Chapter 5

Aggregate Structures

Our analysis concerning scalar variables was presented in Chapters 3 and 4. In order to analyze real C-language benchmarks, one should also consider arrays and structures. In this chapter, we explain how these more general cases are handled. We first discuss the abstract stack representation for non-scalar variables in Section 5.1. Then, in Section 5.2, we explain how updating structured variables is related to the updating of their components. Finally, in Section 5.3, we give the basic rules for calculating the points-to relationship of each possible case.

5.1 Abstract Stack Locations for Arrays and Structures

The major difference between aggregate structures and scalar variables is that aggregate structures are composed of many stack locations. Our objective is to have a name for each location in the abstract stack. We explain our abstract stack representation for arrays in Section 5.1.1, for non-recursive structures[1] in Section 5.1.2, for recursive structures in Section 5.1.3, and for union type in Section 5.1.4.

5.1.1 Arrays

An array is a sequence of variables of the same type. In our analysis, we consider all the elements of an array as one location in the abstract stack. Thus, the points-to analysis finds the relationships between entire arrays, and this information is then

[1] A structure is *non-recursive* if none of its fields (directly or indirectly) points to the same structure, otherwise, it is called a *recursive* structure. Linked list is an example of recursive structure.

used in the array dependence tester which reasons about locations within the arrays [Ban76, Wol89, ZC90]. Our points-to analysis is in fact a pre-processing step for a practical array dependence analyzer which computes dependences for array variables in nested loops [Jus93].

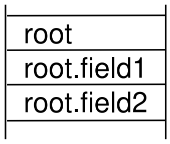
5.1.2 Non-recursive Structures

Unlike in the case of arrays, each element of a structure is explicitly accessible (e.g. a.b). This makes structures more complicated than arrays. Each possible access to a field of a structure should have a unique location, and consequently a unique name, in the abstract stack.

In order to have a unique location for all the possible fields of a structure in the abstract stack, we enumerate the fields of the structure. In the example given in Figure 5.1(a), all possible ways to access the structure `root` (the root of data structure) are `root, root.field1`, and `root.field2`. The corresponding abstract stack representation is shown in Figure 5.1(b).

```
struct {
    int     *field1 ;
    float    field2 ;
} root ;
```

(a) The C-program



(b) The abstract stack representation

Figure 5.1: An example of structure representation in the abstract stack.

In the case of a nested structure, this process should be done recursively until all the possible accesses to a structure are considered. Figure 5.2 represents an example of a nested data structure along with its related abstract stack representation.

What we explained is sufficient for any structure variable which is local to a function but it is not sufficient for global variables and actual parameters of structure type that have a field of pointer type. Assume that `a` is a global variable or actual parameter of structure type that has a field of pointer type `b`. Location `a.b` can point to the location(s) that are not in the scope of that function. Therefore, there are not enough names in the abstract stack for accessing `*(a.b)`. To solve this problem, we use the concept of *invisible variables for structures*.

The concept of invisible variables for structures is similar to the corresponding

```
typedef struct{
    int **sub_field1 ;
    float sub_field2 ;
} FOO ;

struct {
    int     *field1 ;
    FOO      field2 ;
    char   **field3 ;
} root ;
```

(a) The C-program

root
root.field1
root.field2
root.field2.sub_field1
root.field2.sub_field2
root.field3

(b) The abstract stack representation

Figure 5.2: An example of nested structure representation in the abstract stack.

concept for scalar variables which was discussed in Chapter 4. For each actual parameter or global variable of structure type which has a field of pointer type, we add all the invisible variables of corresponding field to the abstract stack. For example, the invisible variables corresponding to variable `x.y` with type `int**` are `x.1_y` and `x.2_y` with type `int*` and `int`, respectively.

Figure 5.3 shows a C-program and its corresponding abstract stacks. The invisible variables corresponding to functions `main` and `f`, with their related types, are shown in Figures 5.3(a) and (b), respectively. Using the invisible variables, we guarantee that any combination for the access to a structure will have a location in the abstract stack.

Without having a specific strategy for *recursive data structures*, this construction method results in infinity. The proposed strategy for handling this problem is given in the next section.

