Chapter 9

Related Work

As mentioned in the introduction, two variables a and b are considered to be aliased at a program point `p`, if they refer to the same memory location. This is represented as $<a,b>$. There are two general approaches to solve the aliasing problem: (i) automatic alias analysis in which no input is needed from the programmer and alias analysis is done by the compiler during the optimization phase, and (ii) directive code annotations in which the programmer provides some information about the properties of pointer-based data structures to enable powerful compiler optimizations [HHN92a, HHN92b]. In this thesis, we have focused on the first approach.

In this chapter, we first give the historical work that has been done in the area of automatic alias analysis, in Section 9.1. Then, we focus on the comparison of our work with the latest work that has been done on interprocedural aliasing in the presence of pointers by Landi and Ryder [Lan92, LR92], in Section 9.2.

9.1 Historical Background on Alias Analysis

An alias analysis is needed for different cases in a programming language. Figure 9.1 shows a general division of these cases.

Traditionally, the alias analysis methods have focused on calculating the aliases generated by the association of actual arguments with call-by-reference parameters. This sort of aliasing is characteristic of languages like Fortran and has the following important properties: (i) only call-sites contribute to the creation of aliases, and (ii) an alias relation valid at the entry of a procedure holds throughout its body. Appropriate techniques for computing such aliases have been developed and described [Bar78,

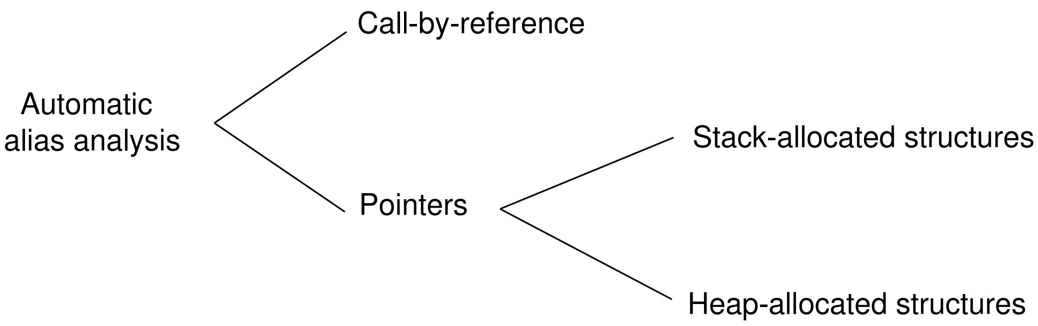


Figure 9.1: A general division of alias analysis.

Ban79, Mye81, Coo85, CK89]. Most of these analyses collect the alias information in two passes over the program: (i) an introductory pass to compute all the obvious aliases generated at call-sites, (ii) a propagation pass to propagate the aliases through the call-graph of the program.

Unlike the call-by-reference mechanism, the presence of pointers considerably complicates the problem of alias computation. This is due to the following factors:

- In addition to the call-sites, the alias information can be affected by any program statement modifying a pointer variable.
- In the presence of multi-level pointers, a function can change the alias information of pointer type variables that are not in its scope. The indirect swap of two-level pointer type variables (given in Section 4.2) is an example of this case.

Further, there are number of features particular to the C-language that make this problem even harder. Such features include: pointer arithmetic, pointers to arrays, and type casting. The complications introduced by these features have been explained in Chapter 6.

As shown in Figure 9.1, pointers can be classified into two broad categories. The basis for this categorization is the location of the pointer target in the memory organization. A pointer is categorized as stack-directed, if it points to a memory object resident on the stack. Similarly, a pointer is classified as heap-directed, if it points to an object allocated in the heap. Stack-directed pointers have the nice property that their target always possesses a unique compile-time name. This does not hold for heap-directed pointers as all the objects in the heap are anonymous. In this thesis, we have focused on the analysis of stack-directed pointers, with our points-to abstraction designed to exploit their special property. A good deal of work has been done on the analysis of heap-directed pointers [JM81, JM82, LH88, HN90, CWZ90, HHN92a]. We now compare our work with the previous works keeping this distinction in mind.

