation of call strings at $Start_p$ is defined as follows:

$$\forall \langle \sigma, x \rangle \in IN_{Start_p} : represent(\sigma, Start_p) = representative(x, Start_p)$$

The regeneration at End_p is performed as follows:

$$\begin{aligned} OUT_{Start_p} &= \{ \langle represent(\sigma, Start_p), x \rangle \mid \langle \sigma, x \rangle \in IN_{Start_p} \} \\ regenerate(\sigma, End_p) &= \{ \langle \sigma', y \rangle \mid represent(\sigma', Start_p) = \sigma, \langle \sigma, y \rangle \in IN_{End_p} \} \\ OUT_{End_p} &= \bigcup_{\langle \sigma, y \rangle \in IN_{End_p}} regenerate(\sigma, End_p) \end{aligned}$$

Regeneration copies the same data flow value to all call strings belonging to the same equivalence class. For general call strings this process has been illustrated in [Figure 9.13](#) on page 317. For call strings in recursive programs, this process facilitates iterative computation of data flow values in cyclic return sequences without having to construct redundant call strings in the corresponding cyclic call sequence. This has been illustrated in [Figure 9.14](#) on the preceding page.

The call string invariants presented in Section 9.4.1 are based on the following assumptions that should be honoured by work list based method used for call strings based interprocedural data flow analysis:

- The work list algorithm is assumed to be intraprocedurally eager. Hence data flow information should be propagated across procedure boundaries only when no further intraprocedural propagation is possible.

This can be handled by maintaining separate work lists for intraprocedural nodes and significant nodes. A significant node should be selected by the method only when there is no pending intraprocedural node.

- It is assumed that the functions in cyclic return sequence are applied only after the data flow values in the corresponding cyclic call sequence have reached a convergence. This matters only in those cases when there is a path from a cyclic return sequence to a cyclic call sequence e.g., when a function call is contained in a loop.

This can be handled by maintaining the following invariant in the work list of significant nodes: A call node always precedes any return node in the work list, regardless of when it is included in the work list.

| Step No. | Selected Node | Qualified Data Flow Value | | Remaining Work List | |
|----------|---------------|---|---|---------------------|-------------------|
| | | IN_n | OUT_n | Intraproc. Nodes | Significant Nodes |
| 1 | $Start_p$ | $\langle c_1, 1 \rangle$ | $\langle c_1, 1 \rangle$ | End_p | C_2 |
| 2 | End_p | $\langle c_1, 1 \rangle$ | $\langle c_1, 1 \rangle$ | | C_2, R_2 |
| 3 | C_2 | $\langle c_1, 1 \rangle$ | $\langle c_1 c_2, 1 \rangle$ | $Start_p$ | R_2 |
| 4 | $Start_p$ | $\langle c_1, 1 \rangle \langle c_1 c_2, 1 \rangle$ | $\langle c_1, 1 \rangle$ | End_p | C_2, R_2 |
| | | $(c_1 c_2 \text{ is represented by } c_1)$ | | | |
| 5 | End_p | $\langle c_1, 1 \rangle$ | $\langle c_1, 1 \rangle \langle c_1 c_2, 1 \rangle$ | | C_2, R_2 |
| | | $(c_1 c_2 \text{ is regenerated from } c_1)$ | | | |
| 6 | C_2 | No change | No change | | R_2 |
| 7 | R_2 | $\langle c_1, 1 \rangle \langle c_1 c_2, 1 \rangle$ | $\langle c_1, 1 \rangle$ | n_2 | |
| 8 | n_2 | $\langle c_1, 1 \rangle$ | $\langle c_1, 0 \rangle$ | End_p | |
| 9 | End_p | $\langle c_1, 0 \rangle$ | $\langle c_1, 0 \rangle \langle c_1 c_2, 0 \rangle$ | | R_2 |
| | | $(c_1 c_2 \text{ is regenerated from } c_1)$ | | | |
| 10 | R_2 | $\langle c_1, 0 \rangle \langle c_1 c_2, 0 \rangle$ | $\langle c_1, 0 \rangle$ | n_2 | |
| 11 | n_2 | $\langle c_1, 0 \rangle$ | No change | | |

FIGURE 9.15

Interprocedural data flow analysis of example program in Figure 9.8 on page 307 using value-based termination of call string construction.

