Chapter 4

Interprocedural Points-to Analysis for Scalar Variables

This chapter is concerned with the interprocedural abstract stack analysis which is the process of collecting the points-to information in the presence of function calls. This is more complicated than the intraprocedural abstract stack analysis discussed in the previous chapter because it takes care of: (i) the renaming process between formal parameters and actual parameters of pointer-type, (ii) the effect of function calls on the points-to information involving global variables, (iii) the points-to information for variables of pointer-type which are returned by a function, and (iv) the effect of recursive function calls.

To perform the interprocedural analysis, we make use of a special representation for the interprocedural structure, namely the *invocation graph*. We first explain the construction of the invocation graph in Section 4.1. Then, we discuss the interprocedural analysis. Since the methods for recursive function calls are more complicated, we first consider the interprocedural analysis without the presence of recursive function calls in Section 4.2. Then, the recursive function calls are included in Section 4.3.

4.1 Invocation Graph

In this section, we explain the construction of the invocation graph in the presence of both recursive and non-recursive function calls. The construction of the invocation graph is independent of points-to analysis and can be used by any other analysis. Before presenting the details, let us first explain the following terminology: If a function `f` calls another function `g`, `f` is called the *caller* and `g` is called the *callee*.

An invocation graph captures the activation order and structure of function calls in a program. Our invocation graph is made up of: (i) a set of nodes representing function call instances, and, (ii) a set of edges specifying the callee-caller relationships between functions.

The following are three major reasons for using an invocation graph:

1. Our invocation graph is specially useful in the presence of recursive function calls. This is discussed in Section 4.3.
2. By using an invocation graph one can save the analysis information of each call separately. This can be used for the optimization purposes. Readers are referred to the section "future work" in Chapter 10 for further information on this subject.
3. The invocation graph can be used in conjunction with function-pointers. Readers are referred to the Section 8.2.2 for further information on this subject.

In the case of non-recursive function calls, the invocation graph is in the form of an ordinary tree while for recursive function calls it is a tree with some implicit back-edges. We first explain the non-recursive invocation graph and then extend it to include recursive calls.

The nodes in a non-recursive invocation graph are called *ordinary nodes* and the edges are called *calling arcs*. To build a non-recursive invocation graph, for each function, we collect the list of all the functions who gets called by that function. Then, we recursively construct the invocation graph starting from the `main` by linking the caller and the callee functions. Consider the example shown in Figure 4.1. Figure 4.1(b) shows the list of functions called by each function. For example, function `main` calls three functions, `f()`, `g()`, and `h()`. The construction of the invocation graph can be understood by following the steps shown in Figure 4.1(c).

Next, we construct an invocation graph for a program with a recursive function call using the same steps as explained before. Consider the example given in Figure 4.2. Using the previous method, we obtain an infinite call to function `f`. This is shown in Figure 4.2(c).

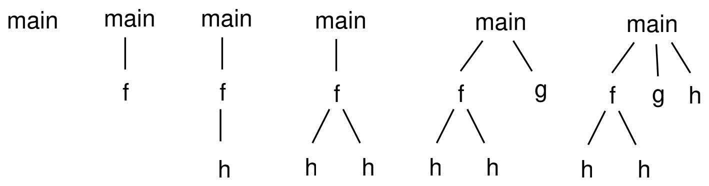
To solve this problem, the number of function calls has to be approximated. This is achieved by cutting down the infinite number of the calls to the function `f()` into two calls: (i) the first appearance of the function call in the invocation graph, namely *recursive node* denoted as `f-R`, and (ii) the second appearance of the function call in the invocation graph, namely *approximate node* denoted as `f-A`. To set up the correspondence between an approximate node and the corresponding recursive node,

```
main ( )
{
  f ( ) ;
  g ( ) ;
  h ( ) ;
}
f ( )
{
  h ( ) ;
  h ( ) ;
}
g ( )
{
}
h ( )
{
}
```

(a) C program

Func	Func-list
main ( )	f -> g -> h
f ( )	h -> h
g ( )	
h ( )	

(b) 'Func' is the function name and 'Func-list' is the list of functions who get called by 'Func'



(c) The steps of generation of invocatin graph from left to right

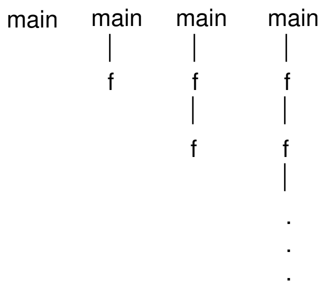
Figure 4.1: An example for the construction of the invocation graph.

```
main ( )
{
  f ( ) ;
}

f ( )
{
  if ( . . . )
    f ( ) ;
}
```

(a) C program

Func	Func-list
main ( )	f
f ( )	f

(b) 'Func' is the function name and 'Func-list' is the list of functions who get called by 'Func'



(c) The steps of generation of invocation graph from left to right

Figure 4.2: An example for the construction of the recursive invocation graph.

a link is established from the approximate node to the recursive node in the invocation graph. Figure 4.3 shows the extended invocation graph for the example of Figure 4.2.

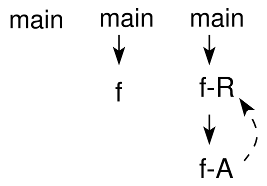
Now that the idea is clear, we give the formal definition and a precise algorithm. A call is *recursive* if it has already occurred once in the call-chain from the root. In the presence of a recursive function call `f`, the following steps are taken:

```
main ( )
{
  f ( ) ;
}

f ( )
{
  if ( . . . )
    f ( ) ;
}
```

(a) C program

Func	Func-list
main ( )	f
f ( )	f

(b) 'Func' is the function name and 'Func-list' is the list of functions who get called by 'Func'



(c) The steps of generation of invocation graph from left to right

Figure 4.3: An example of the construction of the recursive invocation graph using our method.

1. Change the first occurrence of `f` from ordinary to recursive node `f-R`.
2. Create an approximate node for the recursive call `f`, denoted as `f-A`, and connect it to the current place (the caller's node in the invocation graph) by a calling arc.
3. Connect the approximate node `f-A` and its corresponding recursive node `f-R` by an *approximate arc*.

In the following, we explain the construction of the invocation graph in the presence of mutual recursion, using the example shown in Figure 4.4(a). Figure 4.4(b) shows all the steps of this construction. The new modifications at each step are shown using bold face notation. Since function `main` calls `f`, we create two ordinary nodes for `main` and `f` and connect them by a calling arc. When `f` calls `g`, we create another ordinary node for `g` and connect it to `f` by a calling arc. Now `g` calls `f` recursively. Since `f` has already occurred once in the call-chain from the root, we do the following: (i) change the first occurrence of `f` to a recursive node denoted as `f-R`, (ii) create an approximate node for `f` denoted as `f-A` and connect `g` and `f-A` by a calling arc, and (iii) connect the approximate node `f-A` and its corresponding recursive node `f-R` by an approximate arc. Approximate arcs are shown by dotted lines in Figure 4.4. Similarly, functions `h` and `w` call themselves recursively. So, we convert the first occurrence of `h` and `w` to `h-R` and `w-R`, create new nodes `h-A` and `w-A`, and connect them by approximate arcs to `h-R` and `w-R`, respectively.

The depth-first algorithm for building invocation graph is given in Appendix B.

Each node in the invocation graph (except `main`) corresponds to a call-site in the

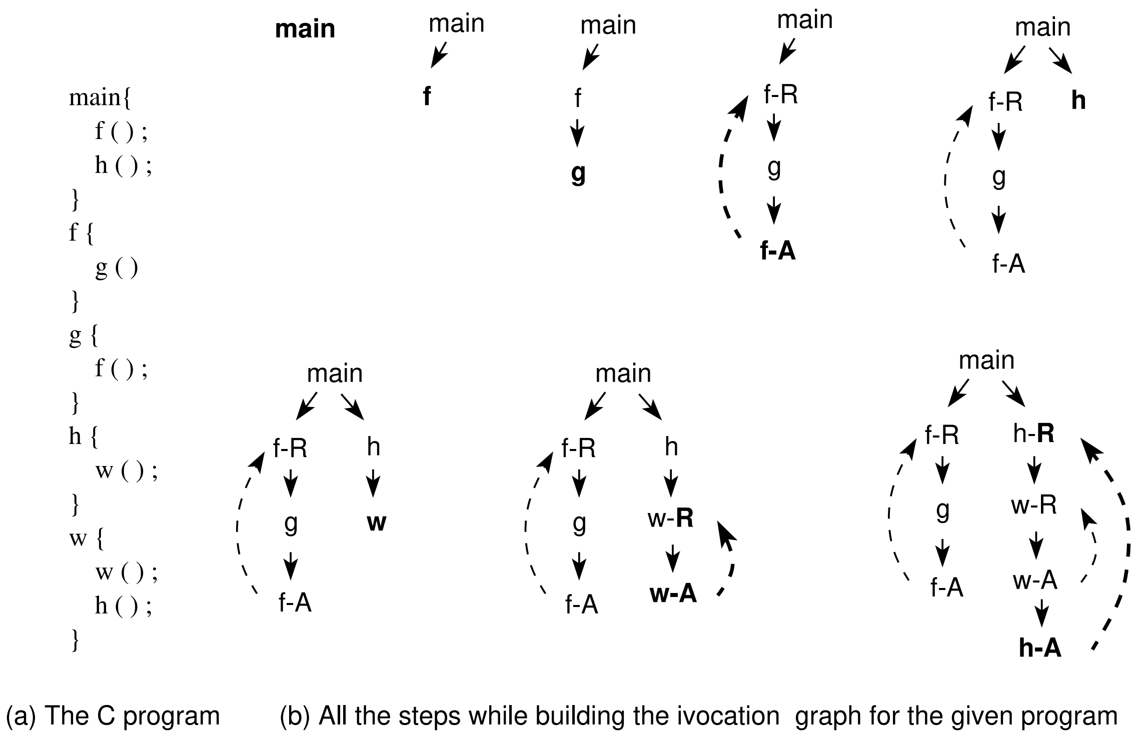


Figure 4.4: An example for construction of the invocation graph in the presence of mutual recursion.

program and each function call in the program corresponds to a unique path in the invocation graph. This fact is shown in the example of Figure 4.5. This example pose the question that how the same function call `g()` (in the body of `f()`) points to two different nodes in the invocation graph. In order to maintain a one-to-one relationship between the paths of invocation graph and the call sites, one should keep track of (i) the invocation graph node related to corresponding caller (for example the `f` nodes from the invocation graph) and (ii) the corresponding call number (for example function `g` is the first function call in the body of function `f`, therefore the call number in one).