Weihl [Wei80] gave an algorithm for interprocedural data-flow analysis in the presence of pointers. He considers programs with procedure variables and label variables. Therefore, the complete control flow information is not available to him. He first calculates the alias information generated by each relevant program statement, independent of other statements. Then, he performs a transitive closure on this information to obtain the set of aliases that can be valid in the program under analysis. This information is not *program-point-specific* (is not related to a specific point in the program), but is *program-specific* (is related to the whole program). The alias information computed by such a flow-free analysis would be very imprecise. Consider the following example:

```
int *a, b, c ;

a = &b ;
a = &c ;
```

Using Weihl's technique, one gets the alias pairs <∗a,b> and <∗a,c> to be valid for this program. Clearly, the first pair does not hold at the end of the program segment, but this can not be recognized by his analysis. Further, he does 1-limiting[1].

Chow and Rudmik [CR82] also developed an alias analysis algorithm for their CHILL Compiling System. They do not perform a flow-free analysis like Weihl, but instead collect program-point-specific aliases. However, they handle only single level pointers. Further, they use the notion of a supergraph which is obtained by linking together the condensed flow graphs (flow graph containing only the nodes relevant to alias analysis) of individual procedures, for interprocedural analysis. They perform intraprocedural analysis on this supergraph to achieve the effects of interprocedural analysis. This approach, however, leads to the incorrect pairing of calls and returns and makes their analysis imprecise. They also do not handle structures and arrays.

Coutant [Cou86] follows the same method as Weihl, of computing program-specific aliases as opposed to program-point-specific aliases. However, she handles multi-level pointers, structures and arrays, more precisely. It is not clear how she would handle recursive data structures. She mentions that program-point-specific alias calculation did not prove to be worth the computational effort in her implementation, but she

[1] *k-limiting* specifies the depth of the pointers which are considered [Lan92, LR92]. For example, if k = 2, **x represents **x, ***x, and etc. Therefore, if k-limiting is used, one obtains less accurate results for pointers with more than k indirections.

does not provide any relevant data. She also suggests the use of pragmas to help the compiler do a better analysis.

Cooper [Coo89] described an alias analysis algorithm in his Master thesis. His algorithm maintains explicit path information in the form of alias histories to ensure correct pairing of calls and returns. This method would give precise results, but does not seem to be feasible for implementation.

In the following section, we first give an introduction to Landi and Ryder's work and then compare their work with ours.

9.2 Comparison

In Landi and Ryder's analysis [Lan92, LR92], a program is represented as an *Interprocedural Control Flow Graph (ICFG)*. An ICFG is the union of control-flow graphs of each function with edges from each call-site to the entry of callee and from the exit of callee back to the call-site. Each call-site is replaced by two nodes, a *call node* and a *return node*. Each function has an *entry node* and an *exit node*. There is an edge from each call node to the entry node of related function (which is unique), and there is another edge from each exit node to the related return node(s). If a function is called more than once, its exit node would have more than one successor. As it is shown in Figure 9.2, there are two edges from exit node of function `f` to the call-site nodes of function `main`.

Two variables a and b have *may alias* relationship, represented as $<a,b>$, at a program point n, iff there exists a path, $n_1 n_2 ... n$ in the ICFG, on which a and b refer to the same location after execution of the program point n where n_1 is the entry node of the function `main`.

Two variables a and b have *must alias* relationship, at a program point n, iff for all the paths $n_1 n_2 ... n$ in the ICFG, a and b refer to the same location after execution of the program point n.

Their may alias relationship is based on the following *may-hold* information:

$$\textit{may-hold}(n_i,\ AA,\ PA),$$

where n_i is a node in ICFG, AA is a set of aliases, and PA is a may alias pair (e.g. $<a,b>$). *May-hold*(n_i, AA, $<a,b>$) is true, iff $<a,b>$ holds on some path from *entry*(n_i), the entry node of the procedure containing n_i, to n_i assuming there is a

