Using Static Single Assignment Form to Improve Flow-Insensitive Pointer Analysis *

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

Abstract

A pointer-analysis algorithm can be either flow-sensitive or flow-insensitive. While flow-sensitive analysis usually provides more precise information, it is also usually considerably more costly in terms of time and space. The main contribution of this paper is the presentation of another option in the form of an algorithm that can be 'tuned' to provide a range of results that fall between the results of flow-insensitive and flow-sensitive analysis. The algorithm combines a flow-insensitive pointer analysis with static single assignment (SSA) form and uses an iterative process to obtain progressively better results.

1 Introduction

Having information about what pointer variables may point to is very useful (and often necessary) when performing many kinds of program analyses. Obviously, the better (or more precise) the information, the more useful the information is. A points-to analysis that takes into account the order in which statements may be executed (*i.e.*, a flow-sensitive analysis) generally provides more precise information than a flow-insensitive analysis; however, flow-sensitive analyses are considerably more costly in terms of time and/or space than flow-insensitive analyses. Thus, the options for pointer analysis one is generally presented with are: (1) flow-insensitive - faster but less precise; and (2) flow-sensitive - more precise but time/space consuming. The main contribution of this paper is the presentation of another option in the form of an algorithm that can be 'tuned' to provide a range of results. The algorithm combines a flow-insensitive pointer analysis with static single assignment (SSA) form and uses an iterative process to obtain progressively better results along the spectrum from flow-insensitive to flow-sensitive. The particular flow-insensitive analysis used will affect the precision of the final results. Whether it is possible to obtain results as precise as those obtained by a flow-sensitive analysis is an open question.

*This work was supported in part by the National Science Foundation under grant CCR-9625656, and by the Army Research Office under grant DAAH04-85-1-0482.

1.1 Flow-sensitive vs. flow-insensitive analysis

Program analyses may be categorized as either flow-sensitive or flow-insensitive. A flow-sensitive analysis takes into account the order in which the statements in the program may be executed; a flow-insensitive analysis does not. In other words, in a flow-sensitive analysis the program is handled as a *sequence* of statements while in a flow-insensitive analysis it is handled as a *set* of statements. Thus, a flow-sensitive analysis produces results at the statement level (*e.g.*, it may discover different properties of a variable p at each statement) whereas a flow-insensitive analysis produces results at the program level (*e.g.*, it can only discover properties of a variable p that hold for the entire program).

(Analyses can be further categorized as context-sensitive or context-insensitive. A context-sensitive analysis takes into account the fact that a function must return to the site of the most recent call; a context-insensitive analysis propagates information from a call site, through the called function, and back to *all* call sites. In this paper, all analyses are assumed to be context-insensitive.)

One way to think about flow-insensitive analysis is in terms of a variation on the standard dataflow framework [Kil73]. The standard framework includes:

1. a lattice of dataflow facts,
2. a set of monotonic dataflow functions,
3. a control flow graph (CFG),
4. a mapping that associates one dataflow function with each graph node (we use f_n to denote the function mapped to node n).

The ideal goal of a flow-sensitive analysis is to find the meet-over-all-paths solution to the dataflow problem [Kil73]. When this is not feasible (*e.g.*, when the functions are not distributive), an acceptable goal is to find the greatest solution (under the lattice ordering) to the following set of equations (one equation for each CFG node n):

$$n.fact = \bigsqcap_{m \in predecessors(n)} f_m(m.fact) \qquad (1)$$

(This equation is for a *forward* dataflow problem. The equation for a *backward* dataflow problem is similar, with successors used in place of predecessors.)

Flow-insensitive analysis uses the same framework, except that it uses a version of the CFG in which there is

$a = 1$	$a_0 = 1$	
$b = 2$	$b_0 = 2$	
$c = a + b$	$c_0 = a_0 + b_0$	
if (...)	if (...)	(1)
then $a = 3$	then $a_1 = 3$	
	$a_2 = \phi(a_1, a_0)$	
$d = b$	$d_0 = b_0$	
$c = a + b$	$c_1 = a_2 + b_0$	(2)
print(c)	print(c_1)	(3)
(a) Original	(b) SSA Form	

Figure 1: Example for constant propagation

an edge from each node to every other node (including itself) [Hor97]. Again, the ideal goal of the analysis is to find the meet-over-all-paths solution (in the modified CFG), and when this is not feasible, to find the greatest solution to the set of equations:

$$n.fact = \prod_{m \in nodes(CFG)} f_m(m.fact) \qquad (2)$$

(Note that this framework is useful for *understanding* flow-insensitive analysis; actual algorithms do not involve creating this modified CFG or directly solving these equations.)