5.1.3 Recursive Structures

Some data structures are defined recursively. This means that they have a field which points to the same data structure (directly or indirectly). Linked list is the simplest example of a direct recursive data structure. We explain our strategy in the design of the abstract stack for the recursive data structures using the linked list example. A linked list is typically defined as:

```
typedef struct{
    int **z ;
    int w ;
} CAT ;

typedef struct{
    int x ;
    CAT *y
} FOO ;

main(){
    FOO *a ;
    ...
    f(a) ;
}

void f(m)
FOO *m ;
{ }
```

	Related type:
a	FOO*

(a) Abstract stack of main()

	Related type:
m	FOO*
1_m	FOO
1_m.x	int
1_m.y	CAT*
1_m.1_y	CAT
1_m.1_y.z	int**
1_m.1_y.1_z	int*
1_m.1_y.2_z	int
1_m.1_y.w	int

(b) Abstract stack of f()

Figure 5.3: An example of the abstract stack representation for invisible variables.

```
struct foo {
    int        data ;
    struct foo *next ;
} cell ;
```

Using the method given in the previous section, we first need to construct the locations shown in Figure 5.4. The process of attaching the fields of the structure

	Related type:
cell	struct foo
cell.data	int
cell.next	struct foo*
cell.1_next	struct foo

Figure 5.4: The incomplete abstract stack for the structure `foo`.

`foo` to the location `cell.1_next` results in an infinite loop. This is due to the fact that there will be always a location with the type of structure `foo`. The solution to this problem is as follows: Each time that we add a new invisible variable of structure type, we first check the type of the previously added fields. If the type already exists, a flag will be set and the new invisible variable points to the corresponding location in the abstract stack. Figure 5.5 shows the abstract stack for the above example. As

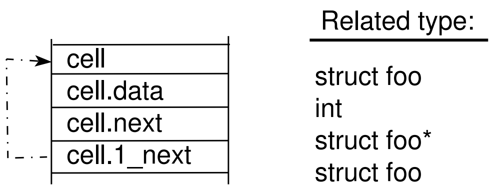


Figure 5.5: The complete abstract stack for the structure `foo`.

the type of cell.1_next already exists (`struct foo`), we do not further add the fields of structure cell.1_next.

An example of a multi-level structure is shown in Figure 5.6(a). The abstract stacks corresponding to functions `main` and `f` are shown in Figures 5.6(a) and (b), respectively.

```
struct cat_type {
    struct foo_type  *bb ;
} ;

struct foo_type{
    int              *cc ;
    struct cat_type  aa ;
    struct foo_type  *next ;
} ;

main()
{
    struct foo_type  x, y ;
    ...
    f(x) ;
}

f(z)
struct foo_type z ;
{...}
```

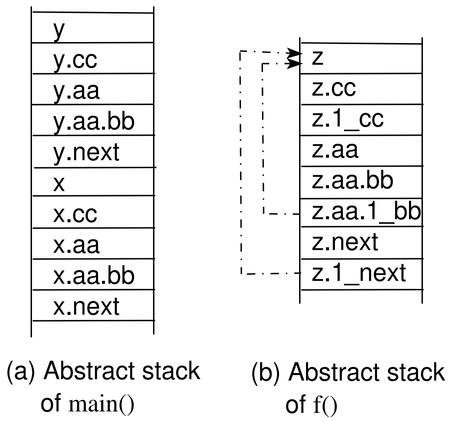


Figure 5.6: An example of a recursive data structure.

In this way, we are actually approximating a recursive data structure to be composed of one cell with a pointer pointing back to itself.

5.1.4 Union Type

Union type variables are not considered in our present version. However, since these are an extension to structures, they can be easily included using the same method. This is achieved by attaching all possible reference names to a union type variable. The only difference between unions and structures is that in the case of unions more names are created. We plan to include unions to our analysis as part of our future work.

5.2 Handling Complex Structure References

A complex structure reference is a variable of structure type. Such variables can appear in a basic statement (e.g., `a = b` when `a` and `b` are of structure type), or as a parameter of a function (e.g. `f(a)` when `a` is of structure type). In this section, we show how our analysis for complex structure references is converted into a sequence of simpler statements that involve only simple structure references. Then, in the next section, we explain our rules to get points-to information for the basic cases involving structure references. We first explain it for the case of intraprocedural analysis (basic statements) and then for interprocedural analysis (function calls).

