Modular Pointer Analysis

Susan Horwitz and Marc Shapiro
Computer Sciences Department, University of Wisconsin-Madison
1210 West Dayton Street, Madison, WI 53706 USA
Electronic mail: {mds, horwitz}@cs.wisc.edu

1 Introduction

Modern languages (such as C and C++) that include pointers can be very difficult to analyze. Most of the traditional analyses (*e.g.*, reaching definitions, live variables, constant propagation) rely on knowing which variables are used and which variables are defined at each program point. They also rely on knowing which variables might be used or defined by a called function, which in turn requires knowing which function is being called at each call site. In the presence of pointers, determining which variables are used/defined at a program point, and which function is called at a call site are non-trivial problems; a pointer analysis must be performed to answer those questions before any subsequent analyses can be done.

There has been a great deal of recent work on techniques for pointer analysis. One promising approach involves *flow-insensitive, context-insensitive* algorithms; *i.e.*, algorithms that treat a program as an unordered set of statements rather than taking the program's actual flow of control into account. The disadvantage of using a flow-insensitive analysis is that the results can be much more conservative than the results produced by a flow-sensitive analysis (*i.e.*, the points-to sets computed using a flow-insensitive approach can be much larger than those computed using a flow-sensitive approach). However, the advantage of using a flow-insensitive analysis is that it usually takes much less time and space than a flow-sensitive analysis.

Even within the somewhat limited context of flow-insensitive algorithms there is a range of approaches that trade precision for speed. At one end is the algorithm defined by Andersen [And94] which may require $O(N^3)$ time, where N is the size of the program. At the other end is the algorithm defined by Steensgaard [Ste96] which runs in almost linear time, but which usually produces less precise results (*i.e.*, larger points-to sets) than Andersen's analysis. (Both algorithms concentrate on tracking stack-allocated storage. A safe approximation to heap-allocated storage is maintained by treating all storage allocated at an individual allocation site as a single location.)

In previous work [SH97b] [SH97a], we carried out several sets of experiments that explored how well Andersen's and Steensgaard's approaches work in practice. Our results indicate that neither algorithm is likely to scale up well enough to be applied to large programs (hundreds of thousands of lines). Steensgaard's algorithm produces results that are too conservative (*i.e.*, the points-to sets tend to be quite large, which in turn leads to overly conservative results in subsequent analyses that make use of the points-to information). While Andersen's algorithm produces much better results, it seems to be limited by its space requirements: when run on a Sparc 20/71 with 256 MB of RAM, it sometimes ran out of memory when applied to medium-sized programs (tens of thousands of lines of code). Even when it did not actually run out of memory, it often took a very long time (wall-clock time, not CPU time) to finish, due to thrashing.

One reason for the large memory requirements of pointer-analysis algorithms like Andersen's is that they work on data structures that represent the assignment statements and the points-to information for the *entire* program. This is because, in general, the statements and the points-to relations of the variables in one function can affect what is pointed to by pointers in all other functions. In a language like C that permits casting, even assignments to non-pointer variables must be considered. For example:

$$\begin{array}{l}\text{int } a, b, c, *p;\\ a = (\text{int})\&c;\\ b = a;\\ p = (\text{int}*)b;\end{array}$$

In this code fragment, although variables a and b are declared to be of type int, they are actually used to store the address of c. If the assignments to a and b were ignored, it could not be determined that p points to c.

In this paper, we explore techniques for *modular* pointer analysis: analyzing small pieces of the program (*e.g.*, individual functions or files), then combining the results of the individual analyses. We expect our modular approach to have two important benefits:

- It will permit algorithms like Andersen's to be applied to much larger programs than would otherwise be possible, thus permitting reasonably precise analysis of at least medium-sized programs.

- It will provide a way to summarize the effects (on pointers) of auxiliary files such as libraries. This means that programs that use libraries can be analyzed in the context of this summary information, rather than requiring source code for the library functions.

To provide some intuition into our approach, consider the two functions shown in Figure 1. Both functions involve assignments to pointers and assignments via pointer dereferences; however (as suggested by their names), function *LocalPtrs* has only local effects, while function *GlobalPtrs* has global effects (it may change what is pointed to by $*g1$, which is visible outside that function). Also note that the five assignment statements involving local variables a, b, q and p are not needed to capture the global effects of function *GlobalPtrs*; those effects can be summarized using the single statement "$*g1 = *g2$;".