For the rest of this paper, when we refer to a flow-sensitive analysis, we mean an analysis that computes the greatest solution to the set of equations (1). Similarly, when we refer to a flow-insensitive analysis, we mean an analysis that computes the greatest solution to the set of equations (2).

Flow-sensitive analyses generally take more time and/or space than their flow-insensitive counterparts; however, the results are usually more precise. For example, consider constant propagation on the code fragment in Figure 1(a). A flow-sensitive constant-propagation analysis determines that:

- At[1] (1), $a = 1$, $b = 2$, $c = 3$
- At (2), $b = 2$, $c = 3$, $d = 2$
- At (3), $b = 2$, $d = 2$

and a flow-insensitive constant-propagation analysis determines that:

- $b = 2$, $d = 2$

Note that the results of the flow-insensitive analysis actually mean that at every point in the program, b is either uninitialized or has the value 2, and similarly for d. This kind of information is sufficient for most uses of the results of constant propagation (*e.g.*, replacing uses of constant variables with their values). Also note that, although the results of flow-insensitive analysis are not as precise as the results of flow-sensitive analysis, they do provide some useful information.

1.2 Using SSA form to improve flow-insensitive analysis

Static Single Assignment (SSA) form [CFR+91] is a program representation in which variables are renamed (via subscripting) and new definitions inserted to ensure that:

1. Each variable x_i has exactly one definition site.
2. Each use of a variable x_i is reached by exactly one definition.

The new definitions (called ϕ nodes) are inserted in the CFG at those places reached by two (or more) definitions of a variable x (the join points) and are of the form:

$$x_k = \phi(x_{i_1}, x_{i_2}, \ldots, x_{i_n})$$

Figure 1(b) shows the SSA form for the example from Figure 1(a). Notice that once the program has been put into SSA form, flow-sensitive and flow-insensitive constant propagation identify the same instances of constant variables in the code. For example, both a flow-sensitive and a flow-insensitive analysis on Figure 1(b) produce the following results (the results are shown in flow-sensitive format; the results from flow-insensitive analysis hold not just at the point given, but at every point in the program):

- At (1), $a_0 = 1$, $b_0 = 2$, $c_0 = 3$
- At (2), $a_0 = 1$, $b_0 = 2$, $c_0 = 3$, $a_1 = 3$, $d_0 = 2$
- At (3), $a_0 = 1$, $b_0 = 2$, $c_0 = 3$, $a_1 = 3$, $d_0 = 2$

In other words, it does not matter whether the constant propagation analysis done on the SSA form is flow-sensitive or flow-insensitive. Thus, if the time and space required to translate a program into SSA form and then perform a flow-insensitive analysis are less than the time and space required to do a flow-sensitive analysis, this approach is a win.

1.3 Points-to analysis

The presence of pointers in a program makes it necessary to have information about what pointer variables may be pointing to in order to do many program analyses (such as constant propagation) correctly. Thus, a points-to analysis must first be done on a program before any further analyses are done. (There are two kinds of points-to analyses, may and must. Whenever we use 'points-to' we mean 'may-point-to'.) This points-to analysis may be flow-sensitive or flow-insensitive. For example, consider the code fragment in Figure 2(a).

A flow-sensitive points-to analysis determines the following points-to information at (*i.e.*, immediately before) each program point ($p \to a$ means that p might point to a, $p \to \{a, b\}$ means that p might point to a or to b):

Point	Points-to information
(2)	$a \to w$
(3)	$a \to w$, $p \to a$
(4)	$a \to x$, $p \to a$
(5)	$a \to x$, $p \to a$, $c \to x$
(6)	$a \to x$, $p \to a$, $c \to x$
(7)	$a \to \{x, y\}$, $p \to a$, $c \to x$
(8)	$a \to \{x, y\}$, $p \to b$, $c \to x$
(9)	$a \to \{x, y\}$, $p \to b$, $c \to x$, $d \to \{x, y\}$
(10)	$a \to \{x, y\}$, $p \to b$, $c \to x$, $d \to \{x, y\}$, $b \to z$

A flow-insensitive points-to analysis determines the following information:

$$p \to \{a, b\}$$
$$a \to \{w, x, y, z\}$$
$$b \to \{y, z\}$$
$$c \to \{w, x, y, z\}$$
$$d \to \{w, x, y, z\}$$

[1] When we say that some fact is true *at* a particular point, we mean that the fact is true immediately before that point.

