Appendix B

The Interprocedure Algorithms

This appendix is devoted to: (i) the depth-first algorithm for building invocation graph, and, (ii) the precise map process algorithm. These are given in the following:

```
/*________________________________________________________
 * Function name : build_invocation_graph
 * Purpose : building an invocation graph for the program.
 * Parameters : main_func -- the node related to 'main' function
 *                           in the program
 * Return : init_ig_node -- the root of invocation graph
 *________________________________________________________*/
build_invocation_graph( main_func)
{
    /* build the first invocation graph node related to the 'main' function,
     * which, at this point, has neither parents nor children */
    init_ig_node = make_ig_node ( null, main_func) ;

    /* get the list of the functions that get called by the 'main' */
    init_func_lst = get_func_lst( main_func) ;

    /* generate the rest of the invocation graph */
    gen_ig( init_ig_node, init_func_lst)) ;

    return( init_ig_node) ;
}
/*________________________________________________________
 * Function name : gen_ig
 * Purpose : recursively build the invocation graph, in depth first
 *           manner.
 * Parameters : ig_node -- the current invocation graph node
 *              func_lst -- the list of functions called by the
 *                          related function to 'ig_node'
 *________________________________________________________*/
gen_ig( ig_node, func_lst)
{
    for each function call 'func' in the 'func_lst' do
    {
        /* make a new invocation graph node related to function 'func',
         * and assign 'ig_node' as its parent */
        new_ig_node = make_ig_node( ig_node, func) ;

        /* getting the list of functions that 'func' is calling */
        new_func_lst = get_func_lst( func) ;

        /* call the same function recursively to build the invocation
         * graph in the depth-first manner */
        gen_ig( new_ig_node, new_func_lst) ;
    }
}
```

```
/*______________________________________________________________
 * Function name : make_ig_node
 * Purpose : allocate an invocation graph node corresponding to 'func_call'
 *           and define 'parent_ig_node' as the parent.
 * Parameters : parent_ig_node -- the parent node of newly generated
 *                                invocation graph node.
 *              func_call -- the function call related to the newly
 *                           generated invocation graph node.
 * Return : new_ig_node -- a newly generated invocation graph node
 *______________________________________________________________*/
make_ig_node( parent_ig_node, func_call)
{
    /* assign the related function call to 'new_ig_node' */
    new_ig_node.func = func_call ;

    /* assign the parent (for the root 'parent' would be null). */
    new_ig_node.parent = parent_ig_node ;

    /* add the newly generated node as the child of the parent-node
     * (if the 'parent_ig_node' is not the root).
     * Note: the invocation graph is an invocation tree with some
     *   backward edges. For the efficiency purpose, an invocation tree
     *   is implemented as a binary tree, each node has one child and
     *   one sibling. The sibling of a node has the same parent as
     *   the node. Following shows an example of this representation:
     *          main                             main
     *          / | \        -- represented as -->  /
     *         f  g  h                          f__g__h
     * It is only for the ease of understanding that we
     * do not limit the number of children of a node. */
    if ( parent_ig_node != null)
        add_child( parent_ig_node, new_ig_node) ;

    /* check if the function call 'func_call' is a recursive call. */
    recursive_node = get_recursive_func( new_ig_node) ;

    if (recursive_node == NULL)
        new_ig_node.mode = "ordinary" ;
    else{
        /* the func call is a recursive-call */
        new_ig_node.mode = "approximate" ;
        recursive_node.mode = "recursive" ;
        new_ig_node.approximate_edge = recursive_node ;
    }
    return (new_ig_node) ;
}
```

```
/*________________________________________________________________
 * Function name : get_recursive_func
 * Purpose : check if the given function call related to 'cg_node',
 *              is a recursive call (it already occurs in the path back
 *              to the root of invocation graph).
 * Parameters : cg-node -- current invocation graph node
 * Return : NULL is returned if the related function to 'cg_node' does not
 *             occur in the path back to the root of the invocation graph,
 *             otherwise, the related invocation graph node is returned.
 *________________________________________________________________*/
get_recursive_func( cg_node)
{
    /* get the function call related to the 'cg_node' */
    func = cg_node.func ;

    /* check all the path back to the root of invocation graph,
     * to check if the same function call has already occurred */
    while (cg_node.parent != NULL){
        cg_node = cg_node.parent ;
        if (cg_node.func == func)
            return (cg_node) ;
    }

    return (NULL) ;
}
```

```
/*________________________________________________________________
 * Function name : map_process
 * Purpose : our accurate map process
 * 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
 * Note: throughout this algorithm we use D for definitely-points-to and
 *       P for possibly-points-to relationship.
 *________________________________________________________________*/
map_process( func_node, arg_lst, caller_in)
{
    /* initialization */
    map_info = { } ;
    callee_in = { } ;
    map_func_ptr = 0 ;  /* a function pointer */

    /* the accurate map process is done in four stages. 'map_func_ptr' is
     * pointing to the related function that has to be called in each stage */

    /* first step: giving names to invisible variables of structure type
     * accessible through D relationships. */
    map_func_ptr = &map_naming_definitely_struct_process ;
    [map_info, callee_in] =
    map_step( func_node, arg_lst, caller_in, map_info, callee_in, map_func_ptr) ;

    /* second step: giving names to invisible variables of structure type
     * accessible through P relationships. */
    map_func_ptr = &map_naming_possibly_struct_process ;
    [map_info, callee_in] =
    map_step( func_node, arg_lst, caller_in, map_info, callee_in, map_func_ptr) ;

    /* third step: giving names to invisible variables accessible through
     * D relationships, and setting all the D relationships of callee. */
    map_func_ptr = &map_definitely_process ;
    [map_info, callee_in] =
    map_step( func_node, arg_lst, caller_in, map_info, callee_in, map_func_ptr) ;

    /* fourth step: giving names to invisible variables accessible through
     * P relationships, and setting all the P relationships of callee. */
    map_func_ptr = &map_possibly_process ;
    [map_info, callee_in] =
    map_step( func_node, arg_lst, caller_in, map_info, callee_in, map_func_ptr) ;

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

