# Chapter 6

# Handling other Pointer Features in C

In the previous chapters, we discussed points-to analysis in detail. In the C-language there are many other kinds of pointer manipulation that can effect points-to analysis. In this chapter, we briefly discuss these cases and give a simple solution for them. These solutions have been implemented to enable us to perform experiments on a reasonable set of benchmarks. In Section 6.1, we discuss the pointer arithmetic, in Section 6.2 we explain our analysis for type casting, and in Section 6.3, we consider heap-allocated data structures.

## 6.1 Pointer Arithmetic

The C-language allows arithmetic operations on variables having pointer-type. Here is a simple example of this case:

```
int *a, i ;
a = a + i ;
```

In most cases, instead of increasing the array index, a pointer to array is increased. In other words, arithmetic operations are used to access different elements of an array. Due to this fact, we treat the arithmetic operations achieved on the pointer to arrays differently. We basically divide different cases into the following two basic groups:

1. The arithmetic operation involves a pointer to a variable which is not of array-type. In this case, there is no limitation in the corresponding arithmetic operations. The only potential problem is that we may access the wrong memory location. Since there is no consistent way of following the variables after the arithmetic operations, we use the worst case assumption. This assumption says that the pointer variable has a possibly-points-to relationship with all the variables of its scope.
2. The arithmetic operation involves a pointer to a variable of array-type. If the pointer `x` points-to the array `a`, the corresponding relationship is divided to the following two subgroups:
   (a) `x` definitely-points-to `a`. This case happens when location `x` points-to the first element of array `a`. If we have an arithmetic operation on `x`, then `x` does not point to the first element any more. In this case, the definitely-points-to relationship should be changed to possibly-points-to. The following is an example:

   ```
   main() {
       int *x, a[7] ;
       x = &a[0] ;  /* stmt1 */
       x = x + 1 ;  /* stmt2 */
   }
   ```

   After processing statement 1, we get the relationship (x,a,D) as the result, while processing statement 2 results in the relationship (x,a,P).

   (b) `x` possibly-points-to `a`. This case happens when location `x` points-to one of the elements of the array `a` (not the first one), or when we have a control statement. Obviously, after performing an arithmetic operation on `x` in either of these two cases, we still do not know which element of array `a` is pointed to by `x`. This means that the corresponding relationship remains unchanged. Here is an example of this case:

   ```
   main() {
       int *x, a[7], i ;
       ...
       x = &a[i] ;  /* stmt1 */
       x = x + 1 ;  /* stmt2 */
   }
   ```

   After processing statement 1, we get the relationship (x,a,P) as the result. The process of statement 2 does not make any difference in the final result.

In accessing the arrays, we assume that the index expression is within the corresponding boundaries.

## 6.2 Type Casting

Type casting in the C-language allows one to assign variables of different types to each other. Since the type of a basic statement is a determining factor in deciding whether that statement should be considered in the points-to analysis or not, type casting plays an important role in this analysis. In the C-language, the programmer is allowed to change the types with no limitation. This flexibility of the C-language necessitates special precautions in the analysis.

Any type casting appearing as an argument in a function call is removed by the simplifier program (by using a temporary variable). Thus all arguments match the type of their corresponding formal parameter. Furthermore, any type casting appearing in the left hand side of an assignment statement is moved to the right hand side of the assignment. As a result, type casting is an issue only in the intraprocedural analysis.

Define the *pointer-level* of a variable `x`, denoted as *pointer-level(x)*, as the maximum number of indirect references that `x` may have. For example, the pointer-level of `x` and `y` with types `float***` and `int` are 3 and 0, respectively. The points-to relationship (x,y,rel) is devised in such a way that the type of the variable `x` is always a pointer to the type of the variable `y`. In other words:

$$\text{pointer-level}(\mathtt{x}) = \text{pointer-level}(\mathtt{y}) + 1 \text{ , and, type } (*\mathtt{x}) = \text{type } (\mathtt{y})$$

The only exception is in the presence of type casting, where one may have points-to relationship between variables of different type and pointer-level. In this case, assuming a points-to relationship (x,y,rel), any of the following relationships between x and y can appear:

- $\text{pointer-level}(\mathtt{x}) = \text{pointer-level}(\mathtt{y}) + 1 \text{ , and, type } (*\mathtt{x}) \neq \text{type } (\mathtt{y})$
  There might be a points-to relationship between the variables of correct difference on pointer-level, but with different base-type. Consider the following example:

```
typedef struct{
    int a ;
} FOO ;

float y ;
FOO *x ;
x = (FOO *) &y ;  /* stmt 1 */
```

Here, `x` has the type `FOO`$*$ while `y` has the type `float`. After processing statement 1, we get the relationship (x,y,D).