Our algorithm for modular pointer analysis takes advantage of these observations to reduce memory requirements. In particular, we propose a three-step approach:

step 1: For each function:

- **(a)** Compute local points-to sets for each variable.
- **(b)** Use the local points-to sets to create a set of statements that reflect the global effects of the function.

step 2: Do global analysis using the collection of global statements from all of the functions.

step 3: Use the results of the global analysis to transform the local points-to sets into final points-to sets for all variables.

This approach is similar to the standard approach to so-called global dataflow analysis (meaning within one function, across basic blocks), in which information is first computed for each basic block, then dataflow analysis is done across all basic blocks, then the results of the analysis are propagated back to the individual statements within each basic block.

We expect that the number of statements that reflect the global effects of all functions will be much smaller than the total size of the program (for example, the program in Figure 1 has 10 assignment statements, but the global effects of the two functions can be summarized using a single assignment statement). Therefore, it seems likely that this approach will permit much larger programs to be analyzed.

The remainder of the paper is organized as follows: The three steps of our algorithm are described in detail in Section 2, assuming that functions have no parameters and return no results, and that there are no "indirect" function calls (calls via function pointers). Section 3 explains how to handle parameters and non-void functions, and Section 4 describes two ways to handle indirect calls. Section 5 presents conclusions.

`int **g1, **g2;`
`void LocalPtrs()`
`{`
`  int **q, *p, a, b;`
`    a = (int)&p;`
`    b = a;`
`    q = (int *)b;`
`    if (...) *q = *g1;`
`    else *q = *g2;`
`}`
	`int **g1, **g2;`
	`void GlobalPtrs()`
	`{`
	`  int **q, *p, a, b;`
	`    a = (int)&p;`
	`    b = a;`
	`    q = g1;`
	`    (int *)*b = *g2;`
	`    if (...) *q = p;`
	`}`

Figure 1: *LocalPtrs* does not affect points-to information in other functions, but *GlobalPtrs* does.

2 Basic Algorithm

In this section, we present our basic algorithm for modular pointer analysis. To simplify the presentation, we begin by assuming that functions have no parameters, and return no results. We use the program shown in Figure 2(a) as a running example to illustrate each step of the algorithm.

2.1 Step 1(a): Compute local points-to sets

As mentioned in the Introduction, the first step of our algorithm involves analyzing each individual function, computing that function's local points-to sets. Since we are interested in flow-insensitive analysis, branches (and the predicates that control them) can be ignored. Declarations are also irrelevant; as discussed in the Introduction, it is not safe to use declarations to exclude non-pointer variables from pointer analysis.

Since we are interested in context-insensitive analysis, and since we are currently assuming that functions have no parameters and return no results, function calls can also (currently) be ignored.

Example: Figure 2(b) shows the sets of assignment statements that represent functions f and *main*. □

Once a function has been transformed to a set of assignment statements, it can be analyzed using any flow-insensitive pointer-analysis algorithm. The only subtlety is how to handle the points-to sets of the global variables that are used in the function. For example, function f includes the statement

$$q = g1;$$

When this function is analyzed, it should be determined that q's points-to set includes everything in $g1$'s points-to set. However, the points-to set of $g1$ is not known when function f is analyzed. Therefore, we use the special symbol $@g1$ to represent $g1$'s points-to set and we add the assignment:

$$g1 = \&@g1;$$

to the set of statements for f to "communicate" this to the pointer-analysis algorithm that is ap-

int $**g1$, $**g2$;
 	
void $f()$
{
int $**q$, $*p$;
$q = g1$;
$p = *g2$;
if (...) $*q = p$;
}
 	
void $main()$
{
int a, b, $*x$, $*y$;
$x = \&a$;
$y = \&b$;
$g1 = \&x$;
$g2 = \&y$;
$f()$;
}
	f:
	
	$q = g1$;
	$p = *g2$;
	$*q = p$;
	
	$main$:
	