- When representation is performed, it is assumed that the corresponding regeneration is guaranteed to be performed.

This can be ensured by adding End_r to the work list whenever representation is performed at $Start_r$; this includes the situation when an equivalence class remains same but the data flow value associated with the call strings in that equivalence class changes. It is possible that the data flow values do not change within procedure r after representation and hence End_r may never be added to the work list. In such a case, the new qualified data flow value may not be generated at End_p .

Example 9.6

Call strings based interprocedural data flow analysis using representation and regeneration of call strings for the example program in Figure 9.8 on page 307 has been illustrated in Figure 9.15. Observe that in step 2, R_2 is inserted in the work list after C_2 rather than before it. In step 4, $\langle c_1 c_2, 1 \rangle \in IN_{Start_p}$ is not propagated to OUT_{Start_p} as it is represented by $\langle c_1, 1 \rangle$. At End_p $\langle c_1 c_2, 1 \rangle$ is regenerated. This reaches R_2 where c_2 is removed and the resulting qualified data flow value $\langle c_1, 1 \rangle$ is propagated to n_2 . Due to the assignment to a in n_2 , this data flow value changes to $\langle c_1, 0 \rangle$ and is propagated to End_p where it is merged with $\langle c_1, 1 \rangle$ arriving from $Start_p$. This causes the value 0 to be

propagated as $\langle c_1c_2, 0 \rangle$ and $\langle c_1, 0 \rangle$. A subsequent traversal over the return sequence ensures that the data flow value become 0 at $Entry(n_2)$ also. \square

Representation and regeneration discards only those call strings which contain redundant values and performs the desired computation iteratively.

Recall that for the points-to analysis of program of Figure 9.6 on page 305, additional call strings are not required for convergence in cyclic return sequence. This does not influence our algorithm in any way; we leave it for the reader to verify that this method computes identical result as in Figure 9.7 on page 306.

THEOREM 9.2

The final data flow values computed by representing and regenerating call strings are identical to the values computed by a call strings method with an unbounded length call strings.

PROOF Regeneration explicitly constructs all acyclic call strings and all cyclic call strings containing $\kappa_c + 1$ occurrences of σ_c . At End_p , $\sigma \cdot \sigma_c^{\kappa_c+1}$ is regenerated and the data flow value associated with $\sigma \cdot \sigma_c^{\kappa_c}$ is propagated to it. From Equation (9.7) and Figure 9.14 on page 318, this value is TR_m . This value is then propagated as $\langle \sigma \cdot \sigma_c^{\kappa_c+1}, z = TR_m \rangle$ along the cyclic return sequence. This traversal removes the last occurrence of σ_c from $\sigma \cdot \sigma_c^{\kappa_c+1}$, computes $f_r(z)$, which is merged with the value of $\sigma \cdot \sigma_c^{\kappa_c}$ along the recursion ending path. Thus $dfVal(\sigma \cdot \sigma_c^{\kappa_c}, End_p) = TR_m \sqcap f_r(TR_m)$ after one traversal. This is same as the value associated with call string $\sigma \cdot \sigma_c^{m-1}$ where $m \geq \kappa_c + \kappa_r$. At End_p , this is again copied to the call string $\sigma \cdot \sigma_c^{\kappa_c+1}$ overwriting the previous value and the pair $\langle \sigma \cdot \sigma_c^{\kappa_c+1}, z = TR_m \sqcap f_r(TR_m) \rangle$ is propagated along the cyclic return sequence. The process repeats as long as new values are computed for $\sigma \cdot \sigma_c^{\kappa_c}$; effectively, traversal i over the cyclic return sequence computes the value T_{m-i} for $\sigma \cdot \sigma_c^{\kappa_c}$. The process terminates after κ_r traversals. This computes the desired value for $\sigma \cdot \sigma_c^{\kappa_c}$. \blacksquare

After the convergence of data flow values in a cyclic call sequence has been reached, this method replaces construction of the subsequent call strings by iteratively computing the data flow values in the corresponding cyclic return sequence using a pair of last two call strings.