A basic statement is of structure type if its left hand side (lhs) and right hand side (rhs) are of structure type. Each basic statement of structure type is converted into several simpler basic statements. This is achieved by attaching the fields of the structure to the variables appearing in lhs and rhs. If the newly generated basic statement is of structure type, the same process will be repeated recursively. Finally, we get a set of basic statements of either basic type (`integer` type, `float` type, ect.) or pointer type. Then, we apply our basic rules to each of the newly generated basic statements which are of pointer type. This is shown in the following example:

```
void main() {
   struct foo{
       int *subfield1 ;
   } a, b ;
   struct{
       float       *field1 ;
       int          field2 ;
       struct foo  *field3 ;
   } x, *y ;

   a = b ;  /* stmt 1 */
   x = *y ; /* stmt 2 */
```

Statement 1 is of structure type. This statement is treated as equivalent to the following basic statement:

```
a.subfield1 = b.subfield1 ;
```

Statement 2 is also of structure type. This statement is treated to be equivalent to the following three basic statements:

```
x.field1 = *y.field1 ;  /* stmt 2a */
x.field2 = *y.field2 ;  /* stmt 2b */
x.field3 = *y.field3 ;  /* stmt 2c */
```

The newly generated basic statement 2a is of pointer type and one of the basic cases. Statement 2b is of basic-type and therefore it does not make any difference in the points-to information. Statement 2c is of structure type, and, consequently, the breaking down process is performed which results in the following basic statement:

```
x.field3.subfield1 = *y.field3.subfield1 ;
```

The above basic statement is of pointer type and one of the basic cases. As there are no more basic statements of structure type left, the process is completed.

For interprocedural analysis, the process is the same as explained in Chapter 4. The only difference is in assigning the actual parameters to formal parameters. If a function is called as `f(avar`$_1$`, avar`$_2$`, ...)` and defined as `f(fvar`$_1$`, fvar`$_2$`, ...)`, we should compute the points-to information resolved from:

$\text{fvar}_1 = \text{avar}_1$

$\text{fvar}_2 = \text{avar}_2$

. . .

If any of the above statements is of structure type, we should do the same thing as we did for the basic statements of structure type. This means that for each equation $\text{fvar}_i = \text{avar}_i$ of structure type with `n` fields $\mathtt{f}_1, \mathtt{f}_2, \ldots, \mathtt{f}_n$, we should add the points-to relationships obtained from the following statements:

$\text{fvar}_i.\text{f}_1 = \text{avar}_i.\text{f}_1$

$\text{fvar}_i.\text{f}_2 = \text{avar}_i.\text{f}_2$

. . .

$\text{fvar}_i.\text{f}_n = \text{avar}_i.\text{f}_n$

The same process will be repeated for any newly generated basic statement which is of structure type.

We conclude that the problem of intraprocedural and interprocedural points-to analysis for complex structures can be broken into smaller problems concerning basic statements of pointer type with aggregate structures in lhs and rhs. In the next section, we explain how these basic statements can be solved.

5.3 Points-to Analysis for Aggregate Structures

In the previous sections, we explained the stack representation for aggregate data structures. In this section, we expand our points-to analysis for aggregate structures. Figure 5.7 shows an overall view of our points-to analysis. The only case that aggregate structures can affect interprocedural analysis is when a parameter is of structure type. In Section 5.2, we showed that this case can be divided into some basic cases and then the general interprocedural rules given in Chapter 4 will be applied. The intraprocedural analysis is divided into two major groups: (i) basic statements, (ii) compositional control statements. As compositional control statements are independent of the type of data structure, there is no need to consider them in this chapter. Consequently, it is sufficient to find a solution for all the basic statements. Table 5.1 summarizes all basic statements that affect points-to relationships (when the statement is of pointer type). This is a generalized version of the Table 3.3 which

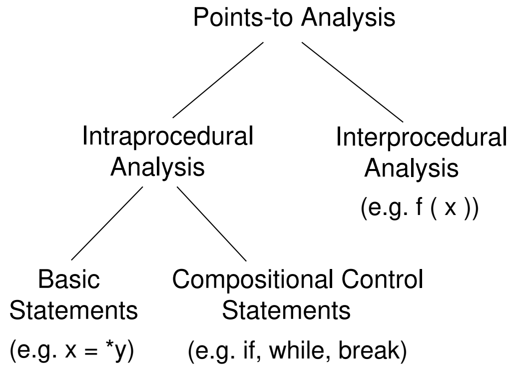


Figure 5.7: The overall view of points-to analysis.

is only for scalar variables. The cases which are denoted by the bold-face notation correspond to the entries of Table 3.3.