	$x = \&a$;
	$y = \&b$;
	$g1 = \&x$;
	$g2 = \&y$;
		f's local points-to sets:
		
		$q \rightarrow \{ @g1 \}$
		$p \rightarrow \{ @@g2 \}$
		$@g1 \rightarrow \{ @@g2 \}$
		
		$main$'s local points-to sets:
		
		$x \rightarrow \{ a \}$
		$y \rightarrow \{ b \}$
		$g1 \rightarrow \{ x \}$
		$g2 \rightarrow \{ y \}$

(a) Original Code	**(b) Ignoring Declarations and Conditions**	**(c) Local Points-to Sets**

f's global statements $*g1 = *g2$; $main$'s global statements $x = \&a$; $y = \&b$; $g1 = \&x$; $g2 = \&y$;	$g1 \rightarrow \{ x \}$ $g2 \rightarrow \{ y \}$ $x \rightarrow \{ a, b \}$ $y \rightarrow \{ b \}$	$g1 \rightarrow \{ x \}$ $g2 \rightarrow \{ y \}$ $x \rightarrow \{ a, b \}$ $y \rightarrow \{ b \}$ $q \rightarrow \{ x \}$ $p \rightarrow \{ b \}$
(d) Global Statements	**(e) Result of Global Analysis**	**(f) Final Points-to Sets Global + Local**

Figure 2: Simple example illustrating our approach to modular pointer analysis.

plied to the statements that represent f. (As mentioned above, we want to be able to use any pointer-analysis algorithm for this step – *i.e.*, we want to use the pointer-analysis algorithm as a "black box"; if we did not include the assignment $g1 = \&@g1$, an analysis might simply determine that q does not point to anything, or it might signal an error when attempting to process the statement "$q = g1;$".)

Note however, that simply adding the statement "$g = \&@g;$" for each global variable g that is used in the current function, may not be enough in some cases. For example, function f also includes the statement:

$$p = *g2;$$

Just adding "$g2 = \&@g2$" is not good enough in this case; we also need something to represent the points-to set of the points-to set of $g2$. We can solve this by also including the assignment

$$@g2 = \&@@g2;$$

but this naturally leads to the question "How many such statements need to be included?", and "Might the number sometimes be unbounded?". Unfortunately, the answer to the second question is yes, and even when the number is bounded there is no obvious way to determine the bound without essentially doing the pointer analysis. If we are willing to abandon our goal of using an existing pointer analysis as a black box, we can add such statements as needed. However, we must still deal with cases where an unbounded number of those statements is needed. For example, consider the following function:

```
Listnode *L;
void f() {
  Listnode *p;
    p = L;
    while (...) {
        p = (*p).next;
    }
}
```

Assuming that the fields of a structure are "collapsed" (a single variable is used to represent all fields), the assignment statements for this function would be:

$$p = L;$$
$$p = *p$$

and pointer analysis would discover:

$$p \to \{@L, @@L, @@@L, ...\}$$

To handle this situation, we propose to use a kind of "k-limiting": for each global variable g that is used in the function, add assignments that set up a points-to chain of length k starting from g, and one more assignment to add a cycle to the end of that chain. (Some ramifications of this approach are discussed below in Section 2.4.) For example, for the Listnode code given above, with $k = 3$, we would add the assignments:

$$L = \&@L;$$
$$@L = \&@@L;$$
$$@@L = \&@@@L;$$
$$@@@L = \&@@@L;$$

which would represent the points-to chain:

and pointer analysis would discover:

$$p \to \{@L, @@L, @@@L\}$$

Example: Figure 2(c) shows the results of pointer analysis on the assignment statements of Figure 2(b), using this approach. (Note: points-to facts like "$g1 \to \{@g1\}$", which are derived directly from the extra assignments that we add to set up points-to sets for the global variables, *e.g.*, "$g1 = \&@g1;$", are omitted from Figure 2(c).) □

2.2 Step 1(b): Create statements representing global effects

The next step of our algorithm is to use the local points-to sets to create a set of statements that reflect the global effects of the function. This set of statements must capture the effects of the function on the values of the global variables as well as on the value of any variable in the points-to set of a global (since a local variable that is in the points-to set of a global may influence the points-to sets of variables outside this function). As pointed out in the Introduction, it can be very inefficient simply to include all statements that involve these variables. Instead, the following steps can be performed to create an appropriate set of assignments:

1. Put all global variables (including variables like $@g$ created to represent the points-to sets of globals) onto a worklist.
2. While the worklist is not empty do:

    Select and remove one variable x from the worklist.