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

    a = &b ;
    f() ;
    a = &c ;
    f() ;
}
f() {
    int *x ;

    x = a ;
}
```

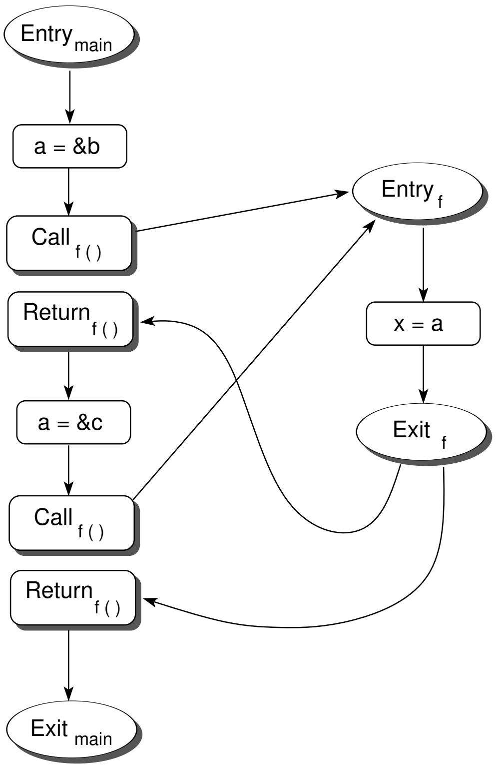


Figure 9.2: An example of ICFG.

path from the entry node of `main` function to *entry(n_i)* on which the assumed alias set AA holds. Having may-hold information, may aliases can be computed using the following formula:

$$may\text{-}alias\ (n_i) = \{PA \mid \exists\ (AA)\ may\text{-}hold\ (n_i,\ AA,\ PA) = true\}$$

Landi and Ryder's algorithm first collects the obvious *may-hold* information, e.g. <∗a,b> for `a = &b` (alias introduction), then collects the complete may-hold information using a worklist technique, and finally computes the may aliases from may-hold information using the above formula. Further, they restrict the cardinality of assumed-alias sets to a maximum of one.

In the following sections, we compare our method with Landi and Ryder's method by discussing alias representation in Section 9.2.1, must aliases in Section 9.2.2, interprocedural analysis in Section 9.2.3, dynamic allocation and k-limiting in Section 9.2.4, and type-casting in Section 9.2.5.

9.2.1 Alias Representation

We collect the alias information in the form of points-to pairs instead of exhaustive alias pairs like Landi and Ryder. Consider the simple example presented in Figure 9.3.

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

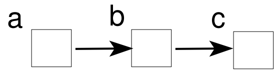


Figure 9.3: A simple C program.

In points-to analysis, we get {(a,b,D), (b,c,D)} as the result, while Landi and Ryder get the following:

$$\{<*a,b>, <**a,*b>, <**a,c>, <*b,c>\}$$

It is evident that points-to pairs represent the alias information in a more compact and informative manner than alias pairs. The example in Figure 9.4 also demonstrates this point[2]. Further, when alias pairs are generated exhaustively, some additional pairs might be reported while handling multi-level pointers. Consider the following example:

```
int **a, b ;
a = &b ;        {<*a,b>, <**a,*b>}
```

Landi and Ryder's analysis would give the alias pair `<**a,*b>`, which is redundant as `b` has still not been assigned the address of a memory location. Our points-to analysis gives the precise information that `a` points-to `b` and `b` does not point to any memory location. This also enables us to detect errors in programs, caused by dereferencing a pointer without assigning it the address of a valid memory location. Readers are referred to Chapter 7 for further details.

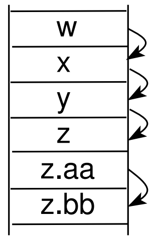
Moreover, points-to abstraction is a natural representation for calculating dependence information. This has been demonstrated in [HGS92]. The authors used the points-to information to build read and write sets for each statement. These sets actually consist of the reads and the writes to the abstract stack locations. Using the read/write sets, it is straight-forward to calculate data dependencies between statements.

[2] **Acknowledgment**: We are thankful to W. A. Landi who provided us with his results mentioned in Figure 9.4.

Consider the calculation of these sets for the statement 1 of Figure 9.4(a). From the points-to information (y,z,D), it can be easily determined that statement 1 reads the stack location `y` and writes to the stack location `z.bb`. However, with the alias pair representation, it would be necessary to inspect all the alias pairs involving `y` and its aliases to compute this information. This would be a considerably more expensive task (specially for multi-level pointers e.g. `(**x).bb`).

```
main()
{
    struct{
        int  *aa ;
        int  bb ;
    } ***w, **x, *y, z ;

    w = &x ;
    x = &y ;
    y = &z ;
    (*y).aa = &((*y).bb) ;
    (*y).bb = 10 ;          /* stmt1 */
}
```



(a) A C-program with its related abstract stack representation.