```
/*_____________________________________________________________
 * Function name : map_step
 * Purpose : this is an intermediate step for the map process to apply
 *           different steps of the map process.
 * Parameters : func_node -- the function declaration node of callee
 *              arg_lst -- argument list
 *              caller_in -- the points-to information of caller
 *              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_func_ptr -- a pointer to the function that has to
 *                              be called in the map_step
 * Return : map_info -- the updated version
 *          callee_in -- the updated version
 *_____________________________________________________________*/
map_step( func_node, arg_lst, caller_in, map_func_ptr)
{
    param_lst = get_param_lst( func_node) ;

    /* doing the map process for each pointer type parameter */
    for each 'param_i' in 'param_lst' and 'arg_i' in 'arg_lst' of pointer type 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_ptr( param_i, arg_i, callee_in, caller_in, 1, map_info) ;

    /* doing the map process for each global variable of pointer type */
    for each global variable 'var_i' of pointer type 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_ptr( var_i, var_i, callee_in, caller_in, 1, map_info) ;

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

```
/*_______________________________________________________________
 * Function name : map_definitely_process
 * Purpose : map the invisible names and the points-to relationships
 *           resolved from the D relationship(s) of 'caller_var'.
 *           This function is recursively called to assign the D
 *           relationships of all the indirectly accessible variables
 *           through 'caller_var'.
 * Purpose : map the invisible names and the D relationships
 *           resolved from 'caller_var'. This function is recursively
 *           called to assign the D information of all the
 *           indirectly accessible variables through caller_var.
 *           It also updates the map information.
 * 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_definitely_process( callee_var, caller_var, callee_in,
                        caller_in, depth ,map_info)
{
    for each 'x_caller' that relationship '(caller_var, x_caller, rel)' exist do
    /* 'rel' can be either D or P */
    {
        /* check if a D relationship with a variable exists */
        if (caller_var, x_caller, D)
        {
            if (depth > 1)
                /* only in the first call 'depth' is 1. Since parameters in C
                 * are passed by value, they should not be mapped to each other.
                 * When 'depth' is more than 1, variables are resolved from
                 * a parameter and should be mapped. */
                add_map_info( caller_var, callee_var, map_info) ;

            if (is_in_callee_scope(x_caller))
                x_callee = x_caller ;
            else{ /* variable 'x_caller' 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_caller' */
        if (exist_invisible_for( x_caller, map_info))
            /* if an invisible variable is already assigned to variable
             * 'x_caller', get that variable and use the same name for
             * the invisible variable. */
            x_invisible = get_invisible_var( x_caller, map_info)) ;

        else{ /* an invisible variable does not exit for 'x_caller'*/
            /* get an invisible variable for 'x_caller' 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 etc. */
            x_invisible = define_invisible( callee_var, depth) ;

            /* set the equivalency of 'x_invisible' and 'x_caller' in
             * 'map_info'. This information is used by unmap process*/
            add_map_info( x_invisible, x_caller, map_info) ;
        }
        x_callee = x_invisible ;
    }
    /* add a relationship between 'callee_var' and 'x_callee' in
     * 'callee_in'. Since this function is a recursive function,
     * 'callee_var' could be an invisible variable that already has
     * some other relationship. Therefore, we can not simply assign
     * the relationship (callee_var, x_callee, D). */

    /* check if 'callee_var' has a D relationship with a variable
     * except 'x_callee'. */
    if (exist_another_definitely_points_to( callee_in, callee_var, x_callee))
    {
        /* This case appears when 'callee_var' is an invisible
         * variable (which can not happen in the first call to this
         * function) and stands for more than one variable, e.g. 'a'
         * and 'b', that one of them has a D relationship with a
         * variable other than 'x_callee'. Since D should be always
         * unique, we should change other D relationships to P
         * relationship and assign a P relationship between
