# Chapter 3

# Intraprocedural Points-to Analysis for Scalar Variables

In this chapter, we present a detailed explanation of our method for collecting the intraprocedural points-to information for stack-allocated data structures in the C language. This concerns the points-to analyses for the programs where no function calls and aggregate structures (e.g. arrays and structures) are present. This presentation forms the basis for extending the analysis to interprocedural analysis (Chapter 4) and handling aggregate structures (Chapter 5).

This chapter is structured as follows. In Section 3.1, we present a simple example to explain our motivation behind the proposed approach. This example is intended to prepare the reader for the main approach which will be explained in Section 3.2. The relationship between our approach and alias information is explained in Section 3.3. In Section 3.4, we discuss our notations together with the main algorithm. The details of basic statements and compositional control statements are explained in Sections 3.5 and 3.6, respectively. Finally, a summary of the chapter is given in Section 3.7.

## 3.1 Motivating Example

Figure 3.1(a) shows a simple C program. As explained in Chapter 2, a C-program is represented in the SIMPLE AST format where all the analyses are performed. After simplification, we obtain the version shown in Figure 3.1(b). Note that the statement `**x = 5` in Figure 3.1(a) is changed into an equivalent form composed of statements 4 and 5 in Figure 3.1(b).

Figure 3.1 exemplifies our abstraction of intraprocedural analysis at each statement. Notice that we use abstract stack location and memory representation to illustrate the value of each variable. After processing statement 1, variable `z` gets the value of constant 1 (Figure 3.1). After processing statement 2, location `x` points to location `y` (Figure 3.1(d)). After processing statement 3, by following the link in abstract stack or memory representation, we can see that $*$`x` is equivalent to `y`. Using this fact, we conclude that the location `y` in the abstract stack points to the location `z` (Figure 3.1(e)). After processing statement 4, `temp0` gets the same value as $*$`x` which is equivalent to `y`. This means `temp0` points to the same location as `y` is pointing to, namely `z` (Figure 3.1(f)). Finally, after processing statement 5, by following the abstract stack, we see that $*$`temp0` is the same as location `z`. Using this fact, `z` gets the value of constant 5 (Figure 3.1(g)).

This example shows how we follow the abstract stack location to collect the information about the points-to relationships between the variables.

In the next section, we explain our approach in more detail.

## 3.2 Our Approach

From the discussion in the previous section, we see that our approach uses points-to relation as the abstraction to describe the relationship between locations in the abstract stack. It is based on a structured analysis which approximates the relationships between abstract stack locations. At each program point, we collect the points-to information abstracting relationship between each pair of the abstract stack locations, say `x` and `y`. For such pair, one of the following relationships hold:

**1.** If the content of the stack location `x` definitely points to the stack location `y` at a given program point, we say that `x` **definitely-points-to** `y`. This is denoted as (x,y,$D$). Figure 3.2 shows an example.

**2.** If the content of the stack location `x` possibly points to the stack location `y` at a given program point, we say that `x` **possibly-points-to** `y`. This is denoted as (x,y,$P$). Figure 3.3 shows an example.

**3.** The content of stack location `x` does not point to stack location `y`.

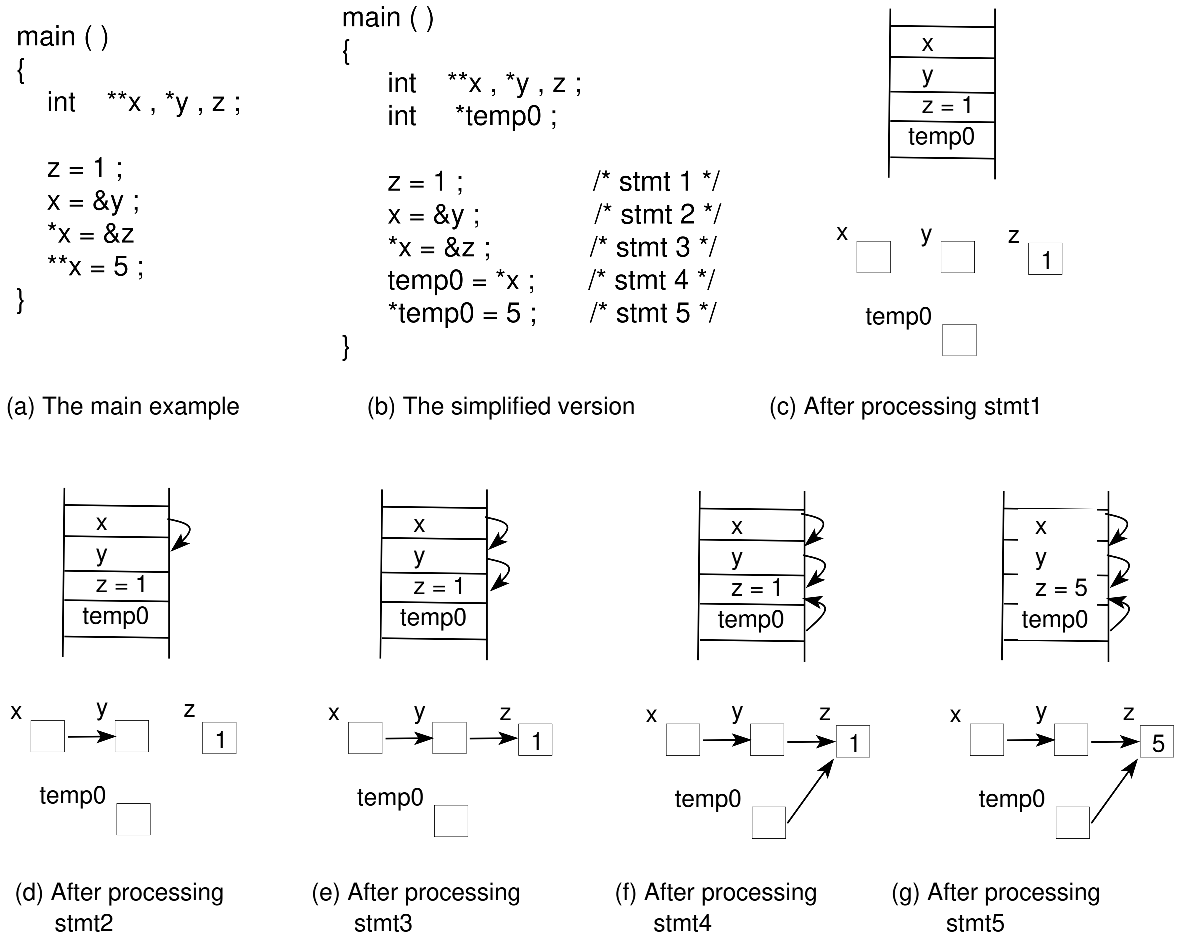


Figure 3.1: A motivating example behind our approach to intraprocedural points-to analysis.

## 3.3 Rules to Convert *Points-to* Information to *Alias* information

As explained briefly in the introduction, traditionally alias pairs were used as an abstraction to determine the relationships between memory locations. In this section, we show how to transform points-to relationships to alias pair relationships. The rules for converting points-to relationships to alias pair relationships are:

1. A relationship of the form (x,y,*rel*), where *rel* is either *D* or *P*, is considered to be equivalent to <$*$x,y>. For example, in the program:

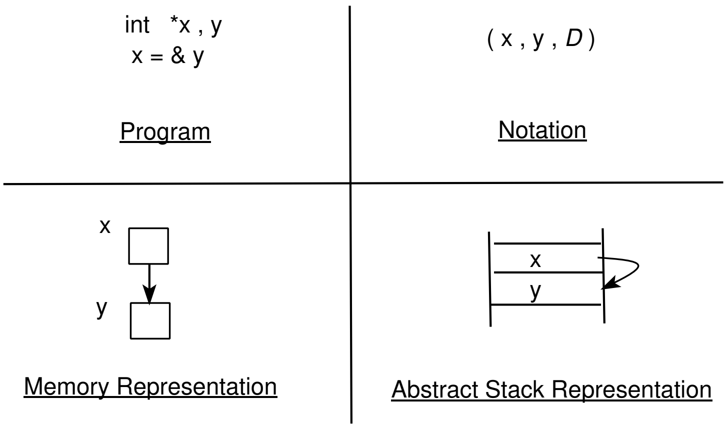


Figure 3.2: An example of a definitely-points-to relationship.