In the subsequent section, we explain how the invocation graph is used in inter-procedural analysis.

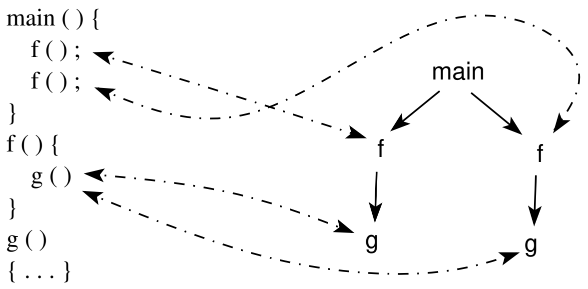


Figure 4.5: Relationship between program and invocation graph.

4.2 Interprocedural Analysis for Non-recursive Function Calls

In this section, we discuss interprocedural analysis in the presence of non-recursive function calls. Interprocedural analysis captures points-to information across functions affected through both function parameters and global variables.

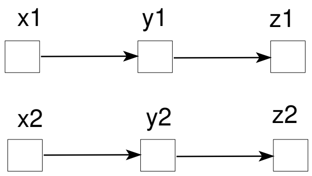
We first give a motivating example to demonstrate the need for the interprocedural analysis. Figure 4.6 represents a program that indirectly swaps two variables. The memory representation of `main` before calling function `indirect_swap`, the memory representation of `indirect_swap` before it is processed, and the memory representation of `main` after the call to `indirect_swap` is processed, are shown in Figure 4.6(a), (b), and (c), respectively. The variable `y1` which points to the location `z1` before the call to the function `indirect_swap` in `main`, points to the location `z2` after this call is completed. This is due to the fact that the memory location `y1` is accessible through the pointer dereference $*$`a` (although `y1` is not in the scope of the function `indirect_swap`). The points-to relationship for the variable `y2` changes in a similar way.

Two important observations can be made here:

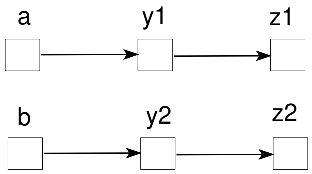
1. A function call can considerably affect the points-to relationships holding in the caller.
2. The program variables, whose points-to information can be modified by a function call, need not lie in the scope of the callee.

The above two possibilities make interprocedural analysis necessary for collecting accurate points-to information.

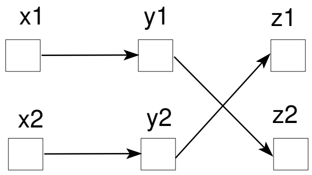
```
int *m ;
main()
{
    int **x1, *y1, z1 ;
    int **x2, *y2, z2 ;

    x1 = &y1 ;
    y1 = &z1 ;

    x2 = &y2 ;
    y2 = &z2 ;

    indirect_swap(x1,x2) ;
}
void indirect_swap(a, b)
int **a, **b ;
{
    int *temp ;

    temp = *a ;
    *a = *b ;
    *b = temp ;
}
```



(a) The memory representation of main() , before function call indirect_swap().



(b) The memory representation of indirect_swap() , before the execution of the function body.



(c) The memory representation of main() , after returning from the call to indirect_swap().

Figure 4.6: A motivating example for interprocedural analysis.

A conservative approach:

In the absence of interprocedural analysis, safe conservative assumptions need to be made about the effects of a function call on the points-to information in the caller. The minimum requirements for any such approximation is to assume that the following types of variables can point to any variable in the scope of the callee:

1. Pointer variables which are global in scope.
2. Pointer variables whose address is passed as a parameter to the callee.
3. Pointer variables accessible through one or more levels of dereference of any variable of the above two types.

As type casting is permitted in C, we have to assume that these pointer variables can point to any variable, irrespective of its type, in the scope of the callee.

Using these assumptions, the memory representation after the call to the function `indirect_swap` is estimated as shown in Figure 4.7. The relationships from/to `m`

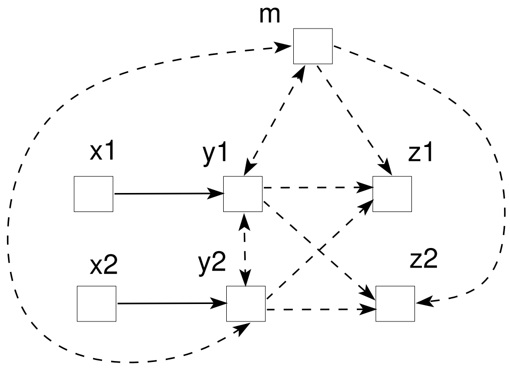


Figure 4.7: The memory representation without the interprocedural analysis based on a conservative assumption instead.

are caused by the fact that `m` is a global variable and therefore can be effected by any function call. The possibly-points-to relationship between the pointer variables of the same level, e.g. (y1,y2,P), can be caused by type casting. Since the address of variables `x1` and `x2` are not accessible in function `indirect_swap`, their points-to information does not change.

We conclude that the conservative approach introduces substantial imprecision in the points-to information. This imprecision accumulates with frequently occurring function calls in C-programs. Thus, the need for interprocedural analysis is evident.

Our approach using the invocation graph:

We have already discussed our representation for the invocation graph which captures the interprocedural structure of the program. The invocation graph is a basic requirement for interprocedural analysis as it makes both the interprocedural program paths and the nature of function calls, explicit. The usage of invocation graph in our analysis is explained in Section 4.2.1.

For interprocedural points-to analysis, we need to propagate points-to information from a call-site to the entry of the corresponding function. As this propagation progresses, we modify the points-to information to take care of the following:

1. The renaming of variables caused by parameter passing.

2. The possibility of accessing a variable inside the callee which is not in the scope of callee (as shown in the indirect swap example).

Next, the body of the function is analyzed using this points-to information. The information computed during this analysis is returned to the call site. Due to the above-mentioned factors, this information has to be modified again. The map and unmap processes, explained in Section 4.2.2, are designed to perform proper propagation of points-to information from the caller to the callee and vice versa.

After simplification, the occurrence of a function call is limited to one of the following cases when each argument, $\arg_i$, is a scalar variable or a constant:

1. $f(\arg_1, \arg_2, ..., \arg_n)$
2. $a = f(\arg_1, \arg_2, ..., \arg_n)$ and `return(var)`

In the following sections, we discuss the first case in detail. The second case, which is obtained by adding some slight modifications to the first case, will be discussed in Section 4.2.3.

4.2.1 Relationship Between Invocation Graph and Interprocedural Analysis

As mentioned before, our interprocedural analysis follows the sequence of function calls. This means that for each function call, analysis proceeds from the call-site to the analysis of the body of the callee and then returns to the call-site.

The relationship between our analysis with invocation graph is explained by studying the example given in Figure 4.8. Analysis of the program starts from the `main` function. For each function, the intraprocedural analysis is done until we reach a function call (in this case `f()`). Then, the analysis proceeds via the invocation graph. If the corresponding invocation graph node is an ordinary node (other kind of nodes will be discussed in Section 4.3), the body of the callee (`f`) is analyzed and the result is returned through the invocation graph to the call-site upon completion. After that, the analysis continues in the body of the caller (`main`) until reaching another call (`f()`). Analysis continues in this fashion until all statements of caller (`main`) have been analyzed. The movement to/from the invocation graph is shown as the labels of the links between the invocation graph and the program in Figure 4.8.

Using this method, we can be sure of the following points:

1. The points-to information coming from different call-sites will not be used at the same time to collect new points-to information.

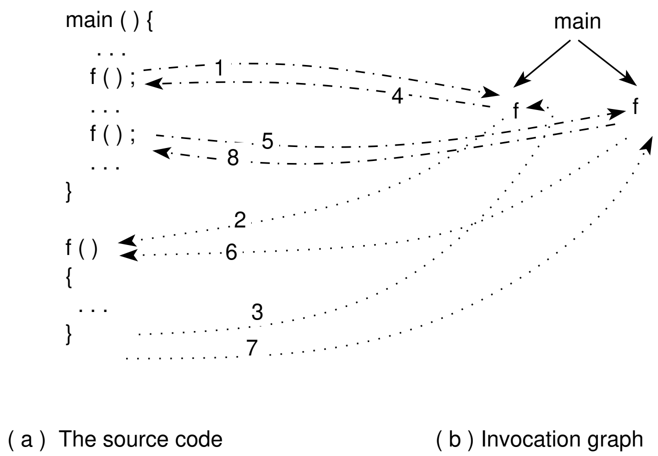


Figure 4.8: Relationship between program and invocation graph (the numbers written on the edges show the order of the process).

2. The points-to information always returns to the appropriate unique call-site.

4.2.2 Map and Unmap Processes

We characterize the interprocedural analysis by the following steps:

- Map process: is a process that compares all the actual and formal parameters and sets all the possible points-to information for formal parameters. It does the same thing for the global variables.
- Function process: is the intraprocedural analysis for the body of the callee. The input to this process is the abstract stack as created by the map process.
- Unmap process: is a process which takes the resulting abstract stack from the function process and unmaps the variable names - changing the names in the callee back to the names in the caller.

Figure 4.9 shows a graph representation of the map and unmap processes.