```
{<(**w)->bb,*(y->aa)>,     <z,*y>,
 <(**w)->bb,*(z.aa)>,      <y,*x>,                    <(**w)->bb,*((*x)->aa)>,
 <z,**x>,                  <x,*w>,                    <*((*x)->aa),(*x)->bb>,
 <y,**w>,                  <*(z.aa),y->bb>,           <*((**w)->aa),y->bb>,
 <(*x)->bb,*((**w)->aa)>,  <*(z.aa),z.bb>,            <*((**w)->aa),z.bb>,
 <*(z.aa),(*x)->bb>,       <*(y->aa),(*x)->bb>,       <*((*x)->aa),y->bb>,
 <*(y->aa),y->bb>,         <(**w)->bb,*((**w)->aa)>,  <z,***w>,
 <*((*x)->aa),z.bb>,       <*(y->aa),z.bb>}
```

(b) The result of alias analysis using the Landi and Ryder's approach.

```
{(z.aa,z.bb,D), (y,z,D), (x,y,D), (w,x,D)}
```

(c) The result of points-to analysis using our approach.

Figure 9.4: A comparison example of alias representation and points-to representation.

Finally, alias pairs can always be generated from points-to pairs, using the conversion algorithm given in Section 3.3.

However, points-to representation can provide extra alias information in certain cases. Consider the example given in Figure 9.5.

```
    int **a, *b, c ;
    if (c)
        a = &b ;      <--- {(a,b,D)}
    else
        b = &c ;      <--- {(b,c,D)}
S1:                   <--- {(a,b,P), (b,c,P)}
```

Figure 9.5: An example of points-to representation.

Landi and Ryder's result at program point `S1` would be {<∗a,b>, <∗∗a,∗b>, <∗b,c>}. If one wants to convert our points-to information to their alias pairs, one would get the set {<∗a,b>, <∗∗a,∗b>, <∗b,c>, <∗∗a,c>}, with the extra alias pair <∗∗a,c>. This happens because it can not be detected that the points-to pairs (a,b,P) and (b,c,P) are propagated to `S1` from different paths and transitive closure operation should not be applied on them.

Similarly, there are cases where points-to abstraction eliminates extraneous alias information, while Landi and Ryder's alias pairs can not. Consider the following example:

```
                            The alias                         Memory
                           information                     representation
                           -----------                     --------------

 int **x, *y,
        z, w ;

S1:   x = &y ;    {<*x,y>,<**x,*y>}                          x->y
S2:   y = &z ;    {<*x,y>,<**x,*y>,<*y,z>,<**x,z>}           x->y->z
S3:   y = &w ;    {<*x,y>,<**x,*y>,<*y,w>,<**x,z>,           x->y->w
                   <**x,w>}
```

When statement `S2` is processed, the alias pair <∗∗x,z> is generated by the interaction of the alias pair <∗x,y> and the assignment `y = &z`. At statement `S3`, the address of `w` is assigned to `y`, therefore, <∗x,z> is not valid anymore. Landi and Ryder's analysis fails to detect this case, because while propagating <∗∗x,z> through `S3`, they do not have the information that this alias pair was generated because of <∗y,z> being valid. While, if alias pairs are generated from the points-to information available after processing `S3`, this alias pair will not be generated. A similar observation was made in [CBC93]. They use the notion of transitive reduction which is quite similar to our points-to abstraction.

9.2.2 Must Aliases

The concept of may alias is equivalent to our possibly-points-to relationship, and the concept of must alias is equivalent to our definitely-points-to relationship. Must aliases are not mentioned in [LR92], although they are mentioned in [Lan92]. However, they consider the calculation of must aliases separately from that of may aliases. While we use must alias information to improve the precision of may alias computation for multi-level pointers by obtaining better kill information, as explained in Chapters 3, 4, and 5.

In [CBC93], it is mentioned that the extraneous alias information eliminated by transitive reduction can be equivalently removed using must alias information. However, in certain cases, extraneous pairs which can not be removed by using transitive reduction, can be removed by using must alias information. This is demonstrated in the following example:

```
                        The alias information         Memory
                      (without transitive closure)   representation
                      ----------------------------   --------------
 int **x, *y,
        z, w ;

S1:    x = &y ;          {<*x,y>}                        x->y
S2:    y = &z ;          {<*x,y>,<*y,z>}                 x->y->z
S3:   *x = &w ;          {<*x,y>,<*y,w>,<*y,z>}          x->y->w
```

At statement S3, if it were not known that `x` definitely points to `y` (i.e., `*x` is a must alias of `y`), we would not be able to kill the points-to relationship (`y->z`) (which can be killed due to the assignment `*x = &w`), and get the precise information `x->y->w`. The extraneous pair `<*y,z>` in the transitively reduced alias information reflects this fact.

Further, must-alias information is very useful for other analyses like constant propagation, register allocation, and dependence analysis, as explained in Chapter 8.

In the presence of multi-level pointers, calculation of must aliases would be quite complicated using the worklist technique of Landi and Ryder. This complexity is due to the fact that the information is processed in random order.

9.2.3 Interprocedural Analysis

Landi and Ryder represent the interprocedural structure of the program as an ICFG and use the conditional approach based on assumed aliases to perform interprocedural analysis.

We use the SIMPLE representation in conjunction with the invocation graph (as explained in Chapter 4), where each call-site (related to a given call chain) in the program has a separate invocation graph node associated with it. For each call, we visit the called function, analyze its body in the context of the call, and return the information to the call-site currently under consideration.

Our approach leads to the following advantages over Landi and Ryder's technique:

- The alias information from function exit points, always gets propagated to the correct call-site. It is never propagated along unrealizable interprocedural paths[3].

  To affect the correct pairing of calls and returns, Landi and Ryder associate an assumed-alias set with each alias pair, at call-sites. To avoid the exponential growth of this set, they restrict its cardinality to a maximum of one. In the presence of multi-level pointers, this restriction can lead to the propagation of information to unrelated call-sites. Consider the example given in Figure 9.6.

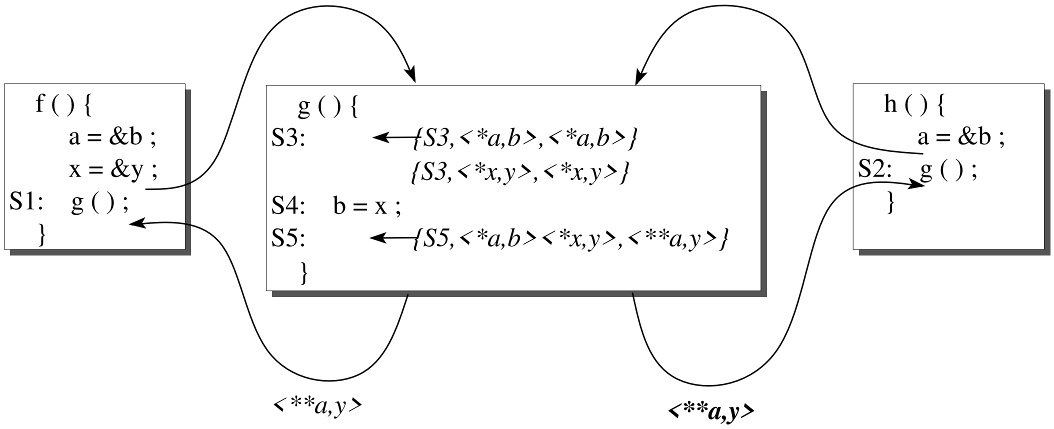
  At statement `S5`, the alias pair <**a,y> results from the interaction of the assignment `b = x` and the alias pairs <*a,b> and <*x,y> holding at program point `S3`, with <*a,b> and <*x,y> as the assumed aliases, respectively. Now, the assumed alias set for <**a,y> should include both of these alias pairs. Since the cardinality of the assumed alias set is restricted to at most one, Landi and Ryder propose to randomly pick one of the pairs. In case, <*a,b> is chosen as the assumed alias pair <**a,y> would be propagated to both the call-sites `S1` and `S2`, as <*a,b> is valid at both these call-sites. However, it should not be propagated to the call-site `S2`, as the other alias pair <*x,y> is not valid there. Our analysis would propagate it only to the call-site `S1`, because it would analyze the body of function g() separately for the two calls.

[3] A path is *realizable* iff it is a path in the program and whenever a procedure on this path returns, it returns to the call-site which invoked it [Lan92, LR92].