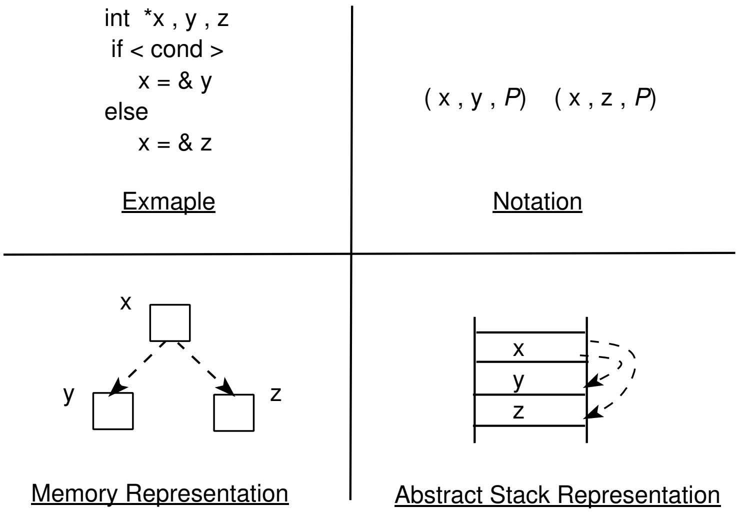


Figure 3.3: An example of a possibly-points-to relationship.

```
int *a, b;
a = &b ;
```

we say that `a` points-to `b`, or equivalently, $*$`a` is aliased to `b`.

2. For each relationship of the form <x,y>, apply the '$*$' as long as the type of variable allows. For example in program:

```
int ***a, **b;
a = &b ;
```

by applying the first rule, we obtain <*a,b>, and by applying the second rule we obtain: <**a, *b>and <***a,**b>.

3. Apply the *commutative rule*[1] and the *transitive closure*[2] on the result of each alias pair obtained so far. This has to be done recursively until no new pairs can be generated. For example in (x,y,*rel*) and (z,y,*rel*), using the first rule we can obtain: <*x,y>, <*z,y>. Then, by applying the commutative rule on the second pair, we get: <*x,y>, <y,*z>. Finally, using the transitive closure results in <*x,*z>.

   Note that the commutative rule does not give a new pair. However, it provides a different way of looking at the pairs which will eventually help us to apply the transitive closure.

Here is a more complicated example which uses all these three rules together:

```
int **d, *a, *b, c
d = &a ;
a = &c ;
b = &c ;

Our result: (d,a,D) (a,c,D) (b,c,D)
Applying rule(1): <*d,a> <*a,c> <*b,c>
Applying rule(2): <*d,a> <**d,*a> <*a,c> <*b,c>
Applying rule(3): <*d,a> <**d,*a> <*a,c> <*b,c> <**d,c>
                  <*a,*b> <**d,*b>
```

We conclude that, in general, the alias information can be easily obtained from our more concisely represented points-to relationships.

In each points-to relationship `(x,y,rel)`, the type of `x` is always a pointer to the type of `y`[3]. As a result of this, the definitely-points-to and possibly-points-to relationships are neither commutative nor transitive.

### 3.3.1 Justification of the Proposed Approach

Our reasons behind using the proposed approach are as follows:

[1] $\mathtt{a}\ op\ \mathtt{b} \Rightarrow \mathtt{b}\ op\ \mathtt{a}$.

[2] $\mathtt{a}\ op\ \mathtt{b} \wedge \mathtt{b}\ op\ \mathtt{c} \Rightarrow \mathtt{a}\ op\ \mathtt{c}$.

[3] An exception to this rule is the special case of *type casting* which will be discussed in Chapter 5.

1. It is straight-forward and easy to understand because it is similar in nature to the way in which variables are actually stored in the stack.
2. It gives accurate results with which we can perform an efficient dependence analysis using the reads and writes to the abstract stack locations.
3. It provides sufficient information about the alias pairs in a compact way.

## 3.4 Notations and Algorithm

The definitions and notations which will be used throughout this thesis are summarized in Table 3.1.

| | | |
|---|---|---|
| $*$ | : | indirect reference in the C-language, e.g., $*$`x` |
| & | : | address operator in the C-language, e.g., & `x` |
| rhs | : | right-hand side of an assignment |
| lhs | : | left-hand side of an assignment |
| $\forall$ | : | for all |
| $\wedge$ | : | and |
| base-type | : | int \| float \| char \| void \| *etc.* |
| ptr-type | : | ptr-type $*$\| base-type $*$ |
| rel1 $\bowtie$ rel2 | : | merge two relationships using the following table: $\begin{array}{c\|\|c\|c} \bowtie & D & P \\ \hline\hline D & D & P \\ \hline P & P & P \end{array}$ |

Table 3.1: Definitions and notations used in the thesis.

In general, each statement is either a basic statement or a compositional control statement. In the intraprocedural analysis, the definitely-points-to relationships are generated/resolved by the basic statements. The data flow merge forced by compositional control statements like loop and conditionals, due to unknown flow of control, causes the appearance of possibly-points-to relationships.

Figure 3.4 gives an overall view of the algorithm for the intra-procedural points-to analysis. The algorithm for basic statements (basic_points_to) is discussed in Section 3.5 (Figure 3.6). The algorithm for compositional control statements (control_points_to) is discussed in Section 3.6 (Figure 3.15).

```
/* Given a basic statement and input information, return the output
 * information (where 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
      return(input) ;
```

Figure 3.4: General algorithm for intraprocedural analysis.

## 3.5 Basic Statements

The simplifying process reduces the many different cases that one may encounter in a typical program (refer to Figure 2.7) to the eight types of basic statements listed in Table 3.2. Case 2, 6 and 7 are discussed in this chapter, case 4 and 8 are discussed in Chapter 4, and case 1 and 3 are discussed in Chapter 5. Note that case 5 does not introduce any points-to information and consequently is not of any concern to us.

| | | | |
|---|---|---|---|
| (1)<br>x = a binop b<br>*x = a binop b | (2)<br>x = y<br>*x = y | (3)<br>x = cast y<br>*x = cast y | (4)<br>f(args) |
| (5)<br>x = unop a<br>*x = unop a | (6)<br>x = *y<br>*x = *y | (7)<br>x = &y<br>*x = &y | (8)<br>x = f(args)<br>*x = f(args) |

Table 3.2: All possible Basic Statements.

The type of the variables plays an important role in our approach to points-to analysis. A basic statement will be studied only if it is involved in an assignment with a pointer type variable. Let us consider the example illustrated in Figure 3.5.

Since `*x` and `*y`, in statement `*x = *y`, are not of pointer type, this statement does not make any difference in the points-to information. Therefore, it is not of concern to us. Now consider the example:

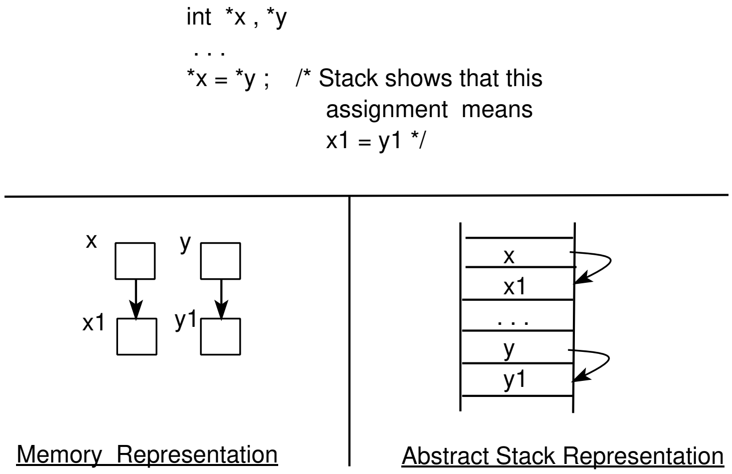


Figure 3.5: An example where no pointer is involved in the assignment statement.

```
int *x, **y ;
...
x = *y ;
```

In this example, since variables `x` and `*y` are of pointer type, after the execution of statement `x = *y`, the points-to information of `x` will be affected (`x` points to the same location as `*y` is pointing to). Therefore, this statement should be considered in the points-to analysis.