Observe that representation is performed afresh every time any *Start* node is visited. On a subsequent visit to $Start_p$ of procedure p , representation could change because of the following reasons:

- A new call string with the value of an existing call string reaches $Start_p$.
- A new call string with a new value reaches $Start_p$.
- A call string that had reached $Start_p$ with a value x now reaches $Start_p$ with a different value x' .

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

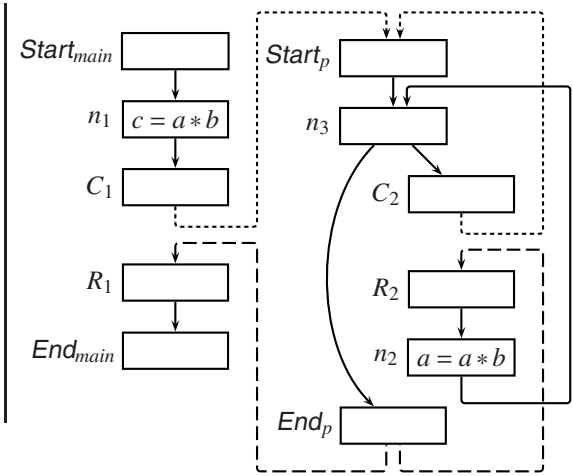


FIGURE 9.16 Modified program of Figure 9.8 on page 307. Expression $a * b$ is not available anywhere in procedure p .

In either case, End_p will be added to the work list. Thus all call strings will get regenerated with appropriate data flow values at End_p .

Example 9.7

Figure 9.16 contains a modified version of the program in Figure 9.8 on page 307. Since now the recursive call is in the loop, expression $a * b$ is unavailable in nodes $Start_p$ and C_2 also. A trace of the call strings method using value-based termination has been provided below.

| Step No. | Selected node | Qualified data flow value | | Remaining work list | |
|----------|---------------|---|---|---------------------|------------|
| | | IN_n | OUT_n | Intra. nodes | Sig. nodes |
| 1 | $Start_p$ | $\langle c_1, 1 \rangle$ | $\langle c_1, 1 \rangle$ | n_3 | |
| 2 | n_3 | $\langle c_1, 1 \rangle$ | $\langle c_1, 1 \rangle$ | End_p | C_2 |
| 3 | End_p | $\langle c_1, 1 \rangle$ | $\langle c_1, 1 \rangle$ | | C_2, R_2 |
| 4 | C_2 | $\langle c_1, 1 \rangle$ | $\langle c_1 c_2, 1 \rangle$ | $Start_p$ | R_2 |
| 5 | $Start_p$ | $\langle c_1, 1 \rangle \langle c_1 c_2, 1 \rangle$ | $\langle c_1, 1 \rangle$ | n_3, End_p | R_2 |
| | | $(c_1 c_2 \text{ is represented by } c_1)$ | | | |
| 6 | n_3 | $\langle c_1, 1 \rangle$ | $\langle c_1, 1 \rangle$ | End_p | C_2, R_2 |
| 7 | End_p | $\langle c_1, 1 \rangle$ | $\langle c_1, 1 \rangle \langle c_1 c_2, 1 \rangle$ | | C_2, R_2 |
| | | $(c_1 c_2 \text{ is regenerated from } c_1)$ | | | |
| 8 | C_2 | No change | No change | | R_2 |
| 9 | R_2 | $\langle c_1, 1 \rangle \langle c_1 c_2, 1 \rangle$ | $\langle c_1, 1 \rangle$ | n_2 | |
| 10 | n_2 | $\langle c_1, 1 \rangle$ | $\langle c_1, 0 \rangle$ | n_3 | |