The interprocedural points-to algorithm is represented in the following:

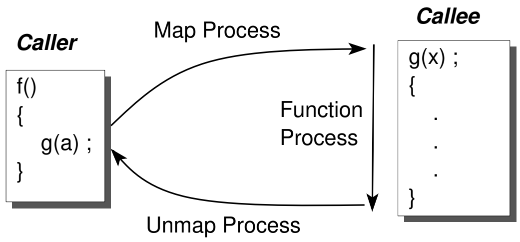


Figure 4.9: The map and unmap process.

```
/* Given a basic statement and input information, return the output
 * information (input.in is the points_to relationships) */
points_to (S,input) =
  if basic_statement(S) then
       /* S is basic statement */
       return(basic_points_to(S,input));
  else if control_statement(S) then
       /* S is a compositional control statement */
       return(control_points_to(S,input));
  else if func_call(S) then
       /* S is a call expression statement */
       return(func_points_to(S,input));
  else
       return(input) ;
```

The algorithm for 'basic_points_to' and 'control_points_to' was given in Chapter 3. The algorithm for 'func_points_to' in the absence of recursive function calls is given in the following (the algorithms for 'map_process' and 'unmap_process' which are used here will be explained later in this chapter). The algorithm of 'func_points_to' in the presence of recursive function calls is given in Section 4.3.

```
/*______________________________________________________________
 * Function name : func_points_to
 * Purpose : find out the points-to information after
 *           processing a non recursive function call, using
 *           the input information to the call.
 * Parameters : call_expr_node -- the call node in caller
 *              in_data -- the input information to
 *                         'call_expr_node'  before it is processed.
 *                         Note that the field '.in' represents the
 *                         points-to relationships.
 * Return : out_data -- the output information of caller
 *                      after processing function call
 *______________________________________________________________*/
func_points_to( call_expr_node, in_data)
{
    /* get the related invocation graph node to the callee by using the
     * invocation graph node of caller ('in_data.cur_ig_node') and
     * related call number of callee  ('call_expr_node.call_expr_num').
     * For example, if 'f(a)' is the third function call in the body of 'g',
     * the related invocation graph node to callee 'f(a)' is the third child
     * of the related invocation graph node of caller 'g'. */
    ig_node = get_related_ig_node( in_data.cur_ig_node,
                                   call_expr_node.call_expr_num) ;

    /* get the function node of the callee from the 'call_expr_node' */
    func_node = get_func_node( call_expr_node) ;

    /* get the list of arguments (actual parameters) */
    arg_lst = get_arg_lst( call_expr_node) ;

    /* 'map_process' maps the points-to information ('in_data.in'), returns
     * the mapped points-to information ('func_in_data.in') and the map
     * information ('map_info') */
    [map_info, func_in_data.in] = map_process( func_node, arg_lst, in_data.in) ;

    /* save the map information and the entry points-to relationships in the
     * invocation graph node related to the function call. */
    save_in_info( ig_node, map_info, func_in_data.in) ;

    /* get the body of the callee */
    func_body = get_func_body( func_node) ;

    /* if the invocation graph node is an ordinary node (non-recursive), analyze
     * the body of callee and return output information in 'func_out_data' */
    if (is_ordinary_node(ig_node))
        func_out_data = points_to( func_body, func_in_data) ;
```

```
    /* save the result of the points-to analysis of callee in the related
     * invocation graph node */
    save_out_info( ig_node, func_out_data.in) ;

    /* the unmap process gets the points-to information of caller
     * and callee and returns the new set of points-to information
     * for caller ('.in' is the field related to points-to info).*/
    out_data.in = unmap_process( in_data, func_out_data.in, map_info) ;

    return( out_data) ;
}
```

As mentioned before, the following two cases complicate the map and the unmap processes.

1. The renaming process between the actual and formal parameters. This is explained in the next two sections.
2. The accessibility of the variables which are not in the scope of the callee (through indirect reference). To solve this problem, we introduce the concept of *Invisible Variables*[1]. These are names used for the variables that are not in the scope of a function but are accessible trough indirect reference. The invisible variables of `x` with type `int**` are `1_x` and `2_x` with types `int*` and `int`, respectively.

The following example explains the use of invisible variables.

```
main(){
    int *a, b, *c, d ;  /* stmt 1 */
    a = &b ;            /* stmt 2 */
    f(a) ;              /* stmt 3 */
    c = &d ;            /* stmt 4 */
    f(c) ;              /* stmt 5 */
}                       /* stmt 6 */
f(x)                    /* stmt 7 */
int *x ;                /* stmt 8 */
{   }
```

After processing statement 2, we get (a,b,D). As the result of the call to function `f` at statement 3, `x` gets a copy of the contents of `a`. This means that `x` should point to

[1] A similar notation of non-visible variables is given in [LR92].

the same location as `a` is pointing to, namely `b`. Even though `b` is not in the scope of function `f`, the memory location represented by `b` is still accessible in the function `f` through the indirect reference $*$`x`. Since we deal with locations of abstract stack, we need a name corresponding to the location `b`. There are two solutions to this problem:

(i) Having a location in the callee for each possible access to a variable even if the variable is not in the scope of the callee. For the above example, we should have abstract stack locations for variables `b` and `d` because they will be used by the calls to `f` at statements 3 and 5.

(ii) Introducing invisible variables and using them instead of the variables which are not available in the scope of callee. In the above example, `x` has only one invisible variable namely `1_x` with type `int`. At the call in statement 3, the invisible variable `1_x` stands for the local variable `b` and leads to the relationship (x,1_x,D) as the result of the map process. At the call in statement 5, the invisible variable `1_x` stands for the local variable `d` and resolves the relationship (x,1_x,D) as the result of the map process.

The following are the advantages of using the invisible variables instead of the local variables of the caller in the abstract stack:

1. Adding the local variables to the abstract stack location of the callee increases the size of the abstract stack. In the above example, without using invisible variables, we would need two locations for `b` and `d` in the abstract stack of `f`. Using the invisible variable `1_x` is sufficient to handle both of these cases (in the first call `1_x` stands for `b` and in the second call it stands for `d`). This results in a more compact representation for the points-to information.
2. The points-to information of each function call can be reused by keeping track of the input-output sets of points-to information of each function. If an input set has already occurred and an output is computed for it, in the second appearance of the same input, we can use the already computed output. The input-output information of each call is saved in its related invocation graph node. In the above example, the input to the function `f` in both cases is the same. Consequently, there is no need to process the second call to `f`, if the output of the first call is already available. On the other hand, as we have two different locations for `b` and `d` in the abstract stack for `f`, it becomes necessary to process both function calls to `f`. Although this could have been possible if a function is called by the same parameters (e.g. if the second call to `f` was `f(a)`), but it can be done more often while using invisible variables.

3. Adding the local variables can result in a problem in the case of recursion. This happens because the number of iterations in a recursive function is not known. In this case, we select a finite number of invisible variables using the type of arguments and global variables.

To make sure that each accessible variable has a location in the abstract stack, we generate names for all the possible locations that each indirection of a formal parameter or global variable may need to access.

A set of relationships between local variables in the caller and invisible variables in the callee is kept for the unmap purposes. This set is called `map_info`. In the presence of possibly-points-to relationship, an invisible variable may stand for more than one variable. The details will be explained in subsequent sections.

In the following sections, we describe: (i) a general algorithm for map process, (ii) a general algorithm for unmap process, and (iii) our accurate algorithm for the map and unmap processes.

Map Process

The idea of the map process was explained in the previous section. In general, one can devise different algorithms for the map process. In this section, we present a general, and straight-forward algorithm for the map process. Later, this is extended to a more accurate method.

Consider the case that the function `f` is called as `f(` $\texttt{arg}_1$`,` $\texttt{arg}_2$`,...,` $\texttt{arg}_n$`)` and is defined as `f(` $\texttt{param}_1$`,` $\texttt{param}_2$`,...,` $\texttt{param}_n$`)`. As in the C-language, parameters are always passed by value, to get the equivalent abstract stack links for each formal parameter $\texttt{param}_i$, we consider the parameters as:

$$\texttt{param}_1 \texttt{ = } \texttt{arg}_1 \texttt{ ;}$$

$$\texttt{param}_2 \texttt{ = } \texttt{arg}_2 \texttt{ ;}$$

$$\ldots$$

$$\texttt{param}_n \texttt{ = } \texttt{arg}_n \texttt{ ;}$$

The rule determining the points-to information is similar to the case for basic statements which was described in Chapter 3. This rule specifies that $\texttt{param}_i$ should point to the same location as $\texttt{arg}_i$ is pointing to. In the present case, we have two additional features for this rule which are as follows:

(i) For multi-level pointers, the rule extends the points-to relationship to each of the corresponding lower level of dereference that is of pointer type. For example, if $\texttt{arg}_i$ is of type `int**`, the rule should also apply to $\texttt{*param}_i \texttt{ = *arg}_i$.

(ii) The invisible variables are used instead of the variables which are not in the scope of callee.

Since a global variable might point to a local variable of caller which is an invisible variable for callee, the map process should be applied to global variables as well. This is achieved by applying the assignment $\texttt{var}_i$ = $\texttt{var}_i$ to all the global variable $\texttt{var}_i$ in the same way as was explained for the function parameter. In this assignment, the global variable $\texttt{var}_i$ appearing in the lhs (left hand side) is related to the scope of the callee and the global variable $\texttt{var}_i$ appearing in rhs (right hand side) is related to the scope of the caller (each function has its own abstract stack).

In the case that an actual parameter or a global variable points to a location which is not in the scope of callee, we use the related invisible variable and set its related location in a set called `map_info` (this set is later used by the unmap process). If the location is already assigned to an invisible variable, we just use that invisible variable. The reason behind not redefining an invisible variable is to avoid a situation where one real stack location is used for two or more different abstract stack locations.