```
int **a, *b ;
int *x, y, z ;
main()
{
    f() ;
    h() ;
}
f()
{
    a = &b ;
    x = &y ;
S1: g() ;
}
```

```
h()
{
    a = &b ;
S2: g() ;
}
g()
{
S3:
S4: b = x ;
S5:
}
```

(a) A C-program



(b) The graph representation of intraprocedural analysis

Figure 9.6: An example of intraprocedural analysis for Landi and Ryder's method.

- Our analysis always ensures that two alias pairs propagated to a statement by two different calls, are not simultaneously used to generate a new alias pair. This is made possible by the structured nature of our analysis. The exact information for the current call is kept in a special data structure (stack matrix) which is used for the analysis. This information is also simultaneously merged with the pre-existing information saved in the statement nodes.

  Landi and Ryder's worklist technique fails to ensure the above mentioned point. For the example given in Figure 9.6, their analysis would infer the existence of the alias pair <**a,y> at statement `S5`. This is resolved from the interaction of the assignment statement `S4`, and the alias pairs <*a,b> and <*x,y> holding at statement `S3`. But it is possible that these alias pairs get propagated to `S3` from two different call-sites and never exist together. In that case, the alias pair <**a,y> would not be valid at `S5` in any program execution. They don't have a mechanism to detect such cases, and safely give the imprecise information.

- The structured nature of our points-to analysis, provides an opportunity to perform other data flow analyses simultaneously with it. This would enable other analyses to use the precise alias information available during points-to analysis, instead of the final merged information. To clarify this point, we demonstrate: (i) the results of performing constant propagation after the completion of points-to analysis, and (ii) the results of performing constant propagation in conjunction with points-to analysis. Consider the following example:

```
     int *a ;
     int b = 1 ;
     int c = 5 ;
     int x, y, z ;
     main() {
S1:     a = &b ;
S2:     f() ;
S3:     x = y ;  <--- {<x,b,P>,<x,c,P>} = {<x,1,P>,<x,5,P>}
S4:     a = &c ;
S5:     f() ;
S6:     z = y ;  <--- {<z,b,P>,<z,c,P>} = {<z,1,P>,<z,5,P>}
     }
     f() {
S7:     y = *a ; <--- {<y,b,P>,<y,c,P>} = {<y,1,P>,<y,5,P>}
     }
```

The arrow links represent the constant propagation information. `<x,y,D>` means that variable `x` is definitely equal to variable `y`. `<x,y,P>` means that variable `x` is possibly equal to variable `y`. For case (i), since the merged information for points-to analysis is used, `y` is either `b` or `c` (the constant propagation information saved at `S7`). Therefore, `x` and `z` can not be concluded to be constant. Below, we show the same information, but for the case that constant propagation is done simultaneously with points-to analysis.

```
     int *a ;
```

```
     int b = 1 ;
     int c = 5 ;
     int x, y, z ;
     main() {                                           main() {
S1:     a = &b ;                                            a = &b ;
S2:     f() ;                                               f() ;
S3:     x = y ;  <--- {(x,b,D)} = {(x,1,D)}                 x = 1 ;
S4:     a = &c ;                                            a = &c ;
S5:     f() ;                                               f() ;
S6:     z = y ;  <--- {(z,b,D)} = {(z,5,D)}                 z = 5 ;
     }                                                  }
     f() {                                              f() {
S7:     y = *a ;                                            y = *a ;
     }                                                  }
```

In this case, statement `S3` is processed when the points-to information is (a,b,D), and therefore, `y` at `S7` would be equal to `b`, which is constant 1. This gives a more precise information about `x` at statement `S3`, that results in the deduction fact that `x` is constant 1. Similarly, at statement `S6`, we get precise information because we have (a,c,D). This statement also resolves `z` as being the constant 5.

- Our interprocedural analysis can be easily extended to handle function pointers, without incurring any additional cost. This is facilitated by the structured nature of our points-to analysis and the powerful representation of the interprocedural structure of the program by the invocation graph. The detailed algorithm is described in [Ghi92], and a brief summary is provided in 8.2.2.
- By using invocation graph, we are able to save some important properties of each call in the corresponding invocation graph node. These are described bellow:
  - The relationship between the invisible variables of the callee, and the caller's variables (map information of each call). This is a useful information for other analyses [HEGV93].

    As an example, consider the constant propagation analysis for the example represented in Figure 9.7. Assume that this analysis starts after the completion of points-to analysis. Using statement `S5` and the relationship `(x,1_x,D)`, one can deduce that the invisible variable `1_x` is constant 7. After returning to the call-sites `S1` and `S3`, one needs to know the variable(s) represented by `1_x` in the caller, so that the constant information can be propagated. Using the map information related to `S1/S3`, we resolve that at b/c is constant 7 at S2/S4. Therefore b/c can be replaced by constant 7 at statement S2/S4.

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

        a = &b ;
S1:     f(a) ;
S2:     d = b ;
        a = &c ;
S3:     f(a) ;
S4:     d = c ;
    }
    f(int *x) {
S5:     *x = 7 ;  <--- (x,1_x,D)
    }
```