    For each variable y in x's points-to set do:

        Add the assignment "$x = \&y$" to the set of "global effects" assignments.

        If y is a local variable then

            make y global

            add y to the worklist

When creating "global effects" assignment statements, the symbol "@" is treated like a star (including canceling with a "&"). So for example, if p's points-to set includes "$@g1$" and "$@@g2$", the corresponding "global effects" statements would be:

$$p = g1;$$
$$p = *g2;$$

Also, statements of the form "$g = g$" (which arise from points-to facts like "$g \to @g$") can be ignored.

Example: In function f of the running example program, only variable $@g1$ is global. In function *main*, $g1$ and $g2$ are initially global; x and y become global because they are in the points-to sets of $g1$ and $g2$, respectively. Figure 2(d) shows the "global effects" statements for functions f and *main*. □

2.3 Step 2: Do global analysis

Step two of our algorithm involves global points-to analysis. As for the local analysis done in step 1, this can be performed by *any* flow-insensitive pointer-analysis algorithm. The input is the union of the sets of "global effects" assignment statements created for each function, and the output is the points-to sets for all (global) variables.

Example: Figure 2(e) shows the results of global analysis for the running example program. □

2.4 Step 3: Use results of global analysis to compute final points-to sets

The final step of the algorithm is to use the results of global analysis to transform the points-to sets computed for the local variables of each function (which may contain values like $@g$) to their final forms. In particular, each variable of the form $@g$ (used to represent the points-to sets of the global variables) is replaced with the global variable's actual points-to set. For example, given the following points-to sets for local variable p, and global variables $g1$ and $g2$:

$$\begin{array}{l} p \to \{x, @g1, @@g2\} \\ g1 \to \{a, b\} \\ g2 \to \{q\} \\ q \to \{c\} \end{array}$$

the final points-to set for p would be: "$p \to \{x, a, b, c\}$".

Example: Figure 2(f) shows the final points-to sets for the running example. The sets for (global) variables $g1$, $g2$, x, and y come from the results of the global analysis phase, while the sets for (local) variables q and p are computed by replacing $@g1$ with the points-to set of $g1$, and $@@g2$ with the points-to set of the points-to set of $g2$, respectively. □

The process described above for carrying out step 3 of our algorithm is not quite complete, because of the k-limiting approach used in Step 1. If a local variable's points-to set includes $@@@g$ (where the number of @'s = k), $@@@g$ must be replaced with all variables reachable in the points-to relation from $@@@g$. For example, given $k = 3$ and:

$$\begin{array}{l} p \to \{@@@g\} \\ g \to \{a\} \\ a \to \{b\} \\ b \to \{c\} \\ c \to \{d\} \end{array}$$

the final points-to set for p would be:

$$p \to \{c, d\}$$

Note that, while safe, the use of this k-limiting approach can lead to a loss of precision in some cases. For example, consider the following program:

$$\begin{array}{l} \text{int} * * * * g1,\ * * * g2,\ * * g3,\ * g4,\ g5; \\ \text{void } main() \\ \{ \\ \quad g4 = \&g5; \\ \quad g3 = \&g4; \\ \quad g2 = \&g3; \\ \quad g1 = \&g2; \\ \quad f(); \\ \} \\ \text{void } f() \\ \{ \\ \ \ \text{int} * * * p; \\ \quad p = *g1; \\ \} \end{array}$$

If $k = 2$, then the set of statements that represents function f is:

$$\begin{array}{l} g1 = \&@g1; \\ @g1 = \&@@g1; \\ @@g1 = \&@@g1; \\ p = *g1; \end{array}$$

and the result of local pointer analysis is:

$$p \to \{@@g1\}$$

The points-to relation computed by global analysis is:

$$g1 \to g2 \to g3 \to g4 \to g5$$

and the variable "$@@g1$" in p's points-to set is replaced with $g3$, $g4$, and $g5$, even though in fact p only points to $g3$.