The general algorithm for the map process is as follows:

```
/*__________________________________________________________
 * Function name : map_process
 * Purpose : finding the points-to information of callee,
 *              and the corresponding map information using
 *              the points-to information of caller.
 * Parameters : func_node -- the function declaration node of callee
 *                  arg_lst -- argument list
 *                  caller_in -- the points-to information of caller
 * Return : map_info -- a set of relationship between invisible
 *                          variables of callee and local variables
 *                          of caller.
 *             callee_in -- the points-to information entering to callee
 *__________________________________________________________*/
map_process( func_node, arg_lst, caller_in)
{
    /* initialization */
    map_info = { } ;
    callee_in = { } ;
    param_lst = get_param_lst( func_node) ;

    /* doing the map process for each parameter */
    for each 'param_i' in 'param_lst' and 'arg_i' in 'arg_lst' do
        /* 'callee_in' and 'map_info' will be updated. The argument
         * corresponding to 1 is the depth of the pointer type. For
         * the first call, depth is 1. */
        [map_info, callee_in] =
        map_func( param_i, arg_i, callee_in, caller_in, 1, map_info) ;

    /* doing the map process for each global variable */
    for each global variable 'var_i' do
        /* 'callee_in' and 'map_info' will be updated. The argument
         * corresponding to 1 is the depth of the pointer type. For
         * the first call, depth is 1. */
        [map_info, callee_in] =
        map_func( var_i, var_i, callee_in, caller_in, 1, map_info) ;

    return( [map_info, callee_in]) ;
}
```

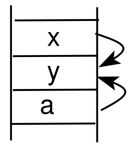
```
/*________________________________________________________________
 * Function name : map_func
 * Purpose : assigning the points-to information of caller_var
 *           to callee_var (considering the invisible
 *           variables). This function is recursively called
 *           to assign the points-to information of all the
 *           indirectly accessible variables through caller_var.
 * Parameters : callee_var -- the variable in callee
 *              caller_var -- the related variable in caller
 *              callee_in -- points-to information of callee
 *              caller_in -- points-to information of caller
 *              depth -- the depth of current invisible variable
 *                       (for the first call, it is 1).
 *              map_info -- the set of map information
 * Return : map_info -- the updated set of map information
 *          callee_in -- the updated set of points-to information
 *                       of callee
 *________________________________________________________________*/
map_func( callee_var, caller_var, callee_in, caller_in, depth ,map_info)
{
    /* if 'callee_var' is not of pointer-type, there is no need of
     * points-to analysis (no extra points-to relationship can be
     * generated) */
    if !(is_pointer_type(callee_var))
        return([map_info, callee_in]) ;

    for each 'x' such that a relationship '(caller_var, x, rel)' exists do
    /* 'rel' can be either definitely-points-to or possibly-points-to */
    {
        if (is_in_callee_scope(x))
        {
            /* add the same type of relationship to 'callee_in' */
            callee_in = callee_in ∪ {(callee_var, x, rel)} ;

            /* recursively check all the relationship of the variables
             * which can be derived from 'caller_var'. In this case,
             * it is all the variables that 'x' points to.
             * As 'x' is one level deeper, depth is increased by one. */
            [map_info, callee_in] =
            map_func( x, x, callee_in, caller_in, depth+1) ;
        }
```

```
        else{ /* variable 'x' is not in the scope of callee */

            /* look at 'map_info' set to check if an invisible variable
             * is already assigned to the variable 'x' */
            if (exist_invisible_for( x, map_info))
                /* if an invisible variable is already assigned to variable
                 * 'x', get that variable and use the same name. */
                x_invisible = get_invisible_var( x, map_info)) ;

            else{ /* an invisible variable does not exist for 'x'*/
                /* get an invisible variable for 'x' using 'depth'
                 * and 'callee_var'. If the name of variable 'callee_var'
                 * is 'data' and 'depth' is 1, the invisible variable
                 * would be '1_data', for 'depth' 2, the invisible variable
                 * would be '2_data', and so on. */
                x_invisible = define_invisible( callee_var, depth) ;

                /* set the equivalency of 'x_invisible' and 'x' in the
                 * 'map_info'. This information is used by the unmap process*/
                add_map_info( x_invisible, x, map_info) ;
            }

            /* add the relationship 'rel' between 'callee_var' and the
             * invisible variable 'x_invisible' in 'callee_in'. */
            callee_in = callee_in ∪ {(callee_var, x_invisible, rel)} ;

            /* recursively check all the relationship of the variables
             * which can be derived from 'caller_var'. In this case,
             * it is all the variables that 'x' points to.
             * As 'x' is one level deeper, depth is increased by one. */
            [map_info, callee_in] =
            map_func( x_invisible, x, callee_in, caller_in, depth+1) ;
        }
    }
    return([map_info, callee_in]) ;
}
```

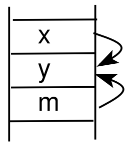
In the following, we further explain the general map process using some examples which are devised to give a clear idea of the concept.

Consider the example given in Figure 4.10. At statement 1, before the process of function `f`, we get {(a, y, D), (x, y, D)} (refer to Figure 4.10(a)). We first apply the map process on the parameters. To do this, we add the points-to relationship resulting from `m = a` to the set of the points-to information of callee. This means that `m` should point to the same location as `a` is pointing to. As the result, we obtain

```
int *x, y ;
main()
{
    int *a ;
    a = &y ;
    x = &y ;
    f(a) ;   /* stmt1 */
}

f(m)
int *m ;     /* stmt2 */
{
    ...
}
```



(a) Abstrack stack of main ( ) at statement 1



(b) Abstrack stack of f ( ) at statement 2

Figure 4.10: A simple example of map process.

the relationship (m,y,D). Next, we perform the map process on global variables. This results in the relationship (x,y,D) for callee. The final result of map process is shown in Figure 4.10(b).

Figure 4.11 shows another example that makes use of the invisible variables. The main program under consideration and its simplified version (obtained by passing through the simplifier program) are shown in parts (a) and (b) of this figure. As already mentioned in Chapter 2, an address operand can never appear as a parameter in the SIMPLE AST. This fact is shown in statement 1 of Figure 4.11(b). In this case, the address operand will be moved out by defining a new variable (in this case `temp0`) and using it as a replacement. At statement 1, before the process of function `f`, we get {(x, c, D), (a, b, D), (temp0, b, D)}. The corresponding abstract stack representation is shown in Figure 4.11(c). At this point, the map process is performed on function parameters and global variables. We first apply the map process to the two function parameters, `m` and `n`. This is described in the following:

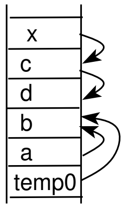
1. We should add the points-to information resolved from `m = a` to the set of the points-to information of callee (`f`). This means that `m` should point to the same location as `a` is pointing to. This results in the relationship (m,b,D). Since variable `b` is not in the scope of function `f`, we do not want to use the name `b` in the analysis of `f`. Rather, at this point, we use the invisible variable `1_m` instead of `b`. We also note in the `map_info` set that `1_m` and `b` are equivalent. As the final result of this step, we get the relationship (m,1_m,D). Since `1_m` is not of

```
int **x ;
main()
{
    int *a, b,
        *c, d ;

    a = &b ;
    x = &c ;
    c = &d ;
    f(a, &b) ;
}

f(m, n)
int *m, *n ;
{
}
```

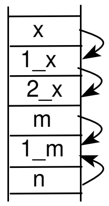
(a) A C-program

```
int **x ;
main()
{
    int *a, b,
        *c, d ;
    int *temp0 ;

    a = &b ;
    x = &c ;
    c = &d ;
    temp0 = &b
    f(a, temp0) ;/*stmt 1*/
}

f(m, n)
int *m, *n ;     /*stmt 2*/
{
}
```

(b) The simplified version



(c) Abstrack stack of main ( ) at statement 1.



(d) Abstrack stack of f ( ) at statement 2.

1_x ⇨ c
2_x ⇨ d
1_m ⇨ b

(e) The map information (map_info).

Figure 4.11: An example of map process where invisible variables are used.

pointer-type, this step is completed.

2. We should add the same points-to information resolved from `n = temp0`. This means that `n` should point to the same location as `temp0` is pointing to. This results in the relationship (n,b,D). Once again, we want to express this relationship in terms of the names in the scope of function `f`. However, in this case, as it is not the first occurrence of `b`, the invisible variable `1_m` already stands for variable `b`. Therefore, we can use `1_m` instead of `b` (there is no need to define another variable). As the final result, we get the relationship (n,1_m,D). Since `1_m` is not of pointer-type, this step is completed.

Now, we apply the map process to the global variable `x` resulting in (x,c,D). In this case, `c` is not in the scope of `f`, and, consequently, it is the first time that `c` is used in the map process. Due to this reason, we use another invisible variable `1_x` and add it to the set of `map_info`. This results in (x,1_x,D). Since `1_x` is of pointer-type (`int*`), the map process continues to a lower level. Because `1_x` is equivalent to `c`, `1_x` points to the same location as `c` is pointing to. The variable `c` definitely-points-to `d` but `d` is not in the scope of function `f` and also is not already assigned to an invisible variable. In this case, another invisible variable has to be used, namely `2_x`. The equivalence of `2_x` and `d` is added to `map_info`. This resolves the relationship (1_x,2_x,D). As `2_x` is not of pointer-type, the process is completed at this point. Figure 4.11(d) shows the abstract stack after the completion of map process and Figure 4.11(e) shows the corresponding `map_info` set.

Unmap Process

In this section, we explain the general unmap process and the use of invisible variables. In the next section, we extend the unmap process to a more accurate method.

As mentioned before, when the map process is completed, the body of the callee will be processed (intraprocedural analysis). After that, the unmap process returns all the changes to the caller. In other words, the unmap process updates the abstract stack of caller according to the changes occurring in the process of callee. The map information (`map_info`) collected from map process is used for this updating purpose. The relationship between local variables of caller and invisible variables of callee are kept in the `map_info` set.