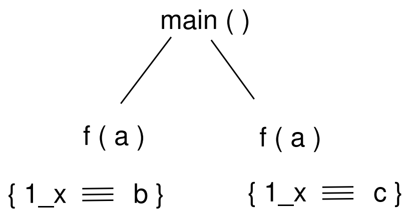


Figure 9.7: An example of the usage of the map and unmap information saved in invocation graph.

  - The input and output points-to information for each node of invocation graph can be saved in a data base, so that if a function is called for the second time with the same input, there is no need to recalculate the output for it.

- Our invocation graph is also suitable for *procedure cloning* [CHK92]. This involves duplicating the body of a function with a different name (i.e. making its clone), and replacing some calls to this function, with a call to its clone. This technique is useful when the data flow information propagated from two different calls to a function is significantly different. In this case, if the information from one of the calls is propagated to the clone, the imprecision resulting from merging significantly different information can be avoided. To replace a function call with a call to its clone, we simply need to modify the invocation graph node corresponding to it. Further, a clone for the function containing this call, also needs to be developed. This process would continue recursively till the root of the invocation graph is reached.

9.2.4 Dynamic Allocation and k-limiting

To handle recursively defined dynamic data structures, Landi and Ryder use the notion of k-limited object names. They limit the number of dereferences permissible in any object name to some constant k. They provide the alias information in the form of alias pairs between these names, which gives a very imprecise picture of the heap.

A different abstraction is required to properly handle the aliasing for dynamically allocated objects. Presently, we use a single location called 'heap' in our abstract stack for the points-to analysis. All heap-directed pointers point to this location. Development of appropriate abstractions for heap analysis, based on the path matrix model proposed in [Hen90], is currently under development [Ghi93a]. There is no need to use k-limiting for abstract stack analysis, as the number of objects on the stack would always be finite and compile-time determinable.

9.2.5 Type Casting

Landi and Ryder do not handle type casting. It is a frequently used feature of the C-language and needs to be taken into consideration. We handle a wide variety of type casting as explained in Section 6.2. The only case that we do not handle, is when the type is transformed to/from a scalar variable, as shown in the following example:

```
int *a, b, c ;
b = (int) (&c) ;
a = (int*) b ;
```

This sort of type casting can also be handled, but it would make the analysis expensive. Further, this case does not occur frequently in real C programs.

The most recent algorithm for *flow-sensitive* interprocedural[4] alias analysis has been presented in [CBC93]. Intraprocedurally, their algorithm is similar to that of Landi and Ryder, with the exception that they use a sparse representation called *Sparse Evaluation Graph*, instead of control flow graph. To ensure the flow of information along realizable paths during interprocedural analysis, they associate two items of information with each alias pair: (i) the call-site from which the alias pair is propagated, and (ii) a source alias set, which is the set of aliases that induce the alias pair through the call. The call-site information alone is not sufficient to propagate aliases along realizable interprocedural paths. Besides, unless they keep source alias sets of exponential size, our analysis would give more precise results. They also proposed the schemes of using transitive reduction and must alias information, to

[4] A flow-sensitive interprocedural analysis is an analysis which makes use of the intraprocedural control flow information associated with individual procedures.

improve the precision of alias analysis. We have already discussed these issues in the previous sections.