Chapter 7

Implementation Details and Limitations

In this chapter, we explain some details of our implementation. This is composed of: our data structure for the abstract stack representation in Section 7.1, some special features of our analysis in Section 7.2, some details of map and unmap processes in Section 7.3, and finally the library functions in Section 7.4.

7.1 Abstract Stack Data Structure

The concept of an 'abstract stack' is introduced in the previous chapters. In this section, we explain its actual data structure and the reasoning behind this implementation.

For several reasons which will be explained later, we use a bit matrix data structure for our abstract stack representation. As a bit matrix requires an array of a fixed size, we first pass through the program to collect the variables corresponding to each function. To minimize the size of the bit matrix, we collect only the variables which are involved in points-to analysis. These are: (i) the actual/formal parameters of pointer types, and (ii) the variables appearing in the lhs/rhs of the assignment statements represented in Table 5.1. Each function has its own bit matrix where the size of this matrix depends on the number of collected variables. If a global variable `x` is one of the collected variables of any of the functions, it should be also included in the collected variables of all other functions (because it is indirectly accessible by those functions). After collecting the variables for each function, we perform the points-to analysis starting from the `main` function. This analysis is composed of the following steps:

1. Building the points-to bit matrix for the function `main` using the collected variables,
2. Calling the 'points-to' function (given in Chapter 4).

Each index of a bit matrix represents a variable in the related function. The global variables keep the same index in all the bit matrices. There are two types of bit matrices, one for definitely-points-to relationships and the other one for possibly-points-to relationships. If the `(i,j)`'th element of the definitely-points-to bit matrix is equal to 1, it means that the variable corresponding to the `i`'th element has a definitely-points-to relationship with the variable corresponding to the `j`'th element. If the content of the `(i,j)`'th element in both definitely-points-to and possibly-points-to bit matrices is equal 0, it means that the variables related to `i` and `j` have no relationship.

Figure 7.1 shows a graph representation of the corresponding data structure. Since dynamically allocated functions (like `malloc`) are costly, the whole memory required for the complete data structure (as represented in Figure 7.1) is computed and allocated only once. The necessary links are computed by performing some arithmetic operations involving sizes of the fields.

The following are the advantages of using the bit matrix data structure:

- **Memory Space**: As a single bit is used to indicate if two variables are related or not, we are using the minimum space for our representation. For example, if there are 60 variables in a function which are involved in points-to analysis and an integer is composed of 32 bits, the related bit matrices are two arrays of integer type with 60 rows and 2 columns (this can be further improved to a 2 by 2 matrix).
- **Speed**: The operations are fast and efficient because they are mainly composed of bit operations. Some examples are given in the following:
  - The merge of two matrices is the 'logical or' of the two.
  - One may need to access the `(i,j)`'th element of a bit matrix. In this case, index `i` provides a bit vector to look for the index `j`. The integer part of the ratio of `j` to the number of bits per word (in this case 32) gives the actual column number for `j`. The remainder in dividing `j` by 32 gives the bit number in that column. For example, for `j = 65`, the column number is