Assume that `caller_input` is the set of points-to information of caller `f` just before the function call to `g`, and `callee_output` is the set of points-to information of callee `g` after the process of `g` is completed. We want to apply the unmap process to the two sets `caller_input` and `callee_output` and thereby compute the `caller_output` set which is the points-to information of `f` after the process of callee `g` is completed. Following is the general algorithm of unmap process:

$$
\begin{array}{l}
\text{unmap\_process(caller\_input, callee\_output, map\_info)} \\
\{ \\
\quad \text{kill} = \{(\text{x, y, rel}) : (\text{x, y, rel}) \in \text{caller\_input} \wedge \\
\qquad\qquad\qquad\qquad (\text{x\_map, y\_map, rel}) \notin \text{callee\_output}\} \\
\\
\quad \text{gen} = \{(\text{x, y, rel}) : (\text{x\_map, y\_map, rel}) \in \text{callee\_output}\} \\
\\
\quad \text{caller\_output} = \text{gen} \cup (\text{caller\_input} - \text{kill}) \\
\\
\quad \textbf{return}(\text{caller\_output}) \\
\}
\end{array}
$$

where x/y is in the scope of caller (`f`) and x_map/y_map is its related variable in the callee (`g`). If x/y is in the scope of the callee (`g`), x_map/y_map is the same as x/y. Otherwise, it can be found from the `map_info` set. Basically, the above procedure removes all the changed information from points-to set of caller and adds the new information to it. The following are some examples to clarify the corresponding algorithm.

Consider the example given in Figure 4.12. As it is shown in Figure 4.12(a), there

```
int *x, y ;
main()
{
    int *a ;
    f() ;    /* stmt 1 */
    a = x ; /* stmt 3 */
}           /* stmt 4 */
f()
{
    x = &y ;
}           /* stmt 2 */
```

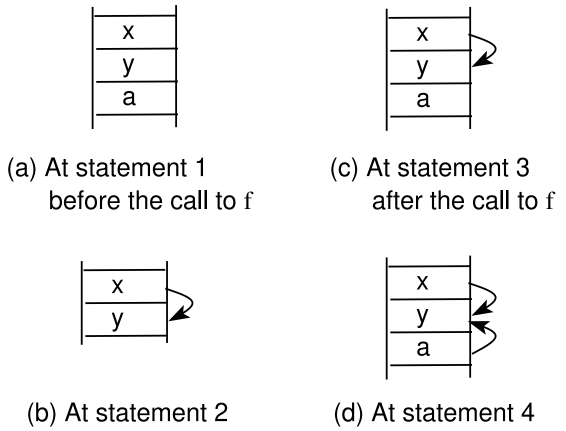


Figure 4.12: An example of unmap process.

is no points-to information that holds before the call to function `f` at statement 1. Due to this fact, the map process does not pass any information to the function `f`.

At statement 2, we get (x,y,D). The corresponding stack representation is shown in Figure 4.12(b). Now is the time for the unmap process which is applied through computing the sets mentioned in the above algorithm. The involved sets are as follows:

caller_input = { }

callee_output = {(x,y,D)}

kill = { }

gen = {(x,y,D)}

caller_output = {(x,y,D)}

Note that since `x` and `y` in the set `caller_input` are in the scope of `main`, their mapped variables do not change. As a result, `gen` set holds the relationship (x,y,D). The result of unmap process is shown in Figure 4.12(c). Finally, after the execution of statement 3, we get (a,y,D) (since `x` definitely-points-to `y`, `a` also definitely-points-to `y`). This is shown in Figure 4.12(d). This example shows that the unmap process is essential in the case that no invisible variable is available. The example in Figure 4.13 shows the importance of the invisible variables in the unmap process.

The points-to information available before the function call `f` at statement 1 is shown in Figure 4.13(a). The result of the map process of function `f` is shown in Figure 4.13(b). The abstract stack representation at statement 3, after processing the body of `f`, is shown in Figure 4.13(c). The map information saved in `map_info` is shown in Figure 4.13(e). The following are the involved sets computed for the unmap process:

caller_input = {(a,b,D), (b,c,D)}

callee_output = {(m,1_m,D), (1_m,x,D), (n,x,D)}

kill = {(b,c,D)}

gen = {(b,x,D)}

caller_output = {(a,b,D), (b,x,D)}

The 'kill' set is {(b,c,D)} because (b,c,D) is in the caller but its related points-to relationship which is (1_m,2_m,D) is not in the callee. The 'gen' set is {(b, x, D)}, because (1_m, x, D) is in callee but its related points-to information in caller which is (b, x, D), is not available. Note that the relation (m,1_m,D) is in callee but, since `m` is not in the scope of `main` and it is not in the `map_info` set, it has no related relationship in the caller. Also, (n,x,D) is in the callee but, since `n` is local to the callee, it has no related relationship in the caller. The final result is shown in Figure 4.13(d).

```
int x ;
main()
{
    int **a, *b, c ;
    a = &b ;
    b = &c ;
    f(a) ;   /* stmt 1 */
}            /* stmt 4 *
f(m)
int **m ;    /* stmt 2 */
{
    int *n ;
    n = &x ;
    *m = &x ;
}            /* stmt 3 */
```

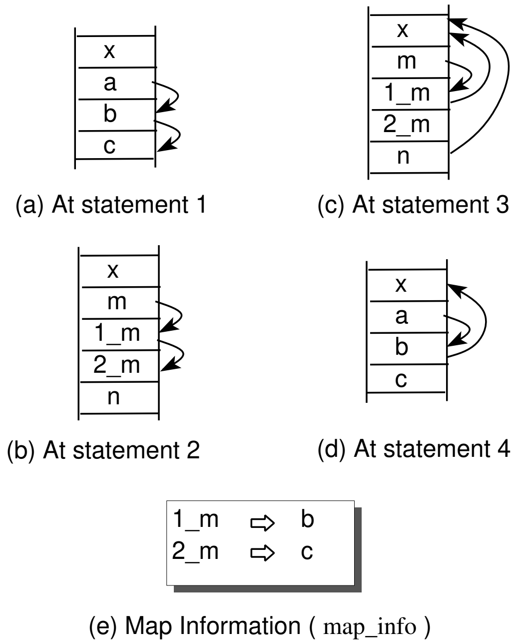


Figure 4.13: An example of unmap process in the presence of invisible variables.

Through this example, we see that variables which are not in the scope of a function may get changed through a pointer variable. One can not take care of this side effect without using the map information (`map_info`).

In the next section, we extend the map and the unmap processes to more accurate methods which are implemented in this thesis.

Extended Map and Unmap Processes

The map/unmap algorithm plays an important role in the accuracy of the interprocedural analysis. We have designed an accurate map/unmap algorithm based on the following two facts: (i) It is safe to say that an invisible variable represents more than one variable. However, since this may result in some extra possibly point-to relationships, we would like to reduce the number of such cases. (ii) Two abstract stack locations are not allowed to represent the same variable and such cases should be avoided.

The following are the major points of these algorithms which will be subsequently discussed in this section:

1. If a variable is a pointer to a structure, it should have a higher priority in the map process. Note that structures are explained in Chapter 5. In the case of having difficulty understanding this part, it might be beneficial to return to this section after reading Chapter 5.
2. In the map process, the definitely-points-to relationships have higher priority than possibly-points-to relationships.
3. In the map and unmap processes, as far as it is possible, definitely-points-to relationships are not changed to possibly-points-to.

The first point concerning our map/unmap process is related to pointers to structures. The goal is to avoid cases that two invisible variables stand for the same location in the abstract stack. When a variable `y` points-to an invisible variable of structure type `1_y` with fields `.f1` and `.f2`, if `1_y` stands for structure variable `a` in the caller, clearly `1_y.f1` stands for `a.f1` and `1_y.f2` stands for `a.f2`. In order to make sure that no other invisible variable is already standing for `a.f1` and `a.f2`, we give a higher priority to pointers to structures. We first assign names for the invisible variables of structure type (at this level we do not assign the points-to information). Then, we apply the map process to all the pointer type variables. In this manner, we guarantee that the double naming for the fields of a structure can not appear. This point is essential for the correctness of the concept of invisible variables. If this point is not considered, two invisible variables can stand for one variable which contradicts the main concept. This point is explained in the example given in Figure 4.14.

At statement 1 of Figure 4.14, if `c` is processed before `a`, we get two invisible variables `1_x` and `1_y.f1` standing for the variable `b.f1`. While in our map process, the naming of variable `a` (a pointer to structure) is applied first. This results in the map information represented in Figure 4.14(e). Then, the map process is applied to `c` and `a`. Variable `c` has a definitely-points-to relationship with `b.f1`. Since name `1_y.f1` stands for `b.f1`, the relationship (x,1_y.f1,D) is resolved and the problem of having another name (`1_x`) for `b.f1` can not appear anymore. The related abstract stack representation and map information for this example are given in Figures 4.14(a) to 4.14(e).

The second point of our map/unmap process is related to the kind of points-to relationship (definitely or possibly). In the map process algorithm explained earlier, there is a potential of changing definitely-points-to relationships to possibly-points-to and/or generating some extra points-to information. This happens when an invisible variable, e.g. `1_x`, stands for more than one variable and a points-to relationship to `1_x` is generated because of a relationship to one of the variables that `1_x` stands

```
typedef struct{
    int f1,
        f2 ;
} FOO ;

int *z ;

main() {
    FOO  *a, b ;
    int *c ;

    a = &b ;
    c = &b.f1 ;
    f(c, a) ;       /* stmt1 */
}                   /* stmt2 */

f(x, y)
int  *x ;           /* stmt3 */
FOO  *y ;
{
    z = &(y->f2) ; /* stmt4 */
}                   /* stmt5 */
```

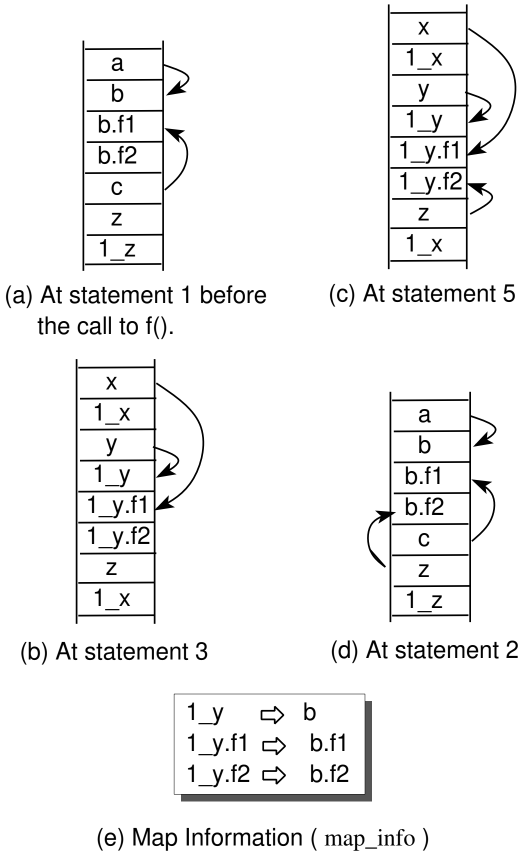


Figure 4.14: An example of our accurate map process when an invisible variable stands for a structure.

for (not to all the variables that `1_x` stands for). An example of this inaccuracy is represented in Figure 4.15.