lhs ↓ rhs→	&y	&y[j]	&y.b	y	y[j]	y.b	*y	(* y).b	(* py)[j][2]	& (*y).b	&(*py)[j][3]
x	**1.1**	1.2	1.3	**2.1**	2.2	2.3	**3.1**	3.2	3.3	7.1	7.2
x[i]	1.4	1.5	1.6	2.4	2.5	2.6	3.4	3.5	3.6	7.3	7.4
x.a	1.7	1.8	1.9	2.7	2.8	2.9	3.7	3.8	3.9	7.5	7.6
*x	**4.1**	4.2	4.3	**5.1**	5.2	5.3	**6.1**	6.2	6.3	8.1	8.2
(*x).a	4.4	4.5	4.6	5.4	5.5	5.6	6.4	6.5	6.6	8.3	8.4
(*px)[i][2]	4.7	4.8	4.9	5.7	5.8	5.9	6.7	6.8	6.9	8.5	8.6

Table 5.1: All possible cases of basic statements which affect the points-to information when basic statement is of pointer type.

The different kinds of basic statements that can affect points-to analysis are divided into eight cases which are numbered from 1 to 8 in Table 5.1. Each case, which is divided into some subcases, has its own general rule which applies to both scalar variables and aggregate structures. The left hand side number in each entry of the table corresponds to the case and the right hand side number corresponds to the subcase. For example, the entry 7.3 refers to subcase 3 of case 7. To increase the readability of the table, multi-indices such as $x[i_1][i_2][\ldots][i_n]$ are represented as one index such as $x[i_1]$, also multi-fields such as $x.f_1.f_2\ldots f_n$ are represented as one field such as $x.f_1$ in the table. This abbreviation does not impose any limitation on the generality and one can easily replace the abbreviated representation by its original form.

[2] (*py)[j] is the semantic representation of py[j] where 'py' is a pointer to an array. This is not a valid syntax in C-language. We just use this notation to differentiate between pointers to arrays and ordinary arrays.

[3] &(*py)[j] is the semantic representation of &py[j] where 'py' is a pointer to an array. We just use this notation to differentiate between pointers to arrays and ordinary arrays.

The following is the list of aggregate structures that can appear in a basic statement. We first explain the effect of each of the aggregate structures and then give the rules corresponding to each case.

1. $z[i_1][i_2][\ldots][i_n]$ and $\&z[i_1][i_2][\ldots][i_n]$

   In this representation, 'z' is explicitly defined as an array. Although in our abstract stack there is only one location related to each access of an array, we treat the first element of an array differently from other elements (the first element is used when indices i_1 to i_n are equal to zero). The reason behind this approach is that the analyses which needs to know when a pointer variable points to the beginning of the array and when it may point to somewhere in the middle of the array. This is particularly important for the array dependence module which needs to ensure that pointers to arrays point to the beginning of an array.

   In the following example after processing statement 1, we would say that `b` definitely-points-to `a` if `i` is zero, otherwise, we would say that `b` possibly-points-to `a`.

   ```
   main() {
       int a[7], *b ;
       int i ;
       ...
       b = &a[i]     /* stmt1 */
   }
   ```

   The replacement of arrays by scalars in a basic statement does not make any difference to our points-to rules when the first index of an array is used. If the middle index is used, the corresponding relationship should be changed to possibly-points-to.

   The notation $\&z[i_1][i_2][\ldots][i_n]$ refers to the address of the abstract stack location related to 'z'.

2. $(*z)[i_1][i_2][\ldots][i_n]$ and $\&((*z)[i_1][i_2][\ldots][i_n])$

   In the C-language, there are two ways to access an array. First is a direct access which is done through the name of the array, e.g. `a[i]` where `a` is an array. Second is an indirect access which is done through a pointer to an array, e.g. `pa[i]` where `pa` is a pointer to array(s). These two ways of accessing are syntactically the same, however they have a complete different meaning and are represented differently in SIMPLE (as explained in Chapter 2). The notation `pa[i]` means the `i`'th element of variable(s) that `pa` points-to. We use the notation `(*pa)[i]` to represent the actual meaning of `pa[i]`.

This case behaves the same as the general category `*x` when the first index of an array is used. When the middle element of an array is used, it behaves differently. In case of indirect reference to arrays, two factors should be considered: (i) the use of the first index of the array, (ii) the possibly/definitely-points-to relationship of the array pointer to other variable.