We have classified all the cases studied in this chapter according to the basic statements and the type information. These are summarized in Table 3.3.

| lhs \ rhs | &y | y | *y |
|---|---|---|---|
| x | 1 | 2 | 3 |
| *x | 4 | 5 | 6 |

Table 3.3: Basic statements to be studied in this chapter, when lhs (or rhs) is of pointer type.

The algorithm for the basic statements of scalar variables is given in Figure 3.6 (we will discuss arrays and structures in Chapter 5). Each entry in the Table 3.3 has a corresponding rule in the function *basic_points_to*. In each case, we compute: (i) the

```
/* Given a basic statement and input information, return the output
 * information (input.in is the points_to relationships) */
basic_points_to (S,input) =
  if ( ! pointer_type(S))
      return( input ) ;
  else case S of
    /* rule 1 */
    < x = &y > :
      { kill = { (x,x1,rel) | (x,x1,rel) ∈ input };
        gen = { (x,y,D) };
        return( gen ∪ (input − kill) );
      }
    /* rule 2 */
    < x = y > :
      { kill = { (x,x1,rel) | (x,x1,rel) ∈ input };
        gen = { (x,y1,rel) | (y,y1,rel) ∈ input };
        return( gen ∪ (input − kill) );
      }
    /* rule 3 */
    < x = *y > :
      { kill = { (x,x1,rel) | (x,x1,rel) ∈ input };
        gen = { (x,y2,rel1 ⋈ rel2) | (y,y1,rel1), (y1,y2,rel2) ∈ input};
        return( gen ∪ (input − kill) );
      }
    /* rule 4 */
    < *x = &y > :
      { kill = { (x1,x2,rel) | (x,x1,D), (x1,x2,rel) ∈ input };
        change = (input − { (x1,x2,D) | (x,x1,P), (x1,x2,D) ∈ input})
                     ∪ { (x1,x2,P) | (x,x1,P), (x1,x2,D) ∈ input};
        gen = { (x1,y,rel) | (x,x1,rel) ∈ input};
        return( gen ∪ (change − kill) );
      }
    /* rule 5 */
    < *x = y > :
      { kill = { (x1,x2,rel) | (x,x1,D), (x1,x2,rel) ∈ input };
        change = (input − { (x1,x2,D) | (x,x1,P), (x1,x2,D) ∈ input})
                     ∪ { (x1,x2,P) | (x,x1,P), (x1,x2,D) ∈ input};
        gen = { (x1,y1,rel1 ⋈ rel2) | (x,x1,rel1), (y,y1,rel2) ∈ input};
        return( gen ∪ (change − kill) );
      }
    /* rule 6 */
    < *x = *y > :
      { kill = { (x1,x2,rel) | (x,x1,D), (x1,x2,rel) ∈ input };
        change = (input − { (x1,x2,D) | (x,x1,P), (x1,x2,D) ∈ input})
                     ∪ { (x1,x2,P) | (x,x1,P), (x1,x2,D) ∈ input};
        gen = { (x1,y2,rel1 ⋈ rel2 ⋈ rel3) | (x,x1,rel1), (y,y1,rel2), (y1,y2,rel3) ∈ input};
        return( gen ∪ (change − kill) );
      }
    /* any other simple statement */
    < _ > :  return(input);
```

Figure 3.6: Algorithm of basic statements for scalar variables.

points-to information which gets killed, (ii) the changes in the points-to information where some definitely-points-to relationship becomes possibly-points-to, and, (iii) the points-to information which is generated. Then, we return *input* (the original set of points-to relationships) while updating the information by removing the killed information, incorporating the changes, and adding the newly created information. In the next sections, we examine each case in detail. To facilitate the understanding, we use a simple pointer type, `int*` or `int**`.

### 3.5.1 Case 1

This is the case denoted as *rule1* in Figure 3.6.

```
int *x, y ;
...
x = &y ;
```

The rules applied in this case are as follows:

kill = { (x,x1,rel) | (x,x1,rel) $\in$ input }
gen = { (x,y,D) }
**return**( gen $\cup$ (input - kill) )

Referring to Figure 3.7, we see that after processing the statement `x = &y`, `x` should point to the location `y`. This means that all the relations in which `x` points to a variable should be removed from the points-to information (*kill* set). Then, `x` definitely-points-to `y` is the unique element of the newly generated set (*gen* set). Finally, by removing the *kill* set and adding the *gen* set to *input* set (the original set of information), we obtain the new set of information.

### 3.5.2 Case 2

This case is denoted as *rule2* in Figure 3.6.

```
int *x, *y ;
...
x = y ;
```

The rules applied in this case are as follows:

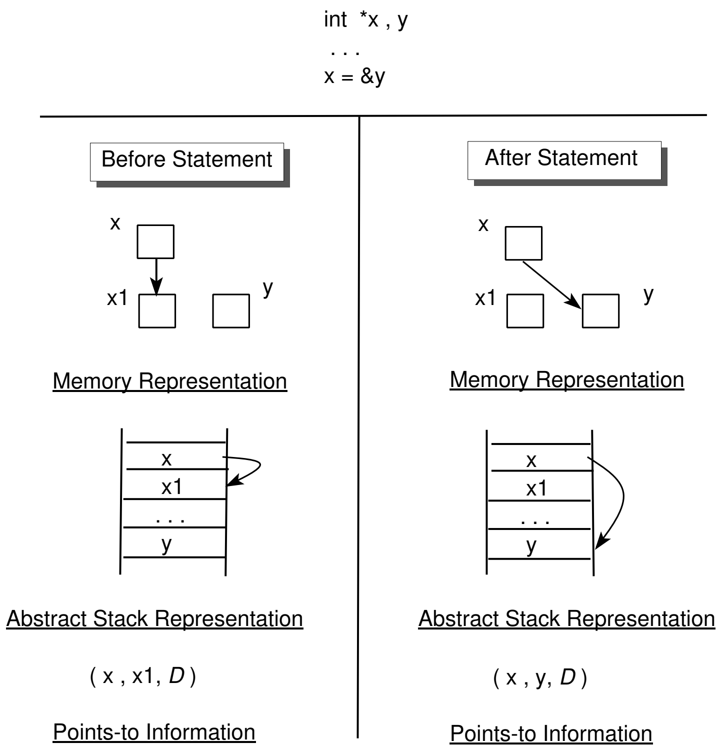


Figure 3.7: An example of rule 1 in the basic statements.

kill = { (x,x1,rel) | (x,x1,rel) $\in$ input }
gen = { (x,y1,rel) | (y,y1,rel) $\in$ input }
**return**( gen $\cup$ (input - kill) )

After processing the statement `x = y`, `x` gets changed. This is shown in Figure 3.8. Therefore, we should remove all the relations in which `x` points to a variable `x1` from the points-to information (*kill* set). Then make `x` point to the same location as `y` is pointing to (*gen* set). The new set of information can be obtained by removing the *kill* set from the *input* set and adding the *gen* set.

A more complicated, self explanatory example is given in Figure 3.9

### 3.5.3 Case 3

This case is denoted as *rule3* in Figure 3.6.