- pointer-level(`x`) = pointer-level(`y`)

  There might be a points-to relationship between the variables of the same pointer-level. Consider the following example,

  ```
  int *x
  float *y ;
  x = (int*) &y ;   /* stmt 1 */
  ```

  Here, pointer-levels of `x` and `y` are equal to 1. After processing statement 1, we get (x,y,D).

- pointer-level(`x`) $<$ pointer-level(`y`), or, pointer-level(`x`) $>$ (pointer-level(`y`) + 1)

  There might be a points-to relationship between the variables of different pointer-level. Consider the following example,

  ```
  int *x, **y ;
  x = (int*) &y ;   /* stmt 1 */
  ```

  Here, `x` has the type `int`$*$ and `y` has the type `int`$**$. After processing statement 1, we get the relationship (x,y,D).

There are special cases of type casting that are more complicated to handle. These cases happen when user forces a base-type (e.g. `int, float`,etc.) variable to get the address of another variable. In other words, user pushes a non-pointer variable to get the address of another variable. Following shows an example of this case:

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

In this example, we should get (a,b,D). At present, we do not handle statements which have a lhs of scalar type. Our analysis can be easily extended to take care of this case, however, we are not sure if it is worthwhile.

## 6.3 Dynamic Data Structures

In this section, we discuss the last possible case that results in a points-to relationship. This is the points-to information obtained from the *heap allocation.* There are some library functions in the C-language which get a location of a given size from the heap and assign it to a variable during the execution time. The following is an example:

```
int *x ;
x = (int*) malloc (sizeof(int))  /* stmt 1 */
```

After the execution of statement 1, a heap location of size 'integer' will be assigned to the variable `x`. This means that `x` is pointing to a location in the heap. As there is no consistent way for using the heap, it is considered as a bunch of memory locations with no specific type which can be pointed to by any type of variable; or can point to any type of variable. Figure 6.1 is devised to explain our way of looking at the heap.

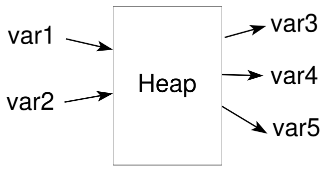



Figure 6.1: An overview of the abstract heap representation.

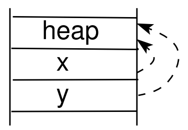
To incorporate heap into our analysis, we define a single location in the abstract stack, called `heap`, which has no specific type. This location represents the whole heap. This approach of approximating the heap as one location introduces conservative approximations for relationships within the heap. However, these relationships can be further refined by a heap points-to analysis that is run after the stack points-to analysis [Ghi93a]. As we have found that it is very rare to have pointers back from the heap to the stack, this division into stack and heap analysis phases is very reasonable. All the points-to relationships dealing with `heap` location are of type possibly-points-to. The reason is that we do not exactly know which location of heap is considered. This idea is further explained using the example given in Figure 6.2(a).

```
typedef struct foo{
    int a ;
    int *b ;
} FOO ;

void main()
{
    FOO *x ;
    int *y ;

    x = (FOO*) malloc (sizeof(FOO)) ; /* stmt1 */
    y = & (*x).a ;                    /* stmt2 */
}
```

(a) The C program




(b) The related abstract stack

Figure 6.2: An example of dynamic data allocation.

In this example, statement 1 allocates some memory locations from heap. As the result, we get the relationship (x,heap,P). The `&(*x).a` at statement 2 is a location in the heap. Due to this fact, after processing this statement, we get the relationship (y,heap,P). The final stack representation of this program is shown in Figure 6.2(b). Functions causing the heap allocations are: `malloc`, `calloc`, `valloc`, `realloc`, `memalign`, `mallopt`, and `alloca`. The same process as explained above has to be done for them.