$a = \&w$	$a_0 = \&w_0$	(1)
$p = \&a$	$p_0 = \&a_0$	(2)
$a = \&x$	$a_1 = \&x_0$	(3)
$c = *p$	$c_0 = *p_0$	(4)
if (...)	if (...)	(5)
then $*p = \&y$	then $*p_0 = \&y_0$	(6)
$p = \&b$	$p_1 = \&b_0$	(7)
$d = a$	$d_0 = a_1$	(8)
$*p = \&z$	$*p_1 = \&z_0$	(9)
print($*a$)	print($*a_1$)	(10)
(a) Original	(b) Naive SSA	

Figure 2: An example with pointers and its naive translation to SSA form

As before, the flow-insensitive analysis is not as precise as the flow-sensitive analysis, but the information it does provide is *safe* (*i.e.*, the points-to sets computed by the flow-insensitive analysis are always supersets of the sets computed by the flow-sensitive analysis).

Given the advantages of SSA form discussed above in Section 1.2, it is natural to ask whether the approach of translating the program to SSA form and then using a flow-insensitive points-to analysis on the SSA form will achieve the same results as a flow-sensitive analysis on the original program. This approach seems reasonable since each variable x_k in the SSA form of the program corresponds only to *certain* instances of the variable x in the original program. Therefore, the 'whole-program' results of the flow-insensitive analysis of the SSA form could be mapped to 'CFG node specific' results in the original program. Unfortunately, this approach will not work.

The basic problem is that it is not possible to translate a program that contains pointers into SSA form without first doing some pointer analysis. For example, Figure 2(b) shows a naive translation to SSA form of the program shown in Figure 2(a). There are several problems with the naive translation. One problem is how the address-of operator (&) is handled. For example, in Figure 2(a) at line (2), p_0 is given the address of a_0. Clearly this is incorrect since it leads to the incorrect inference that the dereference of p_0 at line (4) is a use of a_0, when in fact it is a use of a_1, defined at line (3).

Another problem is that when a variable is defined indirectly via a pointer dereference, that definition is not taken into account in (naively) converting the program to SSA form. For example, at (6) the assignment to $*p$ is a definition of a (since at that point p contains the address of a). However, since variable a does not appear textually on the left-hand side of the assignment, the naive conversion to SSA form does not take this into account. The result is that the program in Figure 2(b) violates the first property of SSA form: that each variable x_i have exactly one definition site. Furthermore, because there is an (indirect) assignment at line (6), the use of a_1 at line (8) is reached by two definitions, thus violating the second property of SSA form.

Nevertheless, we believe that SSA form can be used to improve the results of flow-insensitive pointer analysis. An algorithm based on this idea is described below. The algorithm is iterative: it starts with purely flow-insensitive points-to information, and on each iteration it produces better information (*i.e.*, smaller points-to sets). We conjecture that when the algorithm reaches a fixed point (the last iteration produces the same points-to sets as the previous iteration) the final results mapped back to the original program will be the same as the results produced by a single run of a flow-*sensitive* pointer analysis algorithm.

Empirical studies are needed to determine how the time and space requirements of the iterative algorithm compare with those of a flow-sensitive algorithm. However, since the results of *every* iteration are *safe* (the points-to sets computed after each iteration are supersets of the actual points-to sets) the algorithm can also be safely terminated *before* a fixed point is reached (for example, after a fixed number of iterations, or when two consecutive iterations produce results that are sufficiently similar). This means that the algorithm can be 'tuned' to produce results that fall along the spectrum from flow-insensitive to flow-sensitive analysis.

2 Algorithm description

The main insight behind the algorithm is that we can use the results of (flow-insensitive) pointer analysis to normalize a program, producing an intermediate form that has two properties:

1. There are no pointer dereferences.
2. The points-to sets of all variables in the intermediate form are safe approximations to (*i.e.*, are supersets of) the points-to sets of all the variables in the original program.

Property 1 means that the intermediate form can be translated into SSA form. Property 2 means that flow-insensitive pointer analysis on the SSA form produces results that are valid for the original program.