The points-to information available before the function call `f` at statement 2 is shown in Figure 4.15(a). The result of the map process of function `f` is shown in Figure 4.15(b). Since `b` definitely-points-to `c` and `1_a` represents `c` and `d` (shown in Figure 4.15(e)), the relationship between `b` and `1_a` is established. The fact that the invisible variable `1_a` stands for more than one variable (in this case `c` and `d`) and `b` points-to `1_a` but not to all the variables that `1_a` stands for (in this case `b` points-to `c` and not to `d`), results in some inaccuracy. After the unmap process, the relationships (b,c,P) and (b,d,P) are propagated to the function `main` and replace (b,c,D). The

```
int *a, *b ;
main() {
    int c, d, e ;

    if (...)
        a = &c ;
    else
        a = &d ;
    b = &c ;    /* stmt1 */
    f() ;       /* stmt2 */
}               /* stmt3 */
f()             /* stmt4 */
{
    int *z, w ;

    z = &w ;
}               /* stmt5 */
```

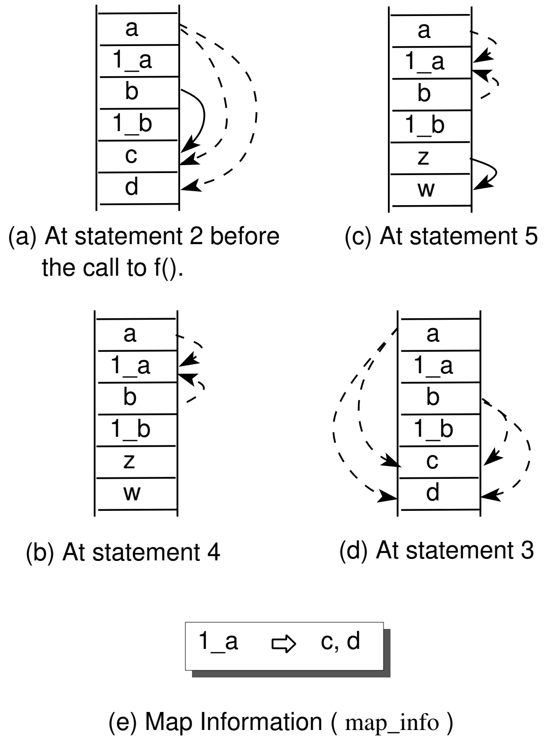


Figure 4.15: An example of general map process when an invisible variable stands for more than one variable.

result of unmap process is represented in Figure 4.15(d). As the result, the definitely-points-to relationship (b,c,D) is replaced by possibly-points-to relationship and an extra triple is generated.

In order to reduce the number of such cases, we associate only one variable to an invisible variable (as long as it is possible). This is achieved in our map process by giving priority to the order of processing the parameters of a function. We first apply the map process to the pointer parameters and global variables which have a definitely-points-to relationship, and then to the pointer parameters and global variables which have possibly-points-to relationship. Using this method on the example of Figure 4.15, the global variable `b` will be processed before `a`. As the result of the map process, `1_b` represents `c`, and `1_a` represents `d`. The relationships after the map process of function `f` are as follows:

$$\{(b,1\_b,D),\ (a,1\_b,P),\ (a,1\_a,P)\}$$

In this case, the unmap process does not change the points-to information of the caller ({(b,c,D), (a,c,P), (a,d,P)}).

This improvement is not always applicable. When the following two cases exist at the same time, one can not avoid the generating of extra triple(s):

1. two invisible variables, `1_a` and `1_b`, stand for more than one variable, and,
2. there is a variable that points-to both of the invisible variables (`1_a` and `1_b`) and not to all the variables that `1_a` and `1_b` stand for.

In the example of Figure 4.15, this case appears by the replacement of statement 1 with the following statements:

```
if (...)
    b = &c ;
else
    b = &e ;
```

The related abstract stacks and map information of this case are represented in Figure 4.16.

Combining the first and second points of our map/unmap process results in the following priority for map process:

1. map naming process for definitely-points-to relationship to structure variables.
2. map naming process for possibly-points-to relationship to structure variables.
3. map process for definitely-points-to relationship.
4. map process for possibly-points-to relationship.

It should be mentioned that for each of these cases all the variables that a parameter or global variable is pointing to are recursively processed to make sure that all the indirectly accessed variables are checked. Therefore, all the relationships related to parameters and global variables are processed four times. Consider the following example:

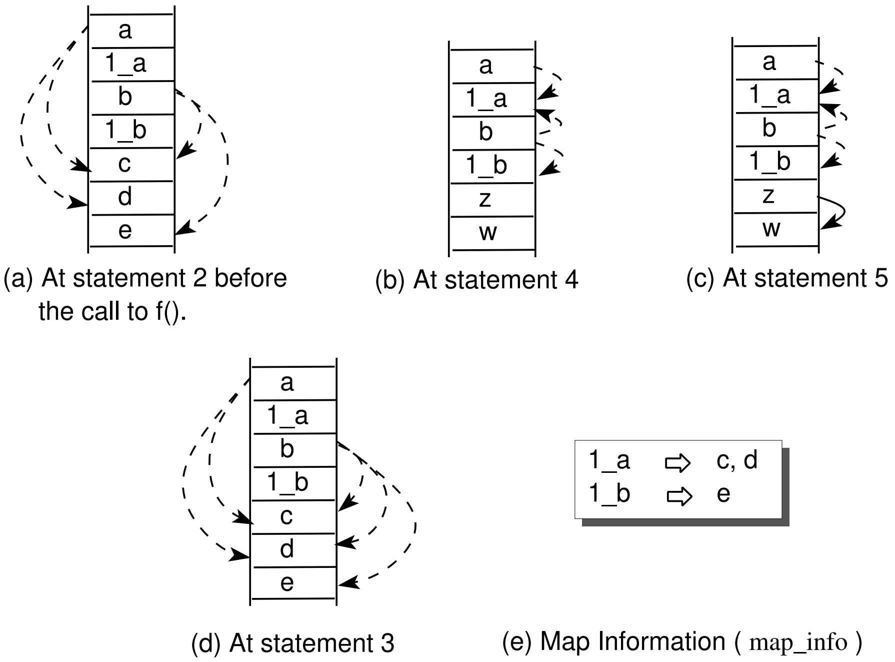


Figure 4.16: An example of our accurate map process when two invisible variables stand for more than one variable.

```
main() {
    int **a, *b, c ;

    if (...)
        a = &b ;
    b = c ;
    f(a) ;    /* stmt1:   {(a,b,P), (b,c,D)} */
}
f(int **x)    /* stmt2:   {(x,1_x,P), (1_x,2_x,D)} */
{
    ...
}
```

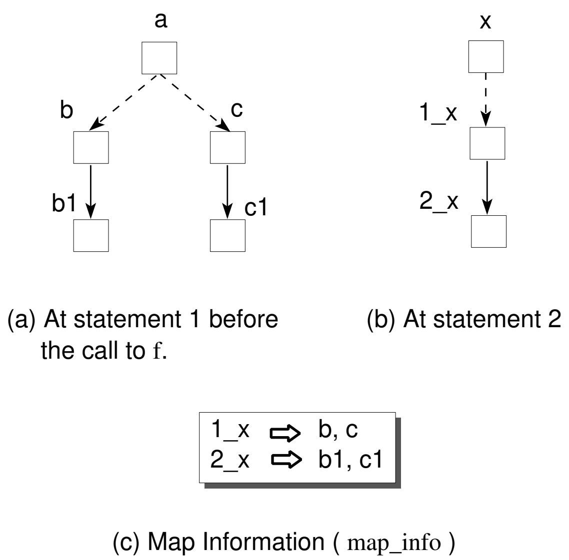
The points-to information before the function call `f` is represented at statement 1. The points-to information after the map process is represented at statement 2. In the third step of our map process, the relationship (1_x,2_x,D) is generated which is indirectly resulting from (a,b,P). In the fourth step of our map process, the relationship (x,1_x,P) is generated which is directly resulting from (a,b,P).

The <u>third</u> point of our map/unmap process has the same goal as the second point which is keeping the definitely-points-to relationships as long as possible. This case is related to multi-level pointers when the following two cases are valid at the same time:

1. An invisible variable `1_x` of pointer type stands for more than one variable, e.g., `a` and `b`, and,
2. `a` and `b` have definitely-points-to relationship with some other variable(s) (e.g., if they both definitely-points-to `c`).

Since a variable (in this case `1_x`) can not have a definitely-points-to relationship with two variables, such a relationship should be replaced by possibly-points-to. This replacement can be avoided when the variables that `1_x` definitely-points-to are represented by a unique variable or by a unique invisible variable. An example of this case is given in Figure 4.17.

```
int *y ;
main() {
    int **a, *b, *c, b1, c1 ;

    if (...)
        a = &b ;
    else
        a = &c ;
    b = &b1 ;
    c = &c1 ;
    f(a) ;       /* stmt1 */
}
f(int **x) {     /* stmt2 */
    y = *x ;
}
```



(a) At statement 1 before the call to f.