         * 'callee_var' and 'x_callee'. */
        change_D_relation_to_P_relation( callee_in, callee_var) ;
        callee_in = callee_in ∪ {(callee_var, x_callee, P)} ;
    }
    else
        callee_in = callee_in ∪ {(callee_var, x_callee, D)} ;
}
```

```
        /* recursively check all the relationships of the variables which can
         * be derived from 'caller_var'. In this case, 'x_caller' is the first
         * of such variables.
         * As 'x_caller' is one level deeper, depth is increased by one. */
        [map_info, callee_in] =
        map_definitely_process( x_callee, x_caller, callee_in, caller_in, depth+1) ;

        if (is_struct_type( x_caller))
            /* if 'x_caller' is of structure type, all its fields has to be
             * processed, because they might be of pointer type. */
            for each field 'fi' of 'x_caller'
            {
                x_callee = get_related_callee_variable(x_caller, caller_in,
                                                       callee_in, map_info, fi) ;

                /* 1 represents the depth. 1 is chosen because the depth of an
                 * invisible variable related to a field should be started
                 * from one. */
                [map_info, callee_in] =
                map_definitely_process( x_callee, x_caller, callee_in, caller_in, 1) ;
            }
    }
    return([map_info, callee_in]) ;
}
```

```
/*______________________________________________________________
 * Function name : map_possibly_process
 * Purpose : map the invisible names and the points-to relationships
 *           resolved from the P relationship(s) of 'caller_var'.
 *           This function is recursively called to assign the P
 *           relationships of all the indirectly accessible variables
 *           through 'caller_var'.
 * Parameters : same as function 'map_definitely_process'.
 * Return : same as function 'map_definitely_process'.
 * Note: Since this function is similar to 'map_definitely_process',
 *       we do not explain the common parts.
 *______________________________________________________________*/
map_possibly_process( callee_var, caller_var, callee_in,
                                caller_in, depth ,map_info)
{
    for each 'x_caller' that relationship '(caller_var, x_caller, rel)' exist do
    /* 'rel' can be either D or P relationship. */
    {
        /* check if a P relationship with a variable exists */
        if (caller_var, x_caller, P)
        {
            if (depth > 1)
                add_map_info( caller_var, callee_var, map_info) ;

            if (is_in_callee_scope(x_caller))
                x_callee = x_caller ;
            else{ /* variable 'x_caller' is not in the scope of callee */

                if (exist_invisible_for( x_caller, map_info))
                    x_invisible = get_invisible_var( x_caller, map_info)) ;

                else{ /* an invisible variable does not exit for 'x_caller'*/
                    x_invisible = define_invisible( callee_var, depth) ;
                    add_map_info( x_invisible, x_caller, map_info) ;
                }
                x_callee = x_invisible ;
            }
```

```
            /* add a relationship between 'callee_var' and 'x_callee' in
             * 'callee_in'. Since this function is a recursive function,
             * 'callee_var' could be an invisible variable that has already
             * taken some D relationship. Therefore, we can not simply assign
             * the relationship (callee_var, x_callee, P). */

            /* check if a D relationship exists for 'callee_var'. */
            if (exist_definitely_points-to( callee_in, callee_var))
                /* This can happen when 'callee_var' is an invisible variable
                 * that stands for more than one variable where one of them
                 * has already a D relationship. Since D should be always
                 * unique, we should change the already exist D relationship
                 * to a P relationship. */
                change_D_relation_to_P_relation( callee_in, callee_var) ;

            /* add the new relationship in 'callee_in'. */
            callee_in = callee_in ∪ {(callee_var, x_callee, P)} ;
        }

        [map_info, callee_in] =
        map_possibly_process( x_callee, x_caller, callee_in, caller_in, depth+1) ;

        if (is_struct_type( x_caller))
            for each field 'fi' of 'x_caller'
            {
                x_callee = get_related_callee_variable(x_caller, caller_in,
                                                        callee_in, map_info, fi) ;
                [map_info, callee_in] =
                map_possibly_process( x_callee, x_caller, callee_in, caller_in, 1) ;
            }

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