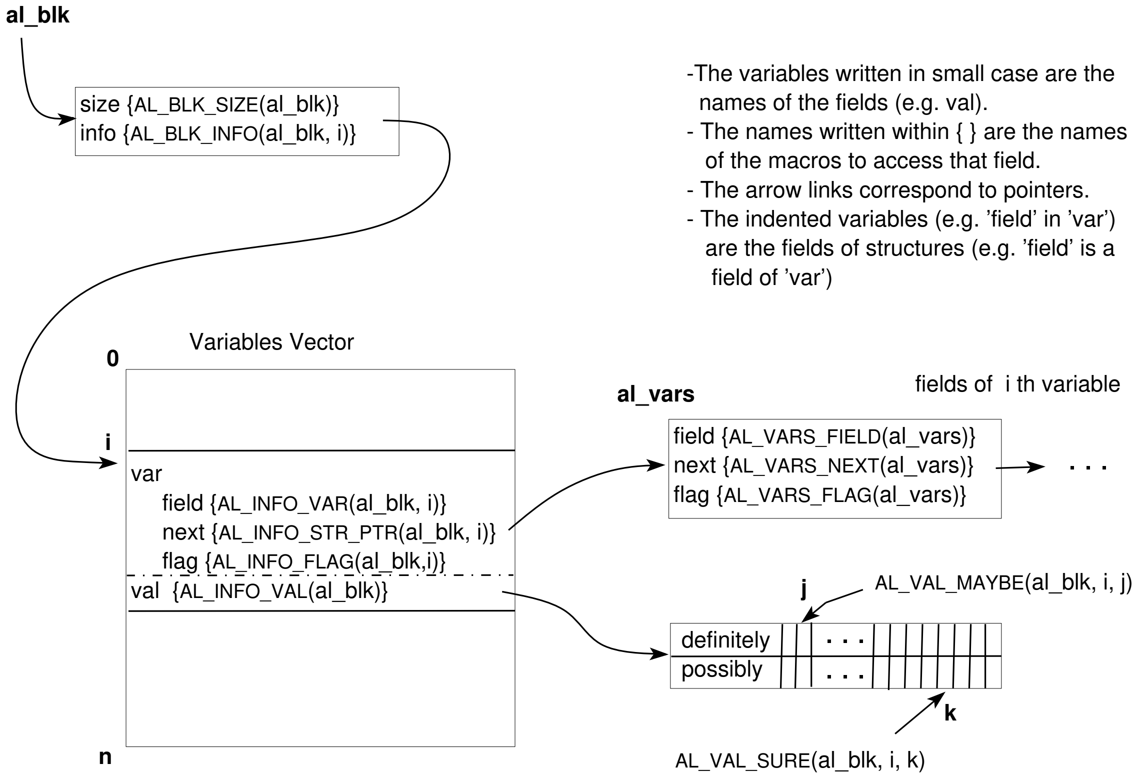


Figure 7.1: The data structure of the abstract stack.

2 and the bit number in 1. This means that the first bit in `(i,2)`'th element of the corresponding bit matrix is demanded. Masking is used to access a given field. An array of masks is defined to speed up the accessing process. The size of this array (maximum number of masks) is equal to the number of bits per integer (in this case 32).

– To determine if $*$`a` and $*$`b` refer to the same locations, one needs only to know if two array pointers are pointing to the same array(s). In this case, a logical 'xor' operand is used. If the result of 'xor' of the rows which are related to the pointer variables is 0, we conclude that the variables are pointing to the same location(s). This is particularly useful for the array dependency module that needs to know if two array pointers point to the same array.

- **Accessibility:** Since we need to frequently check the points-to information, it is important to have a fast access from a variable to its index in the bit matrix and vice versa. In our case, this is quite fast because there is a correspondence between each index of the bit matrix and a variable node in the tree.

7.2 Some Special Features of Our Analysis

- As the result of points-to analysis is used by other analyses, the related information should be saved for later use. The corresponding bit matrices are saved at the statement level. If a function is called more than once, to save some space, the information from different calls is first merged together and then the result is saved. Therefore, the possibly-points-to information may be due to the merge between two different calls and not due to a loop or to a condition. If space were not an issue, one could easily save the points-to information for each statement separately (each containing the related invocation graph node). We are planning to further investigate this issue in our future work. Presently, the points-to information can be used in one of the following two ways:

  1. Other analyses are done simultaneously with the points-to analysis. In this way, those analyses obtain the precise points-to information. The disadvantage of this approach is that it is more complicated from the implementation point of view.
  2. Other analyses are done after the points-to analysis is completed. The advantage of this approach is that the implementation is straight-forward. Its disadvantage is the possibility of getting more possibly-points-to relationships.

- As mentioned before, the points-to information is saved for each statement. The sequence of statements that do not affect the points-to analysis have the same points-to information. To save some memory space, this sequence of statements point to the same bit matrix.
- Our method can perform some correctness checking for the program. This is normally done at run-time. These checks produce some error/warning messages to the user. For precise checking, we initialize all the pointer variables to point to NULL at the beginning of each function. The following is the list of the checks which are performed:

  1. If the user applies an indirect reference operand to a variable which definitely points to NULL, an error message together with the corresponding line number is reported and that line is ignored. The following are two examples of this case:

```
main() {
    int **a, c ;
    *a = &c ;
}
```

```
main() {
    int **a, c ;
    if (...)
        *a = &c ;
}
```

In these examples an error message appears for statement `*a = &c`. The error occurs because `a` does not point to any variable (it definitely points to NULL) and, consequently, `*a` does not exist.