3. $z.f_1.f_2 \ldots f_n$ and $\&z.f_1.f_2 \ldots f_n$

   Variables of this form have a unique location in the abstract stack. Their only difference with scalar variables is that they have a more complicated name. After finding their location in abstract stack, they are treated in the same way as scalar variables.

4. $(^*z).f_1.f_2 \ldots f_n$ and $\&(^*z).f_1.f_2 \ldots f_n$

   This case is similar to `*z`. The only difference is that to get the actual location, one should attach fields $.f_1.f_2 \ldots f_n$ to the variable(s) that `z` points-to.

In the rest of this section, we give the detailed rules for the cases 1.8 and 8.4 of Table 5.1. These cases are selected to explain the idea. Some more rules are given in Appendix C. We use the following notations in conjunction with these rules:

- D: represents definitely-point-to relationship.
- P: represents possibly-point-to relationship.
- $\bowtie$: represents merging which is defined in Table 3.1.
- first_elem: is defined to distinguish the first element of an array. The definition is as follows:

  $$\text{first\_elem}([i_1][i_2][\ldots][i_n]) = \begin{cases} D & \text{if} \quad i_1 = i_2 = \ldots = i_n = 0 \\ P & \text{otherwise} \end{cases}$$

  In case of scalar variables, first_elem is always D.
- input: the points-to relationships at the entry point of a basic statement.
- gen: the newly generated points-to relationship(s) by a basic statement.
- kill: the points-to relationship(s) that are killed in a basic statement.
- changed_input: is an updated version of 'input' where some of the definitely-points-to relationships are changed into possibly-points-to.

Case 1.8: The general format of this case is:

```
struct{
    int *a ;
} x ;
int y[size] ;
. . .
x.a = &y[j] ;
```

The rules applied in this case are as follows:

kill = { (x.a,x1,rel) | (x.a,x1,rel) $\in$ input }
gen = { (x.a,y, first_elem([j])) }
return(gen $\cup$ (input - kill))

Consider the following example:

```
main() {
    struct{
        int *f1 ;
    } x, *z ;
    int y[70], w ;

    z = &x ;              /* stmt 1 */
    (*z).f1 = &w ;        /* stmt 2 */
    x.f1 = &y[0] ;        /* stmt 3 */
}
```

Case 1.8 appears at statement 3. Using the given rules, the sets related to this statement are as follows:

input = {(z,x,D), (x.f1,w,D)}

kill = {(x.f1,w,D)}

gen = {(x.f1,y,D)}

output = {(z,x,D), (x.f1,y,D)}

Case 8.4: The general format of this case is:

```
struct foo{
    int *a ;
} *x ;
int *py ;
. . .
(*x).a = &(*py)[j] ;
```

The rules applied in this case are as follows:

kill = { (x1.a,x2,rel) | (x,x1,D), (x1.a,x2,rel) $\in$ input }
gen = { (x1.a,y1,rel1 $\bowtie$ rel2 $\bowtie$ first_elem([j])) | (x,x1,rel1), (py,y1,rel2) $\in$ input}
changed_input = (input - { (x1.a,x2,D) | (x,x1,P), (x1.a,x2,D) $\in$ input})
$\cup$ { (x1.a,x2,P) | (x,x1,P), (x1.a,x2,D) $\in$ input}
return(gen $\cup$ (changed_input - kill))

Consider the following example:

```
f(float b, int i) {
    struct{
        float *a ;
    } *x, z ;
    float *py, w[70] ;

    x = &z ;             /* stmt1 */
    if (b == 0) ;
        z.a = &b ;       /* stmt2 */
    py = &w[0] ;         /* stmt3 */
    (*x).a = &py[i] ;    /* stmt4 */
}
```

Case 8.4 appears at statement 4. Since `py` is a pointer to an array, we treat `&py[i]` as `&(*py)[i]`. Using the given rules, the sets related to this statement are as follows:

input = {(x,z,D), (z.a,b,P), (py,w,D)}

kill = {(z.a,b,P)}

gen = {(z.a,w,P)}

changed input = input

output = {(x,z,D), (z.a,w,P), (py,w,D)}

5.3.1 Interprocedural Points-to Analysis for Recursive Data Structures

In this section, we explain the interprocedural analysis in the case of a recursive data structure. The major point is that, when an invisible variable, say `1_x`, is of recursive data structure type, all the instances resolved from `1_x` are represented by `1_x`. This compact representation can result in some imprecision. This is shown in the example given in Figure 5.8 Readers can skip the rest of this section without losing the continuity.

```
typedef struct foo{
        int          data ;
        struct foo *next ;
} FOO ;

FOO *a, d ;

main()
{
    FOO b, c ;
    a = &b ;
    b.next = &c ;
    f() ;
}
f()
{
    a->next->next = &d ;
}
```