```
/*______________________________________________________________
 * Function name : map_naming_definitely_struct_process
 * Purpose : assigning names for invisible variables of structure type
 *           which already have a D relationship. This function is recursively
 *           called to assign names for all the structure type variables.
 * Parameters : same as function 'map_definitely_process'.
 * Return : same as function 'map_definitely_process'.
 * Note: Since this function is similar to map_definitely_process, we do
 *       not explain the common parts.
 *______________________________________________________________*/
map_naming_definitely_struct_process( callee_var, caller_var, callee_in,
                                      caller_in, depth ,map_info)
{
    for each 'x_caller' with the relationship '(caller_var, x_caller, rel)' do
    /* 'rel' can be either D or P relationship. */
    {
        /* check if a D relationship with a structure type variable exists */
        if ((caller_var, x_caller, D) and (is_struct_type( x_caller))
        {
            if (is_in_callee_scope(x_caller))
                x_callee = x_caller ;
            else{ /* variable 'x_caller' is not in the scope of callee */
                if (exist_invisible_for( x_caller, map_info))
                    x_invisible = get_invisible_var( x_caller, map_info)) ;
                else{ /* an invisible variable does not exit for 'x_caller'*/
                    x_invisible = define_invisible( callee_var, depth) ;
                    /* set the equivalency of 'x_invisible' and 'x_caller' in 'map_info'. This
                     * information is used by unmap process. It also sets the equivalency of
                     * all the fields of 'x_invisible' and 'x_caller'. */
                    add_map_info( x_invisible, x_caller, map_info) ;
                }
                x_callee = x_invisible ;
            }
        }
        [map_info, callee_in] =
        map_naming_definitely_struct_process( x_callee, x_caller, callee_in, caller_in, depth+1) ;
        if (is_struct_type( x_caller))
            for each field 'fi' of 'x_caller'
            {
                x_callee = get_related_callee_variable(x_caller, caller_in, callee_in, map_info, fi) ;
                [map_info, callee_in] =
                map_naming_definitely_struct_process( x_callee, x_caller, callee_in, caller_in, 1) ;
            }
    }
    return([map_info, callee_in]) ;
}
```

```
/*________________________________________________________________
 * Function name : map_naming_possibly_struct_process
 * Purpose : assigning names for invisible variables of structure type
 *           that a P relationship to them exist. This function is
 *           recursively called to assign names for all the structure type
 *           variables.
 * Parameters : same as function 'map_definitely_process'.
 * Return : same as function 'map_definitely_process'.
 * Note: Since this function is similar to map_definitely_process and
 *       map_naming_definitely_struct_process, we do not explain the
 *       common parts.
 *________________________________________________________________*/
map_naming_possibly_struct_process( callee_var, caller_var, callee_in,
                                    caller_in, depth ,map_info)
{
    for each 'x_caller' that relationship '(caller_var, x_caller, rel)' exist do
    /* 'rel' can be either D or P relationship */
    {
        /* check if a P relationship with a structure type variable exists */
        if ((caller_var, x_caller, P) and (is_struct_type( x_caller))
        {
            if (is_in_callee_scope(x_caller))
                x_callee = x_caller ;
            else{ /* variable 'x_caller' is not in the scope of callee */
                if (exist_invisible_for( x_caller, map_info))
                    x_invisible = get_invisible_var( x_caller, map_info)) ;

                else{ /* an invisible variable does not exit for 'x_caller'*/
                    x_invisible = define_invisible( callee_var, depth) ;
                    add_map_info( x_invisible, x_caller, map_info) ;
                }
                x_callee = x_invisible ;
            }
        }
        [map_info, callee_in] =
        map_naming_possibly_struct_process( x_callee, x_caller, callee_in, caller_in, depth+1) ;
        if (is_struct_type( x_caller))
            for each field 'fi' of 'x_caller'
            {
                x_callee = get_related_callee_variable(x_caller, caller_in, callee_in, map_info, fi) ;
                [map_info, callee_in] =
                map_naming_possibly_struct_process( x_callee, x_caller, callee_in, caller_in, 1) ;
            }
    }
    return([map_info, callee_in]) ;
}
```