If the indirect reference operand is applied to a variable that possibly points to NULL, it is not guaranteed to be an error. To clarify this point, consider the following example:

```
main() {
    int **a, *b, c ;
    if (...)
        a = &b ;
    if (...)        /* s1 */
        *a = &c ;   /* s2 */
}
```

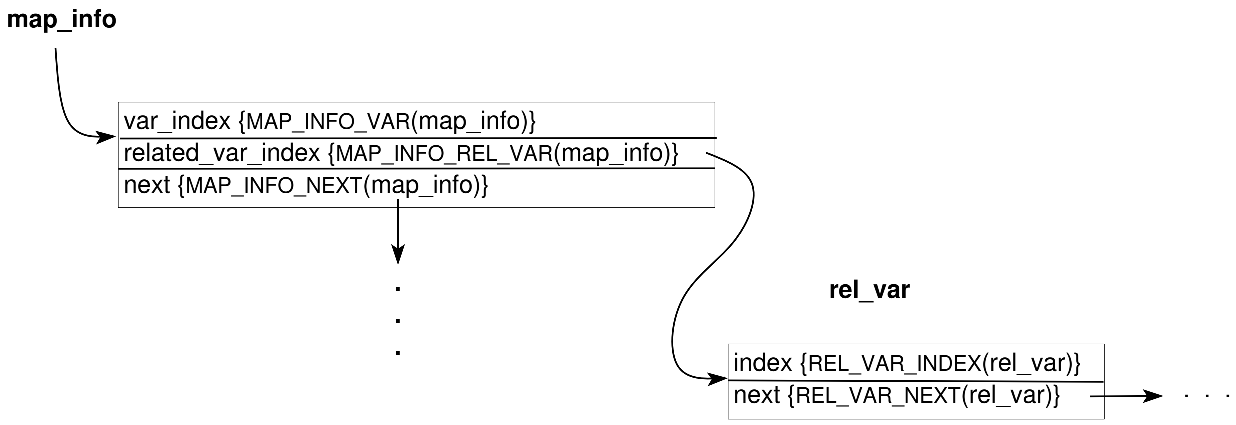
At `s1`, we get `(a,b,P)`, `(a,NULL,P)`. Although at `s2`, `*a` might be NULL, but since this is not definite, no message is given.

2. If there is an arithmetic operation on a pointer variable which is not pointing to an array, the worst case assumption is made, that is, the variable points to all other variables of its scope. As such arithmetic operation might be due to a programming error, a warning message is generated.

- Since system commands, e.g. `exit()`, are not part of the C-language, they are not supported by our analysis. This means that our analysis may be overly conservative since we could include non-realizable paths after the `exit()`.
- *Union* type variables are not considered in the present version of our analysis. However, since these are an extension to structures, a possible solution is given in Section 5.1.4.
- Functions with a variable number of parameters are not considered.

7.3 Some Implementation Details of Map and Unmap Processes

As it was mentioned in Chapter 4, the map information (`map-info`) is used to save the relationship between the invisible variables of callee and the local variables of caller. The data structure of map-info is a two dimensional linked list which is represented in Figure 7.2.



-The variables written in small case are the names of the fields (e.g. var_index).
- The names written within { } are the names of the macros to access that field.
- The arrow links correspond to pointers.
- Following is a description of 'map_info' (map information) fields:
  - var_index : is the index of an invisible variable in callee
  - related_var_index : is a pointer to the variable(s) in caller which are represented by 'var_index'
  - next : is the pointer to the next invisible variable
- Following is a description of 'rel_var' (related variables) fields:
  - index : is the index of a variable in caller
  - next : is the pointer to the next variable in caller

Figure 7.2: The data structure of map information (`map_info`).

Any interprocedural analysis that uses the points-to information, may need to access the invisible variables. Therefore, we need to save the map information. Following shows an example of constant propagation where the map information is used.

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

    a = &b ;
    f(a) ;
}
void f(int *x)
{
    *x = 1 ;      /* stmt 1 */
}
```

Since at statement 1 we have `(x,1_x,D)`, after constant propagation `1_x` is set equal to constant 1. After returning to the `main` function, we should conclude that `b` is equal to constant 1. This can be done only if we know that `1_x` is equivalent to `b`.

The map information (`map-info`) related to each function call is saved in the related node in the invocation graph. This information can be easily used by other analyses.

7.4 Library Functions

All the library functions except those listed in the following are side effect free.

`malloc, calloc, valloc, realloc, memalign, mallopt, alloca`

The above functions generate a points-to relationship between variables and the heap which is discussed in Chapter 5. Note that function `alloca` allocates a location from stack and releases it after returning from the function. In this case, we still give the points-to relationship to heap, because it does not change the generality of the heap analysis. Note that we do not handle `memcpy` because we believe it is rarely used to affect the points-to information. However, we will send a message to the user. If this case happens frequently, we can easily handle it.