(a) The C-program

```
typedef struct foo{
        int          data ;
        struct foo *next ;
} FOO ;

FOO *a, d ;

main()
{
    FOO b, c ;
    a = &b ;              /* stmt 1 */
    b.next = &c ;         /* stmt 2 */
    f() ;                 /* stmt 3 */
}
f()
{
    FOO *temp0 ;
    temp0 = (*a).next ; /* stmt 4 */
    (*temp).next = &d ; /* stmt 5 */
}
```

(b) The simplified version

Figure 5.8: An example of interprocedural points-to analysis in the presence of recursive data structures.

The points-to information at statement 3 of Figure 5.8 (before the function call) is shown in Figure 5.9. The points-to information are shown in the following three different formats: (i) abstract stack representation, (ii) memory representation, and (iii) points-to triples.

The map process should pass the information concerning the global variables to the function `f`. From the relationship (a,b,D), we resolve (a,a_1,D) for function `f` (because `b` is not in the scope of `f`). From the relationship (b.next,c,D), we resolve

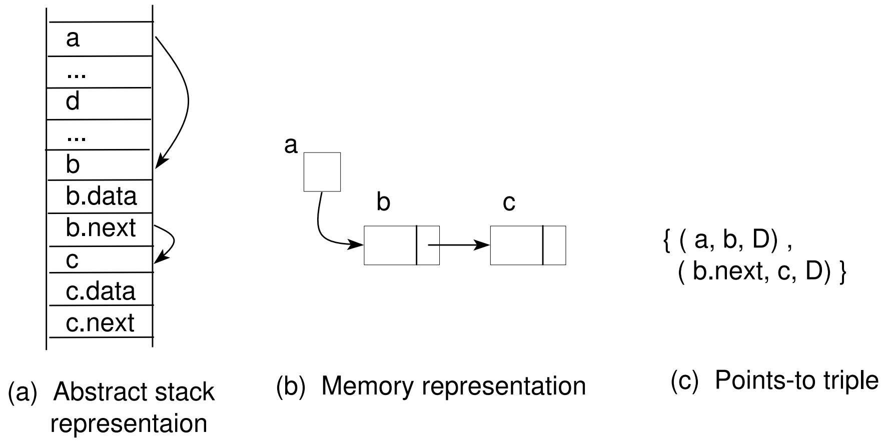


Figure 5.9: The points-to information of function `main` before the map process.

the relationship (1_a.next,1_a.1_next,P). Note that as variable `c` is not in the scope of function `f`, we have incorporated the new invisible variable (1_a.1_next). The reason that the definitely-points-to relationship in caller `main` is changed to possibly-points-to relationship in callee `f` is that the location 1_a is of recursive structure type (explained in Section 5.1.3), and, consequently, has the potential of standing for more than one stack location. As a result, the points-to relationships related to this type of location must be possibly-points-to relationships. Figure 5.10 shows the corresponding points-to information. Figure 5.10(b) shows the way that we consider the memory locations. This is called as the 'simulation of memory location' where the word 'simulation' reflects the fact that it is different from the way that the actual memory looks like.

Statement 4 resolves the relationship (temp0,1_a.1_next,P). Statement 5 refers to location `(*temp).next` and location `*temp` refers to `1_a.1_next`. As it is shown in Figure 5.11(a), this location does not have any other fields. However, as this location is related to the recursive data structure `1_a`, it will take the `next` field corresponding to `1_a`. That is why we get the relationship (1_a.next,d,P) (the reason for having a possibly-points-to relationship is already explained). The points-to information after finishing the process of function `f` and before unmap process is shown in Figure 5.11.