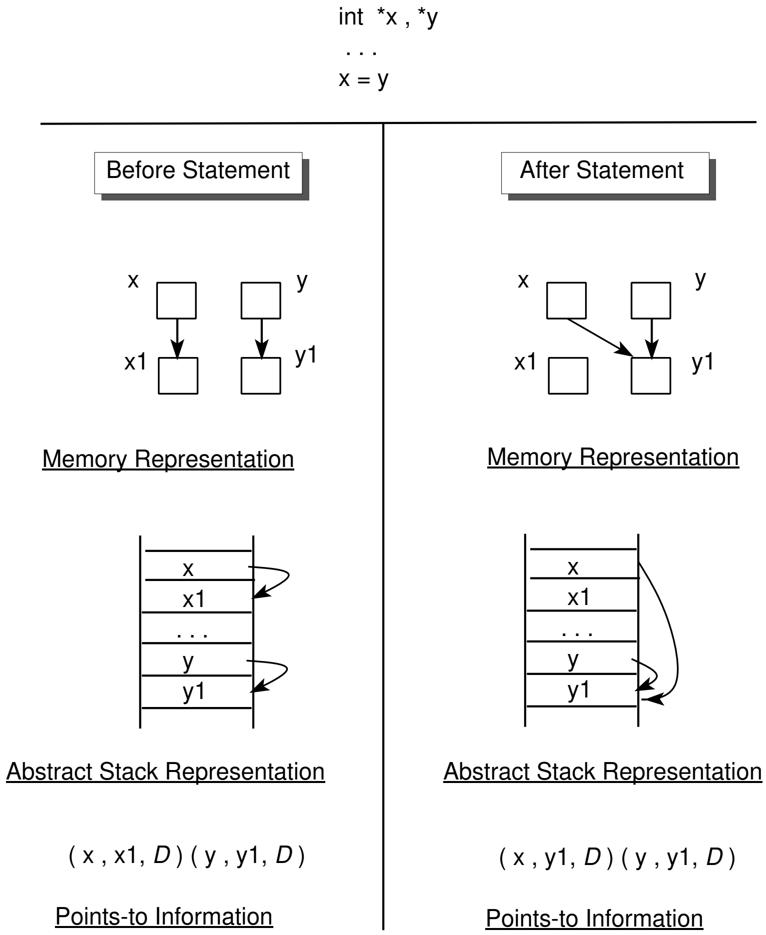


Figure 3.8: An example of rule 2 in the basic statements.

```
int *x, **y ;
...
x = *y ;
```

The rules applied in this case are as follows:

kill = { (x,x1,rel) | (x,x1,rel) ∈ input }

gen = { (x,y2,rel1 $\bowtie$ rel2) | (y,y1,rel1), (y1,y2,rel2) ∈ input}

**return**( gen ∪ (input - kill) )

Referring to Figure 3.10, after processing the statement `x = *y`, `x` should point to the same location as `*y` is pointing to. This means that all the relations in which `x` points to a variable, should be removed from the points-to information (*kill* set) and `x`

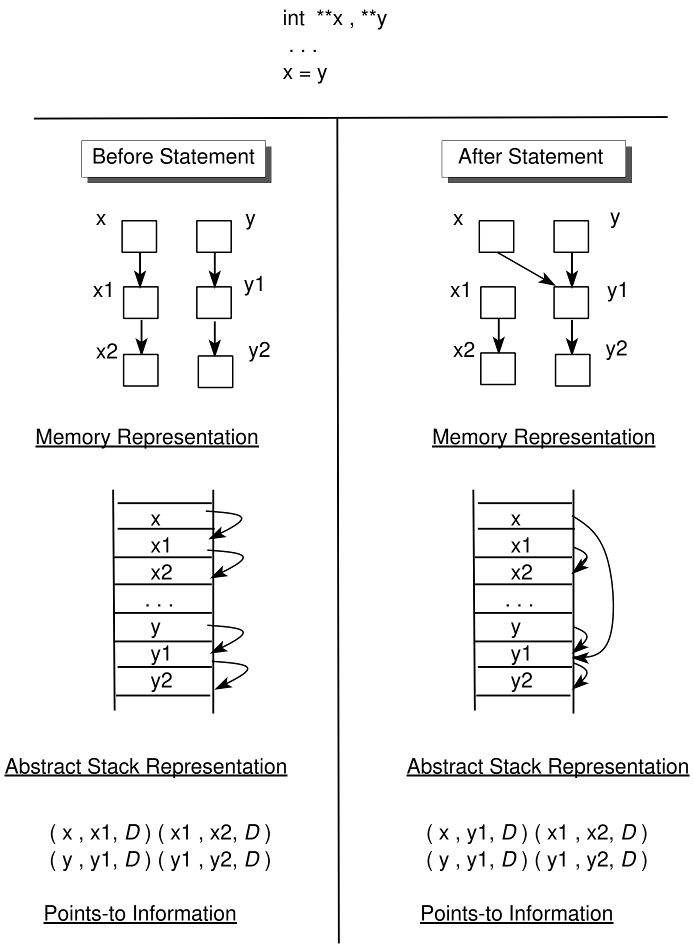


Figure 3.9: A more complicated example of rule 2 in the basic statements.

should point to the same location as $*$`y` is pointing to. If `y` definitely-points-to location `y1`, and `y1` definitely-points-to location `y2`, we can say that `x` definitely-points-to `y2` (because it is guaranteed that $*$`y` points to `y2`). If one of these relationships is possibly-points-to, then `x` possibly-points-to `y2` (because it is not guaranteed that $*$`y` points to `y2`). The new set of information can be obtained by removing the *kill* set from the original set of information and adding the *gen* set.

### 3.5.4 Case 4

This case is denoted as *rule4* in Figure 3.6.

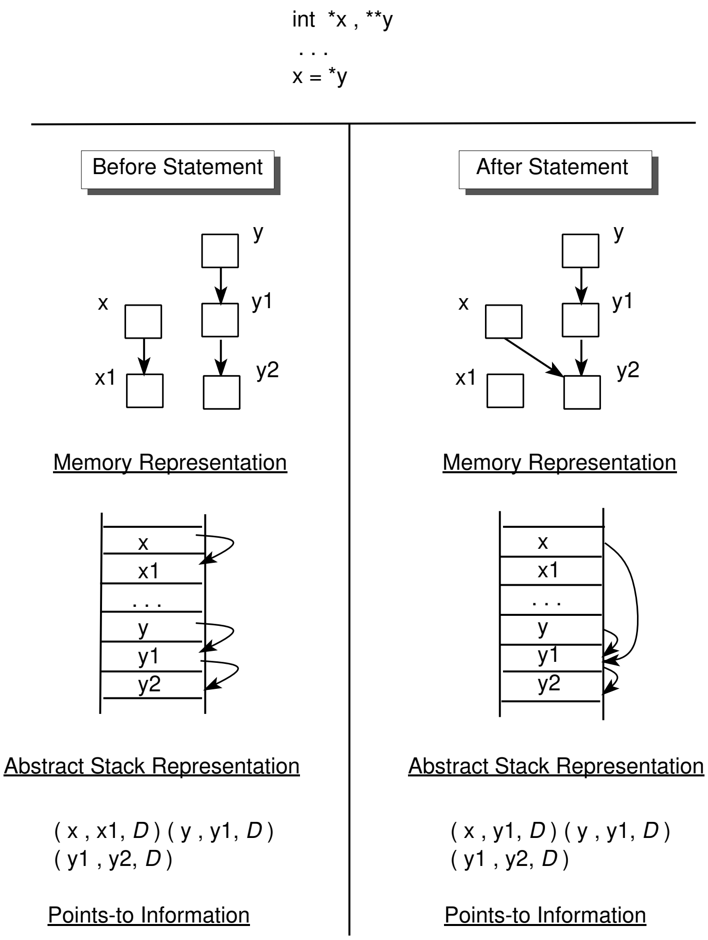


Figure 3.10: An example of rule 3 in the basic statements.

```
int **x, y ;
...
*x = &y ;
```

The rules applied in this case are as follows:

kill = { (x1,x2,rel) | (x,x1,D), (x1,x2,rel) ∈ input }
gen = { (x1,y,rel) | (x,x1,rel) ∈ input}
changed_input = (input - { (x1,x2,D) | (x,x1,P), (x1,x2,D) ∈ input})
∪ { (x1,x2,P) | (x,x1,P), (x1,x2,D) ∈ input}
**return**( gen ∪ (changed_input - kill) )

Referring to Figure 3.11, after processing the statement `*x = &y`, if `x` definitely-points-to `x1`, the modification of location `x1` is guaranteed and we remove the relations in which `x1` points to a variable from points-to information (*kill* set). If `x` possibly-points-to `x1`, changing of `x1` is not guaranteed. In this case, we change all the definitely-points-to relationship of `x1` into possibly-points-to (*changed_input* set). If `x` definitely-points-to `x1`, then `x1` definitely-points-to `y`. But, if `x` possibly-points-to `x1`, then `x1` possibly-points-to `y`. So, the way that `x1` points to `y` depends on the way that `x` points to `x1` (*gen* set). Finally, by removing the *kill* set from the *changed_input* set and adding the *gen* set to it, we obtain the final information.