| Step No. | Selected node | Qualified data flow value | | Remaining work list | |
|----------|---------------|---|---|---------------------|------------|
| | | IN_n | OUT_n | Intra. nodes | Sig. nodes |
| 11 | n_3 | $\langle c_1, 0 \rangle$ | $\langle c_1, 0 \rangle$ | End_p | C_2 |
| 12 | End_p | $\langle c_1, 0 \rangle$ | $\langle c_1, 0 \rangle \langle c_1 c_2, 0 \rangle$ | | C_2, R_2 |
| | | $(c_1 c_2 \text{ is regenerated from } c_1)$ | | | |
| 13 | C_2 | $\langle c_1, 0 \rangle$ | $\langle c_1 c_2, 0 \rangle$ | $Start_p$ | R_2 |
| 14 | $Start_p$ | $\langle c_1, 1 \rangle \langle c_1 c_2, 0 \rangle$ | $\langle c_1, 1 \rangle \langle c_1 c_2, 0 \rangle$ | n_3 | R_2 |
| | | $(\text{Representation has changed})$ | | | |
| 15 | n_3 | $\langle c_1, 0 \rangle \langle c_1 c_2, 0 \rangle$ | $\langle c_1, 0 \rangle \langle c_1 c_2, 0 \rangle$ | End_p | C_2, R_2 |
| 16 | End_p | $\langle c_1, 0 \rangle \langle c_1 c_2, 0 \rangle$ | No change | | C_2, R_2 |
| 17 | C_2 | $\langle c_1, 0 \rangle \langle c_1 c_2, 0 \rangle$ | $\langle c_1 c_2, 0 \rangle \langle c_1 c_2 c_2, 0 \rangle$ | $Start_p$ | R_2 |
| 18 | $Start_p$ | $\langle c_1, 1 \rangle \langle c_1 c_2, 0 \rangle$ $\langle c_1 c_2 c_2, 0 \rangle$ | No change | End_p | R_2 |
| | | $(c_1 c_2 c_2 \text{ is represented by } c_1 c_2)$ | | | |
| 19 | End_p | No change | $\langle c_1, 0 \rangle \langle c_1 c_2, 0 \rangle$ $\langle c_1 c_2 c_2, 0 \rangle$ | | R_2 |
| | | $(c_1 c_2 c_2 \text{ is regenerated from } c_1 c_2)$ | | | |
| 20 | R_2 | $\langle c_1, 0 \rangle \langle c_1 c_2, 0 \rangle$ $\langle c_1 c_2 c_2, 0 \rangle$ | $\langle c_1, 0 \rangle \langle c_1 c_2, 0 \rangle$ | n_2 | |
| 21 | n_2 | $\langle c_1, 0 \rangle \langle c_1 c_2, 0 \rangle$ | $\langle c_1, 0 \rangle \langle c_1 c_2, 0 \rangle$ | n_3 | |
| 22 | n_3 | No change | No change | | |

Observe that first the call string $c_1 c_2$ is represented by c_1 but since the call is in a loop, after unwinding the recursion once, the data flow value 0 reaches C_2 along the call string c_1 . This changes the representation at $Start_p$ and the call string $c_1 c_2$ must be explicitly propagated further. Eventually, call string $c_1 c_2 c_2$ has the same value as $c_1 c_2$. This results in a different representation and the data flow analysis terminates after a few steps. \square

THEOREM 9.3

Using the value-based termination of call strings, the maximum number of call strings at any internal program point is $|L|$.

PROOF At $Exit(Start_p)$ for any procedure p , the call strings are partitioned by the data flow values associated with them and there can be at most $|L|$ distinct data flow values. \blacksquare

THEOREM 9.4

Let the maximum number of call sites in any acyclic call chain be K . Then, using the value-based termination of call strings, the maximum length of any

call string is $K \times (|L| + 1)$.

PROOF Consider a call string $\sigma = \dots (C_i)^1 \dots (C_i)^2 \dots (C_i)^3 \dots (C_i)^j \dots$ where $(C_i)^j$ denotes j^{th} occurrence of C_i . Let $j \geq |L| + 1$ and let C_i call procedure p . The set of call strings reaching p is prefix closed in the following sense: All prefixes of σ ending in C_i must reach entry **Start** _{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 **Start** _{p} and the longer prefix will get represented by the shorter prefix. Since one more C_i is suffixed to discover fixed point, $j \leq |L| + 1$. In the worst case, all call sites may occur in σ thus the worst case length of any call string is $K \times (|L| + 1)$. ■