Next, we apply the unmap process. Since locations `1_a` and `1_a.1_next` stand for variables `c` and `b`, respectively (the corresponding information is saved in the `map-info` and is represented in Figure 5.10(d)), we get the relationships (b.next,c,P) and (b.next,d,P). The location `1_a.next` also stands for location `1_a.1_next.next` (because it is a recursive data structure). This fact resolves the relationships (c.next,c,P)

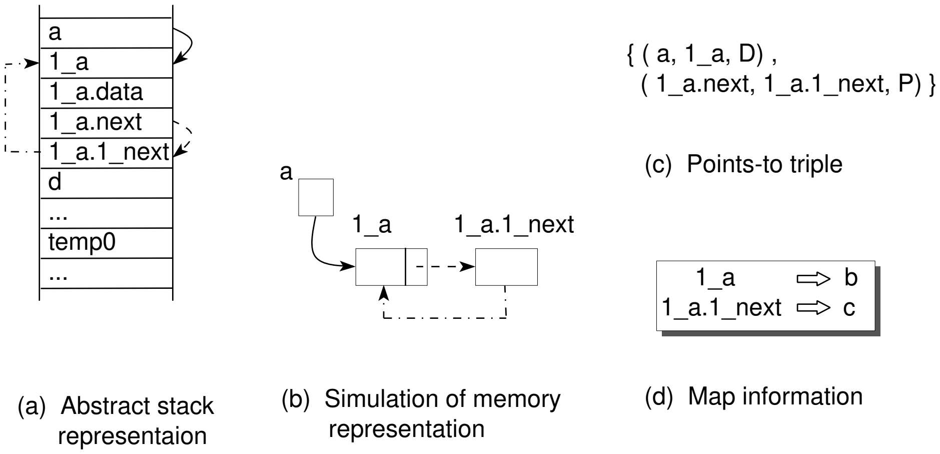


Figure 5.10: The points-to information of function `f` after the map process.

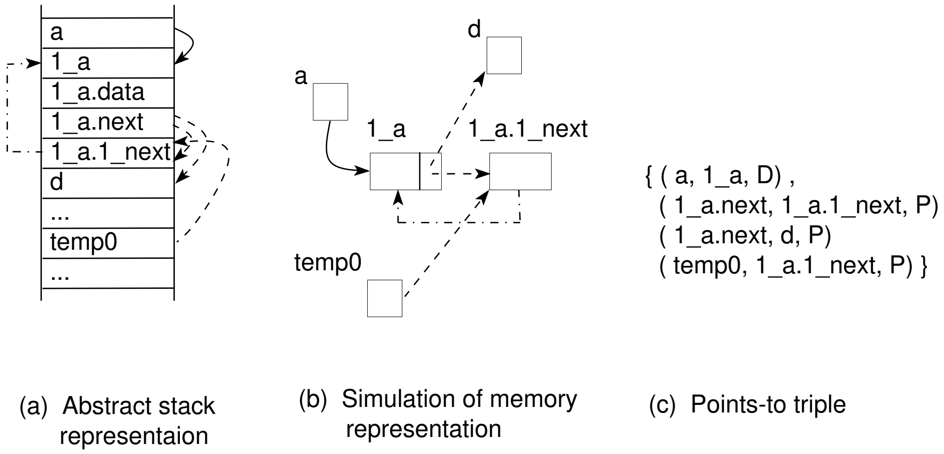


Figure 5.11: The points-to information of function `f` before the unmap process.

and (c.next,d,P). The final points-to information of function `main` is shown in Figure 5.12.

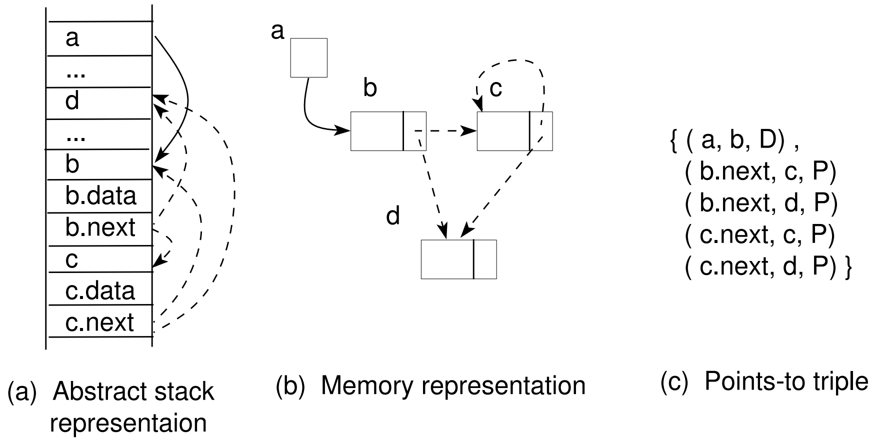


Figure 5.12: The points-to information of function `main` after the unmap process.