In spite of this example, we believe this approach will work well in practice, because we think that in actual programs, pointer chains are built up using heap-allocated storage (and, as mentioned in the Introduction, all storage allocated at a given site is "collapsed" into a single location by the analysis, anyway) rather than by a sequence of assignments like those in function *main* above. Experiments are clearly needed to verify or refute this hypothesis.

3 Handling parameters and returned values

In this section, we describe how to extend our algorithm to handle functions with parameters, and functions that return results. We illustrate the extensions using a modified version of the program from Figure 2, shown in Figure 3. In the new version, the variables $g1$ and $g2$ are made local to *main*, but are passed as parameters to function f, and function f returns a pointer.

Our approach is to replace function calls and return statements with plain assignment statements, and to model function entry with more assignment statements. Using this approach, the only step of the algorithm that needs to be modified is the first part of Step 1(a); in addition to ignoring branches

int $*f$(int $**\,h1$, int $**\,h2$)
{
int $**\,q$, $*p$;
$q = h1$;
$p = *h2$;
if (...) $*q = p$;
return($*q$);
}
 	
void $main$()
{
int a, b, $*x$, $*y$;
int $**\,g1$, $**\,g2$;
$x = \&a$;
$y = \&b$;
$g1 = \&x$;
$g2 = \&y$;
$y = f(g1, g2)$;
}
	f:
	
	$h1 = f_1$;
	$h2 = f_2$;
	$q = h1$;
	$p = *h2$;
	$*q = p$;
	$f_{result} = *q$;
	
	$main$:
	
	$x = \&a$;
	$y = \&b$;
	$g1 = \&x$;
	$g2 = \&y$;
	$f_1 = g1$;
	$f_2 = g2$;
	$y = f_{result}$;
		f's local
		points-to sets:
		
		$h1 \rightarrow \{\ @f_1\ \}$
		$h2 \rightarrow \{\ @f_2\ \}$
		$q \rightarrow \{\ @f_1\ \}$
		$p \rightarrow \{\ @@f_2\ \}$
		$@f_1 \rightarrow \{\ @@f_2\ \}$
		$f_{result} \rightarrow \{\ @@f_1\ \}$
		
		$main$'s local
		points-to sets:
		
		$x \rightarrow \{\ a\ \}$
		$y \rightarrow \{\ b,\ @f_{result}\ \}$
		$g1 \rightarrow \{\ x\ \}$
		$g2 \rightarrow \{\ y\ \}$
		$f_1 \rightarrow \{\ x\ \}$
		$f_2 \rightarrow \{\ y\ \}$

(a) Original Code	**(b) Transformed to Assignments**	**(c) Local Points-to Sets**

f's global statements $*f_1 = *f_2$; $f_{result} = *f_1$; $main$'s global statements $f_1 = \&x$; $f_2 = \&y$; $x = \&a$; $y = \&b$; $y = f_{result}$;	$f_1 \rightarrow \{\ x\ \}$ $f_2 \rightarrow \{\ y\ \}$ $f_{result} \rightarrow \{\ a,\ b\ \}$ $x \rightarrow \{\ a,\ b\ \}$ $y \rightarrow \{\ a,\ b\ \}$	$x \rightarrow \{\ a,\ b\ \}$ $y \rightarrow \{\ a,\ b\ \}$ $g1 \rightarrow \{\ x\ \}$ $g2 \rightarrow \{\ y\ \}$ $q \rightarrow \{\ x\ \}$ $p \rightarrow \{\ a,\ b\ \}$
(d) Global Statements	**(e) Result of Global Analysis**	**(f) Final Points-to Sets Global + Local**

Figure 3: Extended example illustrating how to handle parameters and non-void functions.

and declarations, the transformations described below must be performed to produce a set of assignment statements for each function.