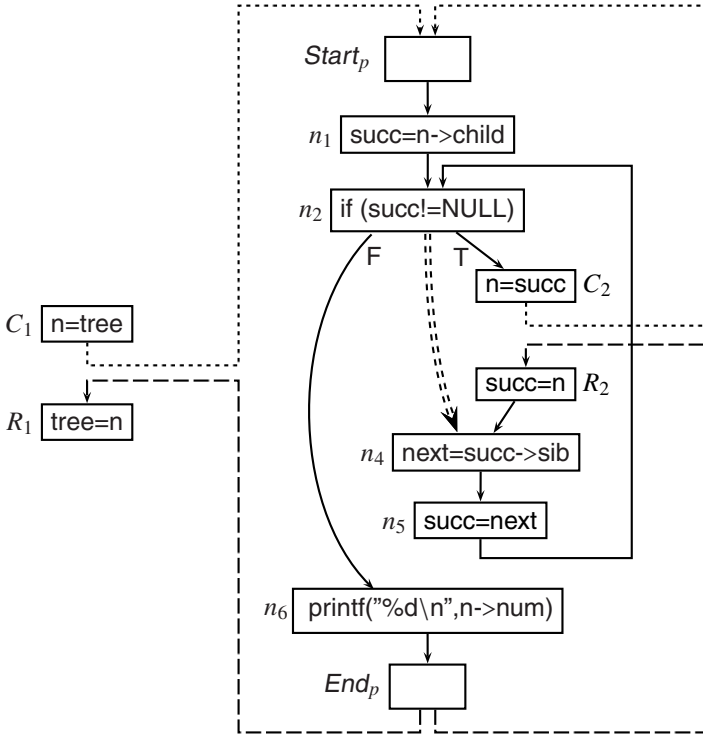
9.5 The Motivating Example Revisited

It is appropriate that our explanation of data flow analysis in this book should end with the example that it began with. This section presents context sensitive interprocedural liveness analysis of the program in Section 1.1.

The examples in this part have considered programs with global variables. However, our motivating example from Figure 1.1 on page 2 contains local pointer variables that are passed as actual parameters. As observed in Section 7.5, this requires data flow information to be propagated between the actual parameters and formal parameters. We model this using a couple of assignments and a special edge in the supergraph as illustrated in Figure 9.17 on the facing page.

For correct Modelling of local pointer variables as actual parameters, we need to assign them to formal variables in the call node (C_2 in our case) and restore them in the return node (R_2 in our case). The assignment in C_2 indicates that the heap memory reachable from succ is reachable from n in a recursive call. The assignment in R_2 indicates that the heap memory reachable from n in a recursive call is reachable from succ in an outer call. Besides, we need to bypass the call by an edge because the local variables are available in the program fragment beyond the call due to call by copy semantics. We have achieved this by adding edge $n_2 \rightarrow n_4$. In the absence of this edge, if formal parameter n is made null in procedure dfTraverse our assignment in R_2 will make succ null in node n_4 . Since the pointer has been passed by copy, this is incorrect. The assignment is required because the heap cells reachable from succ could be influenced by n but the address contained in succ is not modified because succ is a local variable and is not passed by reference.

Liveness analysis is a backward analysis. Hence we interchange the roles of call nodes and return nodes. Now a call site is appended and call strings grow when a return node is visited. Call sites are removed at the call nodes. By the same token, representation is performed at **End** _{r} of procedure r and regeneration is performed

**FIGURE 9.17**

Supergraph for procedure `dfTraverse` from the program in Figure 1.1 on page 2. Observe the assignments in call and return nodes and the edge $n_3 \rightarrow n_4$ for handling parameters.

at $Start_r$. Besides, a return node always precedes the corresponding call node in the work list of significant nodes. Note that the recursive call in this example is contained in a loop and hence we can expect the representation made at End_p to change.

Our data flow values are access graphs as defined in Section 4.4.3. We use the data flow equations defined in Section 4.4.4 for computing the effect of intraprocedural nodes on access graphs representing explicit liveness. Since field name `sib` is dereferenced only in node n_4 , summarization can be achieved without subscripting this field name with the node number. Similar remarks apply to the field name `child`. Hence, we drop the subscripts of field names.