(b) At statement 2

(c) Map Information (map_info)

Figure 4.17: An example for the third point of the map process.

The memory representation before/after the map process and the map information are shown in Figure 4.17(a) to (c). Variable `1_x` stands for two variables `b` and `c` which have a definitely-points-to relationship with `b1` and `c1`, respectively. Since `2_x` represents both `b1` and `c1`, we keep the definitely-points-to relationship between `1_x` and 2_x. By applying a proper unmap process, we avoid generating extra information like (b,c1,D) .

Our unmap process does not affect the definitely-points-to information of the caller that already exists in the callee. This is shown in the 'gen' set of the following unmap process.

```
/* our accurate unmap process */
unmap_process( caller_input, callee_input, callee_output, map_info)
{
    kill = {(x, y, rel) : (x, y, rel) ∈ caller_input ∧
                          (x_map, y_map, rel) ∉ callee_output}

    gen = {(x, y, P) : (x_map, y_map, P) ∈ callee_output} ∪
          {(x, y, D) : (x_map, y_map, D) ∈ callee_output ∧
                       (x_map, y_map, D) ∉ callee_input}

    caller_output = gen ∪ (caller_input − kill)

    return(caller_output)
}
```

The precision of this new condition in 'gen' set is obvious when an invisible variable stands for only one variable. Following is an informal proof for the correctness of this new condition in our unmap process when an invisible variable stands for more than one variable: If there is no change in the points-to relationship of an invisible variable, say 1_x, the unmap process is correct (because it will not change the points-to information of caller). Otherwise, since an invisible variable is not directly accessible, the corresponding change has to be done through an indirect reference, say *x. Since 1_x stands for more than one variable, x and 1_x has a possibly-points-to relationship. Therefore, any change of 1_x (*x), changes all its definitely-points-to relationships to possibly-points-to (this is a basic points-to rule, refer to Chapter 3). In the example of Figure 4.17, if *x changes, the relationship (1_x,2_x,D) changes to (1_x,2_x,P). As the result, the newly change in the unmap process will not be applicable. We conclude that the new changes of the unmap process are only for the cases when the definitely-points-to information is not changed in the callee.

The algorithm for the precise map process is given in Appendix B.

4.2.3 Return Statement

All our previous discussions concerning function calls were based on the form `f( var`$_1$`, var`$_2$`, etc.)`. In this section, we discuss the form `x = f( var`$_1$`, var`$_2$`, etc.)`. This form happens along with at least one occurrence of `return` statement in the body of function `f` (e.g. `return(var)`).

For each function `f` returning a pointer-type variable, we define a global variable `return_f` with the same type as `f`. Using this newly defined variable, we treat (the C-program is not changed) `return(var)` as:

```
return_f = var ;
return ;
```

and we treat `x = f( var`$_1$`, var`$_2$`,..., var`$_n$`)` as:

```
f( var1, var2,..., varn) ;
x = return_f ;
```

The above lines are processed in the usual way as described before (note that `x` can be replaced by `*x`). The `return` statement is similar to `break` statement in the sense that it returns the control to the caller. To process the `return` statement, we use the structure `return-list` which is a list of points-to information reaching `return` statements. The following should be done in the presence of a `return` statement:

- Merge the points-to information entering to a given `return` statement with the corresponding `return-list`.
- Pass BOTTOM as the output. In this case, BOTTOM means that the statements after `return` does not contain any valid information. In other words, they are dead code.

At the end of each function, we merge the points-to set reaching the end of function with the information saved in `return-list`. This would be the merge of all potential outputs.

Figure 4.18 shows the corresponding algorithm together with a graph representation of `return` statement.

Figure 4.19 illustrates an example of `return` statement. Figures 4.19(a) and (b) are the main program and its simplified version. The simplifier replaces the address operand occurred in a `return` statement by a newly defined variable, in this case `temp0` and `temp1` at statements 4 and 8. Figure 4.19(c) to (n) shows the abstract

```
process_return(in) {
    result_list = merge_info(result_list, in) ;
    return(BOTTOM) ;
}
```

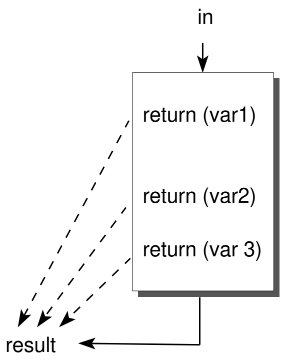


Figure 4.18: The algorithm for `return` statement.

stack corresponding to each statement during our analysis. We explain some of these cases in detail. After the execution of the statement 4, location `f_return` points to the same location as `temp0` is pointing to (Figure 4.19(f)). This information is saved in `return-list` and BOTTOM is returned (Figure 4.19(g)). The information entering statement 6 is the same as the information entering statement 2 (Figure 4.19(h)). The same process as explained above is repeated for the `return` appearing at statement 8. This is composed of saving the information in `return-list` and passing BOTTOM. Statement 10 contains the merge of the information at statements 5 and 9. When the process of the function is completed, we merge the final result (Figure 4.19(k)) with the points-to information in `return-list` (Figures 4.19(f) and (j)). This results in the points-to information shown in (Figure 4.19(l)). Now we treat statement:

```
a = f() ; /* stmt12 */
```

as if it were composed of the following two statements:

```
f() ;              /* stmt12a */
a = return_f ;     /* stmt12b */
```

The result after unmaping of the function `f` is shown in Figure 4.19(m). The last step is the process of statement 12b. When the process of statement 12b is completed, as the points-to information of `return_f` has no further use, it will be removed. The final result is shown in (Figure 4.19(n)).

```
int x, y ;
main()
{
    int *a ;
    a = f() ;
}
int *f()
{
    if (x == 1)
        return(&y) ;

    else
        return(&x) ;

}
```

(a) A C-program

```
int x, y ;
main()
{
    int *a ;                /* stmt 1  */
    a = f() ;               /* stmt 12 */
}
int *f()
{
    int *temp0, *temp1 ;

    if (x == 1)             /* stmt 2  */
    {
        temp0 = &y ;        /* stmt 3  */
        return(temp0) ;     /* stmt 4  */
    }                       /* stmt 5  */
    else{                   /* stmt 6  */
        temp1 = &x ;        /* stmt 7  */
        return(temp1) ;     /* stmt 8  */
    }                       /* stmt 9  */
                            /* stmt 10 */
}                           /* stmt 11 */
```

(b) The simplified program

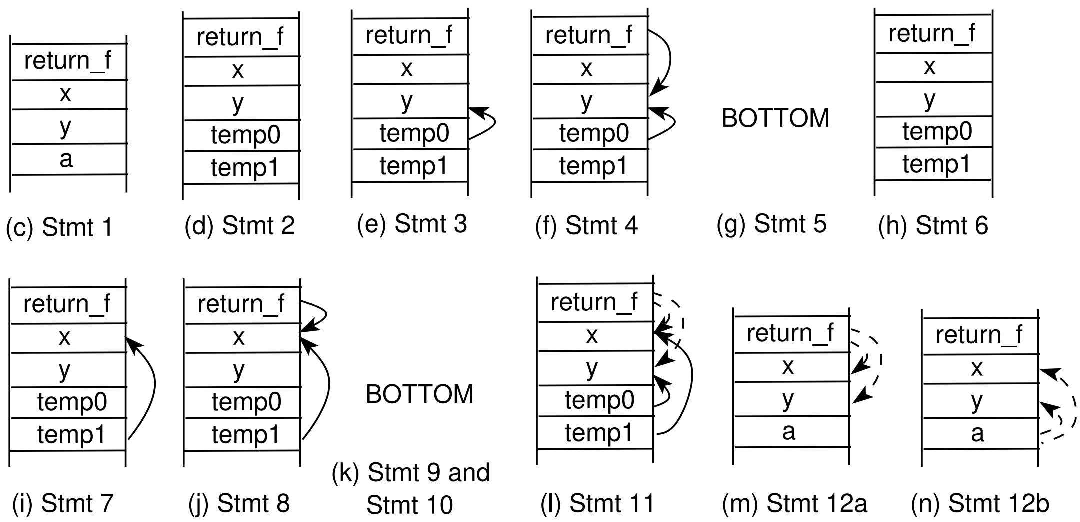


Figure 4.19: An example of `return` statement.

4.3 Interprocedural Analysis for Recursive Function Calls

Now that the process of non-recursive function calls has been completed, we discuss the processing of recursive function calls. The recursive calls are more complicated because the number of iterations is not known.

The difference between recursive and non-recursive calls in our analysis is reflected in the corresponding invocation graphs. Recall from Section 4.1 that the invocation graph is composed of three different kinds of nodes, ordinary nodes, recursive nodes, and approximate nodes; whereas edges could be either calling edges or approximate edges. If the invocation graph node corresponding to a function call is an ordinary node, we apply the map process, analyze the body of the callee, merge the output with the `return-list`, and perform the unmap process, as explained in Section 4.2.

If the invocation graph node corresponding to a function call is a recursive node, a fixed-point has to be found. A fixed-point for a recursive node is reached when: (i) there is only one input to the recursive node (otherwise the inputs are merged and the process of the body restarted), (ii) the input does not generate a different output from the previous iteration to the recursive function. This is the same idea as the fixed-point for `while` loop statement.

Unlike recursive and ordinary nodes, an approximate node does not cause a process to the body of the function. It just approximates the output. If the input (`in`) to an approximate node (`f`) is already entered to the recursive node, it means that this input is already processed, therefore, the related output will be taken, otherwise, `in` is a new possible input to `f` implying that `f` should be processed accordingly.

The process of recursive calls does not make any difference in map and unmap processes. The only difference is in the way that we treat recursive/approximate node in the invocation graph (every call has to pass through invocation graph).