We model parameter passing using special intermediate parameter-passing variables for each function, and we model returning a result using a special return-result variable for the function. (For function f, we use f_1, f_2, *etc.* as the parameter-passing variables, and f_{result} as the return-result variable.) Function entry is modeled by adding a set of assignments that copy the values of the parameter-passing variables to the formal parameters. Returning a result is modeled by replacing the return statement with an assignment that copies the value of the result expression to the return-result variable. A function call is modeled by adding a set of assignments that copy the values of the actual parameters to the parameter-passing variables, and replacing the call itself with the return-result variable.

Thus, entry to a function whose header is of the form:

$$f(formal_1, formal_2, ...formal_n)$$

is represented by the set of assignments:

$$\begin{array}{l} formal_1 = f_1; \\ formal_2 = f_2; \\ ... \\ formal_n = f_n; \end{array}$$

Similarly, a return statement of the form:

$$\text{return } exp$$

is replaced with the assignment:

$$f_{result} = exp;$$

And a function call of the form:

$$x = f(actual_1, actual_2, ...actual_n);$$

is transformed to:

$$\begin{array}{l} f_1 = actual_1; \\ f_2 = actual_2; \\ ... \\ f_n = actual_n; \\ x = f_{result}; \end{array}$$

Note that the parameter-passing variables and the return-result variables must be considered global to all functions (so that assignments that set their values appear in the "global effects" statements computed for all functions).

Example: Figure 3(b) shows the transformation of the code shown in Figure 3(a) to accommodate parameters and function results. Figures 3(c) - (f) show the remaining steps of the algorithm for this example. □

4 Handling indirect calls

In this section we describe two ways to handle "indirect" calls (calls via pointers). The advantage of the first approach is that it requires no modification to the pointer-analysis algorithm used to do the local and global analyses. The disadvantage is that it can only be used on "well typed" programs: programs in which variables that are pointers to functions are declared consistently with their uses (in particular, the declared number of parameters must be no less than the actual number of parameters in any call via this pointer), and there can be no "hiding" of function pointers via casting (*e.g.*, assigning from a function pointer to a variable of a different type). The second approach overcomes this restriction at the expense of requiring a minor modification to the pointer-analysis algorithm.

4.1 Method 1

The basic problem with indirect calls is how to get the values of the actual parameters into the appropriate parameter-passing variables, and how to get the result of the call back from the appropriate return-result variable. For example, consider the following code, which contains a call via function pointer P:

```
int *f(int *x, int *y)
{
    if (...) return x;
    else return y;
}

void main()
{
  int *(*P)(int*, int*);
  int *q;
  int a, b;
    P = &f;
    q = (*P)(&a, &b);
}
```

Entry to function f is represented using the statements

$$\begin{array}{l} x = f_1; \\ y = f_2; \end{array}$$

to copy values from f's parameter-passing variables to its formal parameters. Similarly, the return statements are transformed to:

$$\begin{array}{l} f_{result} = x; \\ f_{result} = y; \end{array}$$

to copy the values of the returned expressions to f's return-result variable. However, the call to f is made via pointer P; when the call is transformed to a set of assignments, it is not known what function P points to, so we cannot translate the call to copy the values of the actual parameters into f_1 and f_2, and to copy the result from f_{result} to q.

Instead, we use a chain of assignments to get the same effect. We transform the assignment "$P = \&f;$" to:

$$\begin{array}{l} P = \&f; \\ f_1 = P_1; \\ f_2 = P_2; \\ P_{result} = f_{result}; \end{array}$$

and we transform the call "$q = (*P)(\&a, \&b);$" to

$$P_1 = \&a;$$
$$P_2 = \&b;$$
$$q = P_{result};$$

Now, for example, the three assignments "$P_1 = \&a; f_1 = P_1; x = f_1;$" serve to copy the value of the actual parameter $\&a$ into the corresponding formal, x.

Additionally, any time the value of a pointer to a function (or a pointer to a pointer to a ... to a function) is copied – via an assignment, by being passed as a function parameter, or by being returned as the result of a function – a similar set of assignments must be created to copy values from one set of parameter-passing variables to another, and from one return-result variable to another. For example, if p and g are pointers to a non-void function with n parameters, the assignment "$g = p;$" is transformed to:

$$p_1 = g_1;$$
$$p_2 = g_2;$$
$$\ldots$$
$$p_n = g_n;$$
$$g_{result} = p_{result};$$

Note that it is the creation of these assignment statements that requires knowing the type of the function pointer (in particular, the number of arguments) so that the right number of assignments between parameter-passing variables can be created.