The final data flow information is provided in Figure 9.18 on page 327. Below we list some path fragments to show the flow of information:

- $\rho = (End_p, n_6, n_2, n_5, n_4, n_2, n_5, n_4, R_2, n_6, n_2)$

The data flow value at the start of ρ is $\langle c_1, \mathcal{E}_G \rangle$ and the data flow value at the

end of ρ is $\left\langle c_1 c_2, \Rightarrow n \rightarrow \text{sib} \right\rangle$.

- Further traversal of n_2, n_1 results in the data flow value $\left\langle c_1 c_2, \Rightarrow n \rightarrow \begin{array}{c} \text{child} \\ \text{sib} \end{array} \right\rangle$.

- Traversal of n_2, n_5, n_4 creates the liveness graph $\Rightarrow \text{succ} \rightarrow \text{sib}$. A further traversal of n_2, n_1 results in the graph $\Rightarrow n \rightarrow \text{child} \rightarrow \text{sib}$. When this combines with the data flow value at n_1 obtained in the previous step, we get the

qualified data flow value $\left\langle c_1 c_2, \Rightarrow n \rightarrow \begin{array}{c} \text{child} \\ \downarrow \\ \text{sib} \end{array} \right\rangle$.

- The above data flow value reaches n_2 along the path $n_1, \text{Start}_p, C_2, n_1$ after

removing the call string suffix c_2 as $\left\langle c_1, \Rightarrow n \rightarrow \begin{array}{c} \text{child} \\ \downarrow \\ \text{sib} \end{array} \right\rangle$. Further, it reaches

End_p along the path $n_2, n_5, n_4, R_2, \text{End}_p$ as $\left\langle c_1 c_2, \Rightarrow n \rightarrow \begin{array}{c} \text{child} \\ \downarrow \\ \text{sib} \end{array} \right\rangle$.

We leave it for the reader to find out how the other edges get included in the above liveness graph and how the graphs are propagated to various nodes in the supergraph.

Observe that in the liveness graphs at the entry of n_4 , there is no edge from `succ` to `child`. Further, there is no graph rooted at `succ` at the entry of n_5 . This confirms our conclusion in Section 1.1 that the pointer `succ` can be freed between n_4 and n_5 .

Also note that the access path $n \rightarrow \text{child}$ is not live in nodes n_2, n_4, n_5 , and n_6 in the data flow information in Figure 9.18. However, it is live in the same nodes in the data flow information computed with conservative interprocedural summarization in Section 4.4.5. This is because $n \rightarrow \text{child}$ is not explicitly live; it is only implicitly live in that it is aliased to an access path that is explicitly live.

9.6 Summary and Concluding Remarks

This chapter has explored the approach of computing distinct values for distinct contexts instead of constructing context independent functions. Bit vector frameworks are amenable to such an analysis when the context is restricted to immediate caller. This method overwrites the context at every call and recovers it after the call is over.

A natural generalization of this method is to remember the entire call history in the form of a call string. This method is attractive because it is simple and general. Beside, it is context sensitive and hence computes precise data flow information. The

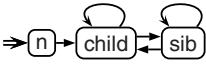
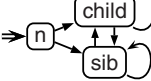
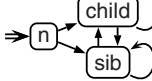

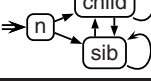
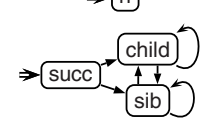
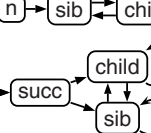
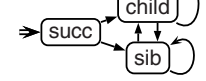
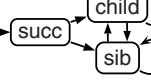
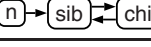
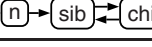
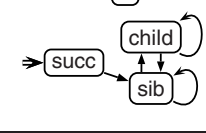
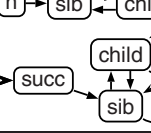
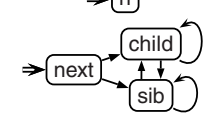
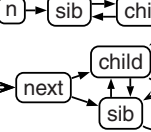
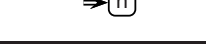
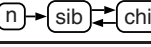
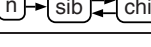
| Node | Liveness Graphs at the Entry of Nodes | | |
|-----------|---|---|---|
| | c_1 | c_1c_2 | $c_1c_2c_2$ |
| $Start_p$ |  |  |  |
| n_1 |  |  | |
| n_2 |  |  | |
| C_2 |  |  | |
| R_2 | |  |  |
| n_4 |  |  | |
| n_5 |  |  | |
| n_6 |  |  | |
| End_p | \mathcal{E}_G |  | $c_1c_2c_2$ is represented by c_1c_2 |