Now that the idea behind our approach is explained, in the following we explain our algorithm.

If the call node corresponding to the function call is a recursive node, we have to iterate until a fixed-point is reached. We have a list of input-output pairs of information stored in a recursive node. These pairs correspond to all the different types of inputs and outputs which can possibly occur during one iteration of the fixed-point calculation. In each iteration, we check if there is more than one input to the node under consideration. If the answer is positive, then we conclude that a recursive call is found with an input not included in the current approximation of input. In this case, we merge all these inputs, store this newly merged input in the tree, and start again with the newly merged input. If there is a single input-output pair left, and that output and the newly created output from the most recent iteration are not the same (fixed-point is not reached), we merge the two outputs, store the newly merged output in the tree, and start again with the input available in the input-output pair. Finally, when a fixed-point is reached, there is a single input-output pair which corresponds to a superset of all the input-output pairs possible for this function. Intuitively, this pair summarizes all the possible input-output pairs of the invocation graph.

If the call node corresponding to the function call is an approximate node, we find its corresponding recursive node in the invocation graph. We check the list of input-output pairs to see if an output exists for this particular input. If it does, we just return this output. Otherwise, we store this new input in the recursive node and return `BOTTOM`. Here, `BOTTOM` has essentially the same properties as the `BOTTOM` described in the previous chapter. In specific, it means that we still do not know the output for this particular input.

General algorithm for the appearance of a call expression is as follows:

```
/*______________________________________________________
 * Function name : func_points_to
 * Purpose : find out the points-to information after
 *           processing function call (that can be a recursive call),
 *           using the input information to the call.
 * Parameters : call_expr_node -- the call node in caller
 *              in_data -- the input information to
 *                         'call_expr_node'  before it is processed.
 *                         Note that the field '.in' represents the
 *                         points-to relationships.
 * Return : out_data -- the output information of caller
 *                      after processing function call
 *______________________________________________________*/
func_points_to( call_expr_node, in_data)
{
    /* get the related invocation graph node to the callee by using the
     * invocation graph node of caller ('in_data.cur_ig_node') and
     * related call number of callee  ('call_expr_node.call_expr_num').
     * For example if 'f(a)' is the third function call in the body of 'g',
     * the related invocation graph node to callee 'f(a)' is the third child
     * of the related invocation graph node of caller 'g'. */
    ig_node = get_related_ig_node( in_data.cur_ig_node,
                                   call_expr_node.call_expr_num) ;

    /* get the function node of the callee from the 'call_expr_node'. */
    func_node = get_func_node( call_expr_node) ;

    /* get the list of arguments (actual parameters). */
    arg_lst = get_arg_lst( call_expr_node) ;

    /* 'map_process' maps the points-to information ('in_data.in'), returns
     * the mapped points-to information ('func_in_data.in') and the map
     * information ('map_info'). */
    [map_info, func_in_data.in] = map_process( func_node, arg_lst, in_data.info)

    /* save the map information and the entry points-to relationships in the
     * invocation graph node related to the function call (the output points-to
     * relationships is saved as BOTTOM). */
    save_in_info( ig_node, map_info, func_in_data.in) ;

    /* get the body of the callee. */
    func_body = get_func_body( func_node) ;
```

```
switch (get_mode( ig_node)){ /* check for the mode of the invocation graph node. */
    case "ordinary":
        /* if the invocation graph node is ordinary , analyze the body of callee. */
        func_out_data = points_to( func_body, func_in_data) ;
        /* merge the result with all the possible output from return statement. */
        func_out_data = merge_info( func_out_data, return_lst) ;
        break ;
    case "approximate":
        /* get the recursive node to the approximate node, by following
         * the approximate arc. */
        recursive_node = ig_graph.approximate_arc ;
        /* search for 'func_in_data.in' in 'recursive_node'. If it is
         * found (if 'func_in_data.in' is a subset of already exist
         * input), return the related output. Otherwise return BOTTOM
         * and save 'func_in_data.in' in the 'recursive_node' as a new
         * possible entry. */
        func_out_data.in = search_info( recursive_node, func_in_data.in) ;
        break ;
    case "recursive":
        /* process the body of the callee. */
        func_out_data = points_to( func_body, func_in_data) ;
        /* merge the result with all the possible output from return statement. */
        func_out_data = merge_info( func_out_data, return_lst) ;
        /* the points-to information from the previous iteration is BOTTOM
         * (because no previous iteration exist for the first iteration). */
        ig_output = BOTTOM ;
        /* check for the fixed-point. A fixed-point is reached when there
         * is one entry to the invocation graph node and for a given input,
         * the output is not changed compare to the previous iteration. */
        while (( num_inputs_in( ig_node) > 1) ||
                ( ig_output not contained in func_out_data.in))
        {
            if ( num_inputs_in( ig_node)) > 1) {
                /* there is more than one input. Merge all the inputs and
                 * save the result in 'func_in_data.in', and in the
                 * 'ig_node' as the next input. */
                func_in_data.in = ig_lst_merge( get_inputs_from( ig_node)) ;
                /* save new input-output (func_in_data.in,BOTTOM) in the ig_node */
                save_ig_node( ig_node, func_in_data.in, BOTTOM) ;
            }
            else
                /* there is a different output compare to the previous
                 * iteration. So merge them. */
                ig_output = Merge( func_out_data.in, ig_output) ;
            /* process the body for the next iteration */
            func_out_data = points_to( func_body, func_in_data) ;
            func_out_data = merge_info( func_out_data, return_lst) ;
        }
}
```

```
    /* save the points-to result of analysis of callee, into related
     * invocation graph node */
    save_out_info( ig_node, func_out_data.in) ;

    /* the unmap process get the points-to information of caller
     * and callee and returns the new set of points-to information
     * for caller ('.in' is the field related to points-to info).*/
    out_data.in = unmap_process( in_data, func_out_data.in, map_info) ;

    return( out_data) ;
}
```

An example is given in Figure 4.20. The process starts from the `main` function. The intraprocedural analysis is done until reaching the function call `f(a)` which invokes the map process. The set of points-to information is saved in the related node in the invocation graph (`f-R`). This is shown in Figure 4.20(b). The information available at statement 1 is shown in Figure 4.20(c). The recursive call `f(n)` invokes the map process again. The information after the map process is saved in the related invocation graph node (f-A) as shown in Figure 4.20(d). The 'In' set of `f-R` is searched for the 'input' set of `f-A`. Since it is not found, the 'input' set will be saved in the 'In' set of f-R (this is considered as a possibly new input), and BOTTOM will be returned. Figure 4.20(e) shows the result after unmaping. The process will be continued by following statements 2 and 3. Since the 'In' set of `f-R` has more than one set, we merge all the elements in 'In' set of `f-R`. The processing of function `f` is restarted with the result of the merge as a new input (refer to Figure 4.20(f)). The intraprocedural analysis is continued until reaching the recursive call to `f(n)`. Set {(x,y,D), (m,y,D)} is the result of map process. A more general case of this set (which is {(a, y,D), (m,y,P)}), is available in the 'In' set of `f-R`. Therefore, we return the related output which in this case is BOTTOM. The process continues by statements 2 and 3. The points-to information reaching the end of the function `f`, which is denoted as 'output', is shown in Figure 4.20(g). At this iteration, there is only one element left in the 'In' set. But, since the 'output' set from Figure 4.20(g) is different from the 'Out' set of `f-R` (which is BOTTOM), we assign 'output' to the 'Out' set of `f-R` as the new output. Then, we start the third iteration. Figure 4.20(h) represents the information at this stage. The same process, as explained for other iterations, has to be done once again. The final result is shown in Figure 4.20(i). There is only one set in the 'In' set, and the 'output' set is same as the 'Out' set. Consequently, the fixed-point is reached and 'output' set is returned as the result of the recursive function `f()`.

```
int *x, y ;
f (m)
int *m ;
{
   int *n ;
   if (y == 1)
   {
       n = &y ;
       /*stmt1*/
       f (n) ;
       /*stmt2*/
   }
} /*stmt3*/
main ( ) {
   int *a ;
   x = &y ;
   f (a) ;
}
```

(a) The C program

main
f - R   In = {{(x, y, D)}}
        Out = BOTTOM
f - A

output = {(x, y, D)}

(b) first iteration

main
f - R   In = {{(x, y, D)}}
        Out = BOTTOM
f - A

output = {(x, y, D), (n, y, D)}

(c) at stmt 1

main
f - R   In = {{(x, y, D)}}
        Out = BOTTOM
f - A   In = {{(x, y, D), (m, y, D)}}
        Out = BOTTOM

output = {(x, y, D), (m, y, D)}

(d) after map process

main
f - R   In = {{(x, y, D)}, {(x, y, D), (m,y,D)}}
        Out = BOTTOM
f - A   In = {{(x, y, D), (m, y, D)}}
        Out = BOTTOM

output = BOTTOM

(e) after unmap process

main
f - R   In = {{(x, y, D), (m, y, P)}}
        Out = BOTTOM
f - A   In = {{(x, y, D), (m, y, D)}}
        Out = BOTTOM

output = BOTTOM

(f) second iteration

main
f - R   In = {{(x, y, D), (m, y, P)}}
        Out = BOTTOM
f - A   In = {{(x, y, D), (m, y, D)}}
        Out = BOTTOM

output = {(x, y, D), (m, y, P)}

(g) at stmt 3

main
f - R   In = {{(x, y, D), (m, y, P)}}
        Out = {(x, y, D), (m, y, P)}
f - A   In = {{(x, y, D), (m, y, D)}}
        Out = BOTTOM

output = {(x, y, D), (m, y, P)}

(h) third iteration

main
f - R   In = {{(x, y, D), (m, y, P)}}
        Out = {(x, y, D), (m, y, P)}
f - A   In = {{(x, y, D), (m, y, D)}}
        Out = {(x, y, D), (m, y, P)}

output = {(x, y, D), (m, y, P)}

(i) The fix-point information

Figure 4.20: An example of the interprocedural points-to analysis with recursive function call.