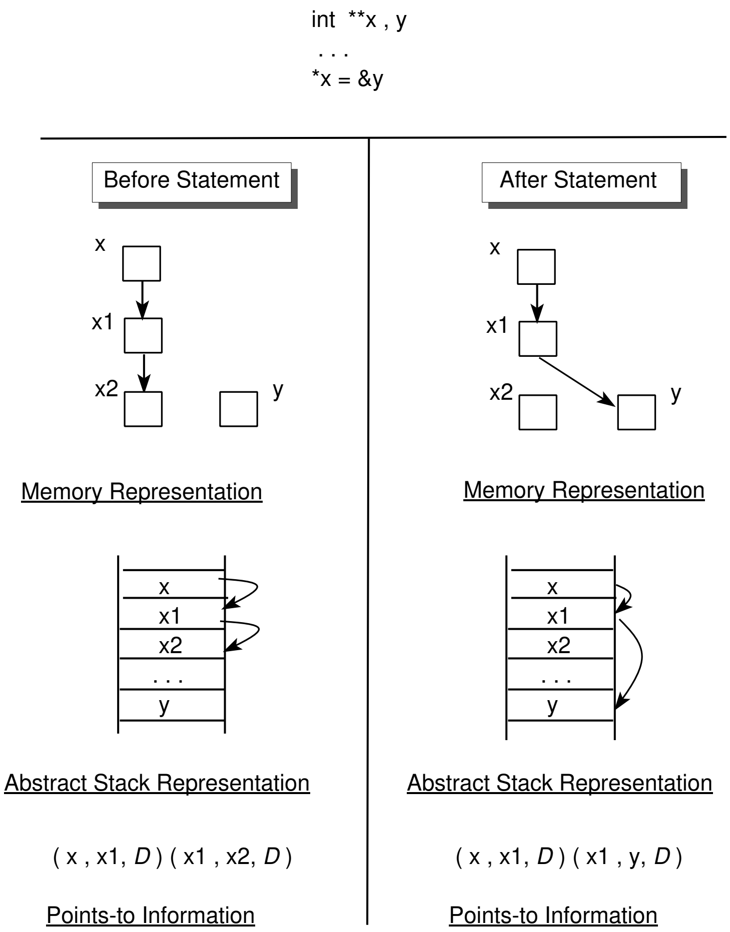


Figure 3.11: An example of rule 4 in the basic statements.

### 3.5.5 Case 5

This case is denoted as *rule5* in Figure 3.6.

```
int **x, *y ;
...
*x = y ;
```

The rules applied in this case are as follows:

kill = { (x1,x2,rel) | (x,x1,D), (x1,x2,rel) $\in$ input }
gen = { (x1,y1,rel1 $\bowtie$ rel2) | (x,x1,rel1), (y,y1,rel2) $\in$ input}
changed_input = (input - { (x1,x2,D) | (x,x1,P), (x1,x2,D) $\in$ input})
$\cup$ { (x1,x2,P) | (x,x1,P), (x1,x2,D) $\in$ input}
**return**( gen $\cup$ (changed_input - kill) )

Figure 3.12 and Figure 3.13 help in understanding the analysis. After processing the statement `*x = y`, the *kill* set and the *changed_input* set are computed in the same way as in case 4. If `x` has a points-to relationship *rel1* with `x1` and `y` has a points-to relationship *rel2* with `y1`, then `x1` will have a points-to relationship of the form $rel1 \bowtie rel2$ with `y1`. This relationship says that `x1` definitely-points-to `y1` if *rel1* and *rel2* are of definitely-points-to type, otherwise, `x1` possibly-points-to `y1`. Finally, by removing the *kill* set from the *changed_input* set and adding the *gen* set to it we obtain the final information.

### 3.5.6 Case 6

This case is denoted as *rule6* in Figure 3.6.

```
int **x, **y ;
...
*x = *y ;
```

The following rules are applied in this case:

kill = { (x1,x2,rel) | (x,x1,D), (x1,x2,rel) $\in$ input }
gen = { (x1,y2,rel1 $\bowtie$ rel2 $\bowtie$ rel3) | (x,x1,rel1), (y,y1,rel2), (y1,y2,rel3)
$\in$ input}

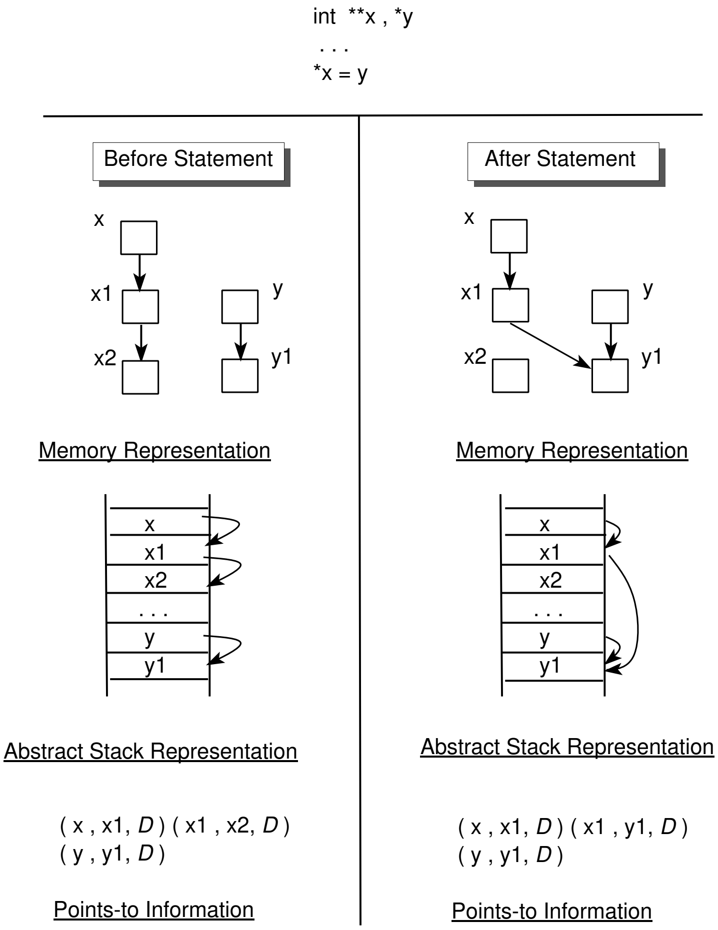


Figure 3.12: An example of rule 5 in the basic statements.

changed_input = (input - { (x1,x2,D) | (x,x1,P), (x1,x2,D) $\in$ input})
$\cup$ { (x1,x2,P) | (x,x1,P), (x1,x2,D) $\in$ input}
**return**( gen $\cup$ (changed_input - kill) )

Referring to Figure 3.14, after processing the statement `*x=*y`, the *kill* set and the *changed_input* set are computed in the same way as in case 4. If `x` has a points-to relationship *rel1* with `x1`, `y` has a points-to relationship *rel2* with `y1`, and `y1` has a points-to relationship *rel3* with `y2`, then `x1` will have a points-to relationship of the form $\{rel1 \bowtie rel2 \bowtie rel3\}$ with `y2`. This relationship says that `x1` definitely-points-to `y2` if $rel1, rel2$ and $rel3$ are all of definitely-points-to type, otherwise, `x1` possibly-points-to `y2`. Finally, by removing the *kill* set from the *changed_input* set, and adding the *gen* set to it, we obtain the final information.