FIGURE 9.18
Interprocedural liveness analysis of heap data for the program in [Figure 9.17](#).

main difficulty in this method is that the number and length of call strings could be exponentially large. Further, in the case of recursive programs, the termination of the construction of call strings must be explicitly ensured. This can be achieved by adapting the “overwrite-and-recover” technique from the method that uses restricted

contexts. This adaptation results in call strings with equivalent values being represented by a single call string at *Start* of a procedure and regenerating the represented call string at the *End*.

The value-based termination criterion presented in this chapter is different from the original termination criterion of constructing all call strings up to the length of $K \times (|L| + 1)^2$ where K is the maximum number of call sites and L is the lattice. This number reduces to $3K$ for bit vector frameworks. This termination length results in a combinatorially large number of call strings. From Theorem 9.4, when value-based termination criterion is used, the worst case length of a call string reduces to $K \times (|L| + 1)$. Empirical measurements show a dramatic reduction in the number and maximum length of call strings compared to those in the original method.

9.7 Bibliographic Notes

The restricted context based analysis presented in this chapter is based on the work by Myers [79]. The call strings method was proposed by Sharir and Pnueli [93]. The termination criterion using convergence of data flow values has been proposed by Khedker and B. Karkare [61]. An orthogonal approach of reducing the space requirements in a context sensitive value-based interprocedural analysis is to use BDDs to encode data flow information. This has been proposed by Whaley and Lam [104]. They have found that this makes the method scalable. Although their approach is context insensitive in recursive contexts, the key idea of using BDDs to increase scalability seems very useful.

Since *ifps* in bit vector frameworks consist only of identity functions, it is possible to use an alternative method of terminating call string construction. As shown by B. Karkare and Khedker [54], it is sufficient to construct all call strings in which a call site appears at most three times. Note that this is different from Sharir and Pnueli's termination length of $3K$. In Sharir and Pnueli's method, if the length of a call string is smaller than $3K$, it is extended even if it results in four occurrences of a call cite. Although the worst case length in B. Karkare and Khedker's method is same, empirical measurements of interprocedural reaching definitions analysis shows a significant reduction in the number and maximum length of call strings.

Sharir and Pnueli [93] also present an approximate call strings method in which call string suffix of a fixed length k is remembered. This retains context sensitivity for call depths of k but for the call sequences beyond this depth, the method essentially becomes context insensitive. Effectiveness of this method has been empirically measured by Martin [72] who concluded that a value of $k > 2$ did not increase the precision significantly for constant propagation. Khedker and B. Karkare [61] have also presented an approximate version where the imprecision can be adjusted on demand. The basic idea is to allow say k occurrences of a call site in a call string and use representation and regeneration for all such call strings. When the call string

grows and the number of occurrences of a call site exceeds k , the data flow values are computed iteratively by retaining the same call string instead of extending it. Unlike Sharir and Pnueli's approximate method, this method is context sensitive until k unfoldings of recursive calls.

The interprocedural points-to analysis by Emami, Ghiya and Hendren [34] can be viewed as a value-based approach. It uses a variant of call graph called an *invocation graph* in which recursive invocations of procedures result in creating two nodes for a procedure: One node is recursive whereas the other node is approximate. Thus it is context sensitive in the first unfolding of recursion but context insensitive beyond that. We leave it for the reader to verify that the Emami's method computes imprecise points-to graphs for the program in [Figure 9.6](#) on page 305 compared to the points-to graphs in [Figure 9.7](#) on page 306 computed using call strings method.