Example: Figure 4(a) shows a program that involves passing a function pointer as a parameter, and assigning from one function pointer to another. Figure 4(b) shows the corresponding set of assignment statements for each function. $\square$

4.2 Method 2

Our second method for handling indirect calls does not rely on type information, but does require modifying the pointer-analysis algorithm that is used to do the local and global analyses. Using this approach, no chains of assignments are created; instead, an indirect call like

$$q = (*P)(\&a,\ \&b);$$

is transformed to a set of assignments very similar to those produced by the basic method presented in Section 2:

$$(*P)_1 = \&a;$$
$$(*P)_2 = \&b;$$
$$q = (*P)_{result};$$

The pointer-analysis algorithm must recognize assignments of this form (that use names like $(*P)_1$) as special, and must replace the assignment "$(*P)_1 = exp;$" with "$f_1 = exp;$" for every function f that is discovered to be in the points-to set of P (and similarly for the assignment that copies back from $(*P)_{result}$). This is clearly a simpler approach than method 1, but has the disadvantage that a pointer-analysis algorithm can no longer be used as a "black box".

5 Conclusions and Future Work

We have presented an algorithm that permits flow-insensitive, context-insensitive, stack-based pointer analysis to be partitioned into three phases:

- a local phase that analyzes each individual module, creating a set of assignments that represent the module's effects on global variables;
- a global phase that uses the "global effects" assignments to compute points-to sets for all global variables;
- a local phase that uses the results of the global phase to finalize local points-to sets.

Although in our presentation we have used individual functions as modules, it may be that in practice it is better to work at the file level; individual files are small enough to process efficiently, and the set of assignments that represent the global effects of a file should be smaller than the union of the sets that represent the global effects of the individual functions in that file.

We expect that there will be two main advantages to our approach:

- It will permit pointer analysis of larger programs than would otherwise be possible.
- It will provide a way to summarize the effects (on pointers) of auxiliary files such as libraries, thus obviating the need for library source code during pointer analysis.

Experiments are needed to determine the best level of granularity for modules, and to confirm that the approach does reduce memory requirements as expected.

References

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[SH97a] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Proceedings of SAS97: 4th International Static Analysis Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 16–34, September 1997.

[SH97b] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *ACM Symposium on Principles of Programming Languages*, pages 1–14, January 1997.

[Ste96] B. Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

typedef int *(*FNPTR)(int *, int *);	

int $*f$(int $*x$, int $*y$)
{
  if (...) return x;
  else return y;
}
	f:
	
	/* function entry */
	$x = f_1$;
	$y = f_2$;
	
	/* function body */
	$f_{result} = x$;
	$f_{result} = y$;

void g(FNPTR $P1$)
{
 FNPTR $P2$;
 int $*q$;
 int a, b;
 	
  $P2 = P1$;
  $q = (*P2)(\&a, \&b)$;
}
	g:
	
	/* function entry
	 * $P1$ and g_1 are fn ptrs so extra assignments are needed */
	$P1 = g_1$;
	$g_{1_1} = P1_1$;
	$g_{1_2} = P1_2$;
	$P1_{result} = g_{1_{result}}$;
	
	/* $P2 = P1$
	 * $P2$ and $P1$ are fn ptrs so extra assignments are needed */
	$P2 = P1$;
	$P1_1 = P2_1$;
	$P1_2 = P2_2$;
	$P2_{result} = P1_{result}$;
	
	/* call to $(*P2)$ */
	$P2_1 = \&a$;
	$P2_2 = \&b$;
	$q = P2_{result}$;

void $main$()
{
  $g(\&f)$;
}
	$main$:
	
	/* call to g
	 * $\&f$ and g_1 are fn ptrs so extra assignments are needed */
	$g_1 = \&f$;
	$f_1 = g_{1_1}$;
	$f_2 = g_{1_2}$;
	$g_{1_{result}} = f_{result}$;

(a) Original Code	**(b) Translated to Assignments**

Figure 4: Example illustrating how to handle indirect function calls, method 1.