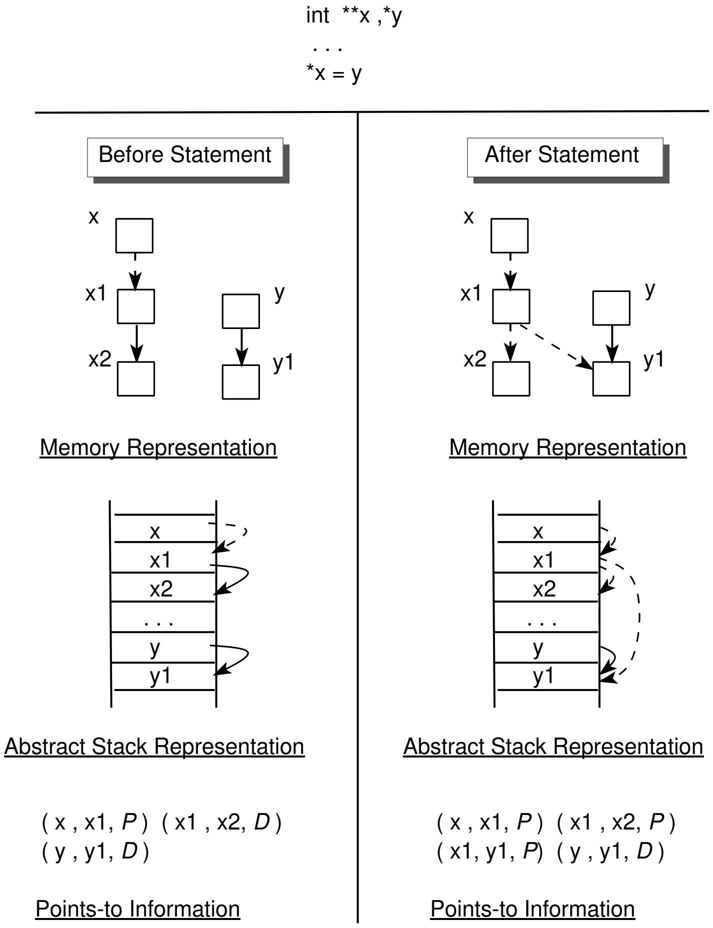


Figure 3.13: An example of rule 5 that contains possibly points-to relationship in the basic statements.

## 3.6 Compositional Control Statements

In the previous section, we presented our analysis method for simple statements. In this section, we extend our method to programs that include control statements. SIMPLE supports the following control statements: `if, for, while, do-while, switch, continue,` and `break` (note that `return` statement appears in interprocedural analysis, therefore it is explained in Chapter 4). To simplify the explanation, we first consider compositional control statements without the presence of `break` and `continue` statements. Then, the more general case containing `break` and `continue` statements is studied. To support control statements such as conditionals and loops,

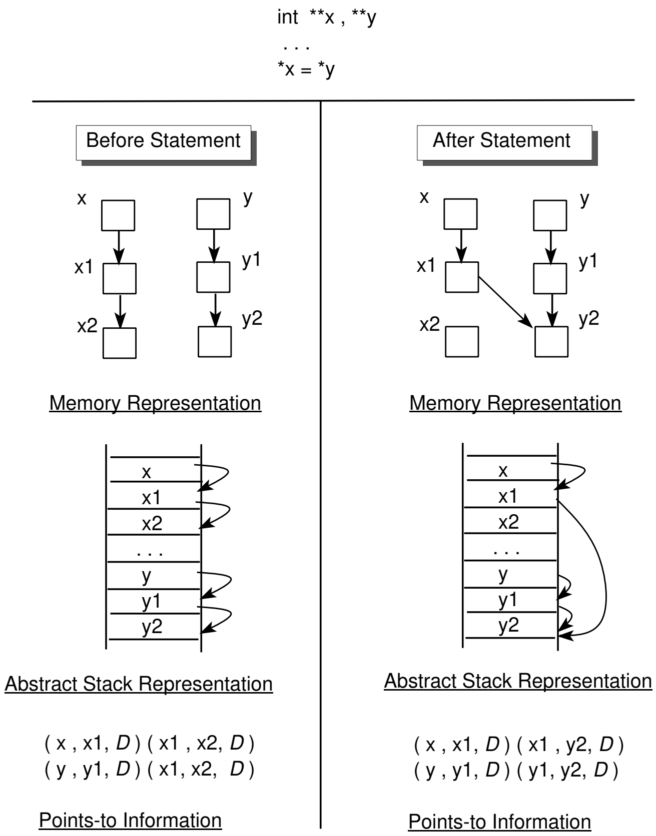


Figure 3.14: An example of rule 6 in the basic statements.

we need to define the following two concepts: (i) merge between sets, and (ii) fixed-point computation.

The concept of merge between definitely-points-to and possibly-points-to relationships was defined in Table 3.1. Here we extend this definition to the case of merging two sets of relationships. This is denoted as **Merge(S1, S2)** where S1 and S2 are the sets of the relationships to be merged. The definition is as follows:

Definite-set $= \{(x,y,D) \mid (x,y,D) \in (S1 \cap S2) \}$

Possible-set $= \{(x,y,P) \mid (x,y,rel) \in (S1 \cup S2) \wedge (x,y,D) \notin$ Definite-set $\}$

Merge(S1, S2) $=$ Definite-set $\cup$ Possible-set

where 'rel' is either D or P. Here is an example:

S1 = { (a,a1,D), (b,b1,D), (c,c1,D) }
S2 = { (a,a1,D), (b,b1,P) }

Definite-set = { (a,a1,D) }

Possible-set = { (b,b1,P), (c,c1,P) }

Merge(S1, S2) = { (a,a1,D), (b,b1,P), (c,c1,P) }

Note that the relationship (a,a1,D) is in the Definite-set because 'a' and 'a1' have a definitely-points-to relationship in both S1 and S2. The relationship (b,b1,P) is in the Possible-set because 'b' and 'b1' have definitly-points-to relationship in S1 but they have possibly-points-to relationship in S2. The relationship (c,c1,P) is in the Possible-set because the relationship between 'c' and 'c1' is in S1 and not in S2.

The concept of **fixed-point** computation is defined in conjunction with the analysis of loops and recursion. We say that a fixed-point is reached when analysis of the loop-body in two successive iterations does not result in any new information. An example of fixed-point will be given later in our discussion concerning `while` loop.

Figure 3.15 gives an outline of the algorithm for the analysis of the control statements. Each of the function calls will be discussed in detail in the rest of this section. Note that in SIMPLE, the expressions in conditionals and loops are side-effect free, so we ignore them during the analysis. For the same reason, the expression of the `switch` statement is also ignored.

### 3.6.1 Points-to Analysis without `break` and `continue`

<u>`if` statement:</u>

For the `if` statement, we propagate the input to both the then-body and the else-body. After processing both parts, we merge the information to get the new input set. In the absence of the else-body, the output of else-body is the same as the input to the `if` statement. Therefore, the output of the then-body is merged with the input to the `if` statement. The corresponding algorithm (denoted as 'process_if' in Figure 3.15), together with its graph representation, is given in Figure 3.16. Note that the parameter `in` is a structure that contains the points-to information and other fields which are useful during the analysis (e.g. the fields related to break and continue that is explained in Section 3.6.2).

<u>`while` statement:</u>

In general, the body of a while-loop, depending on the while-loop condition, is executed for `n = 0,1,2,...` times. As we are not sure about the exact number of the iterations, we have to approximate the output of our analysis. The first approximation corresponds to the initial stage before starting the iterations. In this case, the output

```
/* Given a basic statement and input information, return the output
 * information (input.in is the points_to relationships) */
control_points_to (S,input) =
  case S of
    < IF(cond,s_then,s_else) > :
       return(process_if(cond,s_then,s_else,input));

    < SWITCH(expr,caselist) > :
       return(process_switch(expr,caselist,input));

    < WHILE(cond,s_body) > :
        return(process_while(cond,s_body,input));

    < DO(s_body,cond) > :
        return(process_do(s_body,cond,input));

    < FOR(s_init,cond,s_iter,s_body) > :
        return(process_for(s_init,cond,s_inter,s_body, input));

    < BREAK > :
        return(process_break(input)) ;

    < CONTINUE > :
        return(process_continue(input)) ;

    < _ > :
        return(input) ;
```

Figure 3.15: Algorithm of the compositional control statements.

```
process_if(cond, then_body, else_body, in) {
    out1 = points_to(then_body, in) ;
    out2 = points_to(else_body, in) ;
    return(merge_info(out1, out2)) ;
}

merge_info(in1, in2) {
    out = in1 ;
    /* merge the points–to information
     * field of the two given input */
    out.in = Merge(in1.in, in2.in) ;
    return(out) ;
}
```

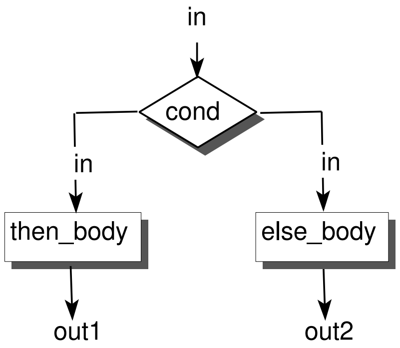


Figure 3.16: Algorithm of the `if` statement for points-to analysis.

is the same as the input. The second approximation is obtained by merging the following sets of information:

- the previous approximation,
- the output obtained from processing the while-body with the previous approximation as the input.

This process is repeated until a fixed-point is reached. The corresponding algorithm (denoted as 'process_while' in Figure 3.15), together with its graph representation, is shown in Figure 3.17.

```
process_while(cond, body, in) {
    {
        last_in  = in ;
        out = points_to(body, in) ;
        in = merge_info(in, out) ;
    } while (last_in != in) ;

    result = in ;
    return(result) ;
}
```

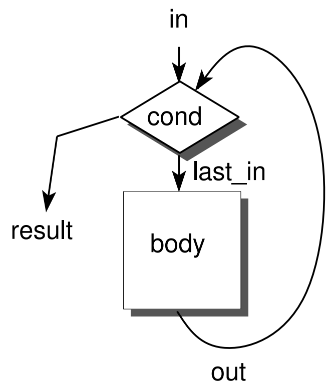


Figure 3.17: Algorithm of the `while` statement for points-to analysis.

Following example demonstrates the process of finding the fixed-point of a while-loop. The information represented for each statement, is after processing the statement. The following shows the fixed-point calculation for a while-loop:

```
int *a, *b, c, d ;                     First approximation
                                       -------------------

b = &c ;         /* s1 */          {(b,c,D)}
while (cond){    /* s2 */          {(b,c,D)}
    a = b ;      /* s3 */          {(b,c,D), (a,c,D)}
    b = &d ;     /* s4 */          {(b,d,D), (a,c,D)}
}                /* s5 */          {(b,c,P), (b,d,P), (a,c,P)}

              Second approximation
              --------------------

/* s1 */  {(b,c,D)}
/* s2 */  {(b,c,P), (b,d,P), (a,c,P)}
/* s3 */  {(b,c,P), (b,d,P), (a,c,P), (a,d,P)}
/* s4 */  {(b,d,D), (a,c,P), (a,d,P)}
/* s5 */  {(b,c,P), (b,d,P), (a,c,P), (a,d,P)}

              Third approximation
              -------------------

/* s1 */  {(b,c,D)}
/* s2 */  {(b,c,P), (b,d,P), (a,c,P), (a,d,P)}
/* s3 */  {(b,c,P), (b,d,P), (a,c,P), (a,d,P)}
/* s4 */  {(b,d,D), (a,c,P), (a,d,P)}
/* s5 */  {(b,c,P), (b,d,P), (a,c,P), (a,d,P)}
```

The input to the loop at each iteration is the set represented at statement `s2` and the corresponding output is the set represented at statement `s5`. We observe that after the third iteration the information does not change anymore (fixed-point of the loop). Therefore, the final points-to information is the information obtained at this stage.

<u>`do-while` statement</u>

The `do-while` statement is similar to the while-loop with the difference that the body of the `do-while` is processed for `n = 1,2,..` times. This means that the body will be processed at least once. Therefore, the first approximation to the `do-loop` is what we have after analyzing the loop-body once. The corresponding algorithm (denoted as 'process_do' in Figure 3.15), together with its graph representation, is given in Figure 3.18.

```
process_do(cond, body, in) {
     out = points_to(body, in) ;
     in = out ;
     {
          last_in   = in ;
          out = points_to(body, in) ;
          in = merge_info(in, out) ;
     } while (last_in != in) ;

     result = in ;
     return(result) ;
}
```

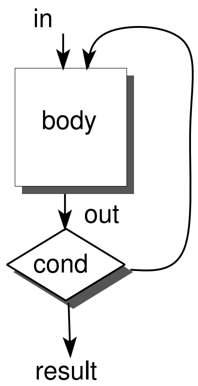


Figure 3.18: Algorithm of the `do-while` statement for points-to analysis.

<u>`for` statement:</u>

The `for` statement is also similar to the `while` statement. The difference is that the initial statement has to be processed before processing the body and the iteration statement has to be processed after processing the body. The corresponding algorithm (denoted as 'process_for' in Figure 3.15), together with its graph representation, is given in Figure 3.19. If it can be determined at compile time that the for loop will iterate at least once, then a more accurate answer might be obtained by processing the body once before computing the fixed point.

```
process_for(init_stmt, cond, iter_stmt, body, in) {
     in = points_to(init_stmt, in) ;
     {
          last_in   = in ;
          out1 = points_to(body, in) ;
          out2 = points_to(iter_stmt, out1) ;
          in = merge_info(in, out2) ;
     } while (last_in != in) ;

     result = in ;
     return(result) ;
}
```

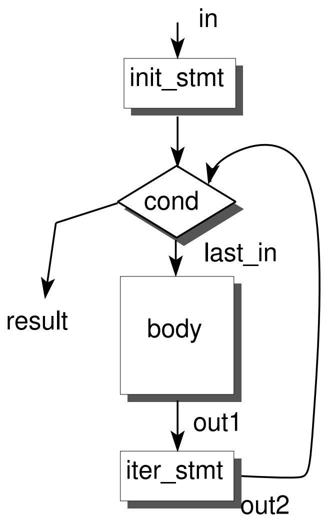


Figure 3.19: The algorithm of `for` statement for points-to analysis.

### 3.6.2 Points-to Analysis with `break` and `continue`

In this section, we present our analysis for programs with `break` and `continue` statements. First, let us recall the semantics of `break` and `continue`. Execution of a `break` statement terminates the execution of the closest `while, do-while, for` or `switch` statement. Then, the program control is immediately transferred to the point just after the body of the corresponding statement.

Execution of a `continue` statement terminates the execution of the body of the closest `while, do-while`, or `for` statement. Then, the program control is immediately transferred to the beginning of the body and the execution continues from that point with a re-evaluation of the loop condition. In the case of the for-loop, the iteration expression is also re-evaluated.

Note that the `continue` statement has no interaction with the `switch` statement. The `continue` statement within the body of a `switch` statement actually belongs to the closest `while, do-while`, or `for` loop.

In our analysis, we handle `break` and `continue` statements as follows: We define two structures, called the `break-list` and `continue-list`, to collect the points-to information at `break` points and at `continue` points encountered during processing the `while, do-while, for`, and `switch` statements. Whenever we encounter a `break/continue` statement, we save the corresponding points-to information in `break-list/continue-list` and pass BOTTOM as the output where BOTTOM denotes no information. Propagating BOTTOM corresponds to taking paths in the program that would never occur in any execution (dead-code). Any statement with the input BOTTOM, produces BOTTOM as the output. The merge rule for BOTTOM is as follows:

$$\text{Merge(S1, BOTTOM)} = \text{Merge(BOTTOM, S1)} = \text{S1}$$

The intuition behind this rule is that a path resulting in BOTTOM corresponds to an impossible execution path and, consequently, can be ignored in the merge. This fact is represented in Figure 3.20. The input to the `if` is called `in`, and the result of `then` statements (`S1` and `S2`) is represented as `out1` and the result of `else` statement (`S3`) is represented as `out2`. After processing the `break` statement, BOTTOM is passed as the result. This path is not a valid path, therefore only the path coming from `S3` can reach to this point. That is why the input to `S4` can be only `out2`. In other words, that is why the merge of `out2` and BOTTOM is `out2`.

In the following, we explain how to use the information gathered in the `break-list`

```
main() {
    ...
    while(cond1) {
        if (cond2)
        {
            S1 ;
            S2 ;
            break ;
        }
        else{
            S3 ;
        }
        S4 ;
    }
        ...
}
```

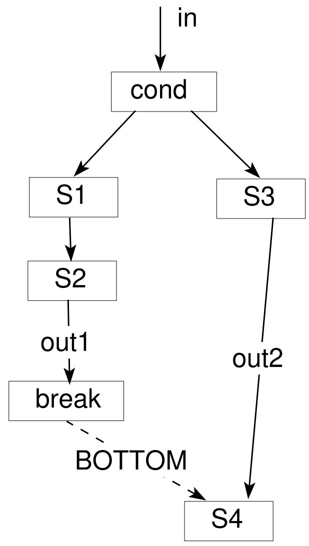


Figure 3.20: An example of `if` statement with a `break` statement.

and in the `continue-list` in our analysis. Let us first consider the `continue-list`. Recall that a `continue` statement takes the program control back to the beginning of the corresponding loop. This means that we should merge the following three sets to get a new approximation (each of these sets can be a new input to the loop):

- the previous approximation (the first approximation for `while` statement is the input set, for `for` statement is the result after processing the initial statement, and for `do-while` is the result after the first process to its body.).
- the output obtained from processing of the loop-body with the previous approximation as the input.
- all sets of information stored in the `continue-list`. Note that each of these sets is a potential input to the loop and corresponds to a path in the body that was terminated by a continue statement.

This process is repeated until a fixed-point is reached.

Unlike the `continue-list`, the `break-list` does not participate in the fixed-point calculation. In the case of the `break-list`, we merge the final approximation after reaching a fixed-point (for loops) with each set of the information contained in the `break-list`. The reason behind this approach is that each set of information in the `break-list` is a potential output of the statement.

The corresponding algorithms for `break` and `continue` (denoted as 'process_break' and 'process_continue' in Figure 3.15) are:

```
process_break(in) {
    /* break_lst field contain the merge of all the information
     * reaching to the break statements related to a loop */
    in.break_lst = merge_info(in.break_lst, in) ;
    return(BOTTOM) ;
}

process_cont(in) {
    /* cont_lst field contain the merge of all the information
     * reaching to the continue statements related to a loop */
    in.cont_lst = merge_info(in.cont_lst, in) ;
    return(BOTTOM) ;
}
```

Figures 3.21 to 3.24 illustrate the algorithm together with a graph representation of the `while`, `do-while`, `for`, and `switch` statements containing `break` and `continue`.

## 3.7 Summary

In this chapter, we discussed the intraprocedural points-to analysis for scalar variables. The analysis rules for basic statements and compositional control statements were described in a structured and compositional way. We also described a systematic strategy to handle the analysis of control statements with `break` and `continue` features embedded. The algorithm devised for points-to analysis can be used as a model to design structured algorithms for other flow analyses. In the next chapter, the interprocedural aspects of calculating points-to information are discussed.

```
process_while(cond, body, in) {
      {
            last_in   = in ;
            out = points_to(body, in) ;
            /* get the continue list */
            cont_lst = in.cont_lst ;
            out1 = merge_info(out, cont_lst) ;
            in = merge_info(in, out1) ;
      } while (last_in != in) ;

      break_lst = in.break_lst ;
      result = merge_info(in, break_lst) ;
      return(result) ;
}
```

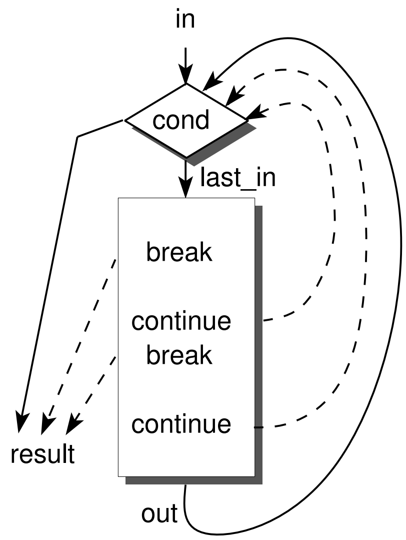


Figure 3.21: Algorithm of the `while` statement in the presence of `break` and `continue` statements.

```
process_do(cond, body, in) {
      /* the body is processed at least once */
      out = points_to(body, in) ;
      /* get the continue list */
      cont_lst = in.cont_lst ;
      in = merge_info(out, cont_lst) ;
      {
            last_in   = in ;
            out = points_to(body, in) ;
            /* get the continue list */
            cont_lst = in.cont_lst ;
            out1 = merge_info(out, cont_lst) ;
            in = merge_info(in, out1) ;
      } while (last_in != in) ;

      break_lst = in.break_lst ;
      result = merge_info(in, break_lst) ;
      return(result) ;
}
```

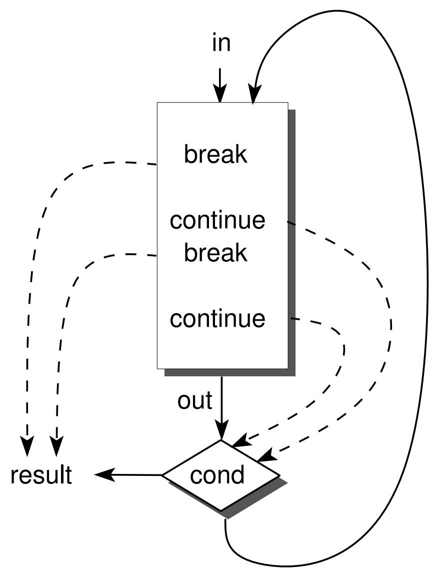


Figure 3.22: Algorithm of the `do_while` statement in the presence of `break` and `continue` statements.

```
process_for(init_stmt, cond, iter_stmt, body, in) {
    in = points_to(init_stmt, in) ;
    {
        last_in   = in ;
        out1 = points_to(body, in) ;
        /* get the continue list */
        cont_lst = in.cont_lst ;
        out2 = merge_info(out1, cont_lst) ;
        out3 = points_to(iter_stmt, out2) ;
        in = merge_info(in, out3) ;
    } while (last_in != in) ;

    break_lst = in.break_lst ;
    result = merge_info(in, break_lst) ;
    return(result) ;
}
```

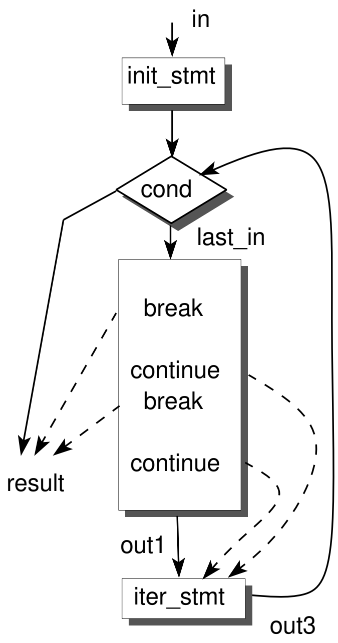


Figure 3.23: Algorithm of the `for` statement in the presence of `break` and `continue` statements.

```
process_switch(expr, caselist, in) {
    for each case in caselist do
        tmp = points_to(case_body, in) ;

    break_lst = in.break_lst ;
    result = break_lst ;
    return(result) ;
}
```

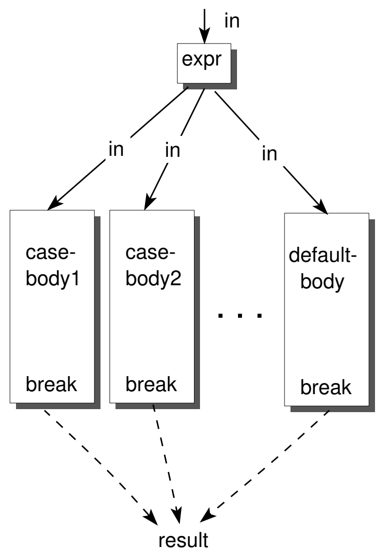


Figure 3.24: Algorithm of the `switch` statement in the presence of `break` and `continue` statements.