When flow-insensitive pointer analysis is done on the SSA form, the results are in terms of the SSA variables. However, each SSA variable x_i corresponds to certain instances of the variable x in the original program. This means that the points-to set for each x_i can be mapped back to those instances of x in the original program that correspond to x_i. Note that in doing this we are producing points-to results that are no longer flow-insensitive, *i.e.*, a variable x may now have different points-to sets at different places in the program. This results in points-to sets that are often more precise than the sets produced by the initial analysis (done on the original non-SSA form of the program). These improved points-to sets can then be used to (re)normalize the program, producing a new intermediate form. If the new intermediate form is different from the previous one, the process of converting the intermediate form to SSA form, doing pointer analysis, and renormalizing can be repeated until a fixed point is reached (no change is made to the intermediate form).

Figure 3 gives an overview of the algorithm. Initially, we will assume that the input program consists of a single function with no function calls. In Section 2.1 we describe how to handle programs with multiple functions and functions calls.

The algorithm first applies flow-insensitive pointer analysis to the CFG, then uses the results to annotate each pointer dereference in the CFG with its points-to set. Only pointer dereferences are annotated because the places where we are ultimately interested in knowing about points-to information are the places where pointers are dereferenced. Note that the CFG itself is never changed, except for the annotations.

Given: a CFG G
Do flow-insensitive pointer analysis on G
Annotate the dereferences in G
Repeat:
    Create the intermediate form (IM) from G
    Convert IM to SSA form creating IM_{SSA}
    Do flow-insensitive pointer analysis on IM_{SSA}
    Update the annotations in G using IM_{SSA}
        and the pointer analysis results
until there are no changes in the annotations

Figure 3: An overview of the algorithm

Example: Figure 4(a) gives an example in which each pointer dereference has been annotated using the results of flow-insensitive points-to analysis. The annotations are shown to the right of each node containing a pointer dereference. For comparison, note that a flow-sensitive points-to analysis would determine that at the dereference of t, $t \rightarrow s$ and at the dereference of s, $s \rightarrow q$. □

The main loop of the algorithm begins by using the annotated CFG to create the (normalized) intermediate form (IM). In the intermediate form, each pointer dereference is replaced with its points-to set. If the points-to set contains more than one element, the single original statement is replaced with a multiway branch in which the k^{th} arm of the branch contains a copy of the original statement with the pointer dereference replaced by the k^{th} element of the points-to set. If the points-to set contains only one element, then rather than creating a branch, the pointer dereference is just replaced with the element in the points-to set.

The intermediate form is then converted to SSA form in two phases. In the first phase, conversion to SSA form is done as usual (ϕ nodes are added and variables are renamed via subscripting) with the exception that the operands of the address-of operator are not given subscripts, *i.e.*, an assignment of the form $p = \&x$ is converted to $p_i = \&x$; all other (non-address-of) uses and definitions of x are subscripted. In the second phase, each assignment of the form $p_i = \&x$ is converted to a multiway branch. The number of arms of the branch is the number of subscripts that x has in the SSA form. The k^{th} arm of the branch is of the form $p_i = \&x_k$. (As in the translation to intermediate form, if x only has one subscript, we just replace $\&x$ with $\&x_0$.)

The purpose of the second phase is to handle the first problem with translating a program with pointers to SSA form discussed in Section 1.3. Since a pointer that is given the address of x could be pointing to *any* of the SSA versions of x, using all possible versions in place of the address-of expression is a safe translation. Note that because we have replaced each pointer dereference with its points-to set we no longer have the problems mentioned in Section 1.3 that arise from the indirect definition of variables through pointer dereferences. Note also that after the second phase, the intermediate form may not be strictly in SSA form because the transformation of $p_i = \&x$ may result in multiple assignments to p_i. However, this will not affect the pointer analysis (which is the only way in which we are using this form). An equivalent way to handle $p_i = \&x$ would be to convert it as described, followed by inserting a ϕ node, and renaming the p_i's in the arms of the branch. In either case, the net result is that the definition of p that is live immediately after the transformation of $p_i = \&x$ has all SSA versions of x in its points-to set.

Example: Figure 4(b) shows the intermediate form for Figure 4(a). The intermediate form after the first phase in the conversion to SSA form is shown in Figure 4(c) and Figure 4(d) shows the final SSA form. □

The next step is to do flow-insensitive points-to analysis on the SSA version of the intermediate form (which we denote by IM_{SSA}). The results of this pointer analysis are then used to update the annotations in the CFG as follows: For each CFG node N with a pointer dereference $*p$:

- Find the corresponding node N' in IM_{SSA}. (If the node has been converted to a multiway branch construct, the branch node is the corresponding node.) Recall that all pointer dereferences were replaced with their corresponding points-to sets during the creation of the intermediate form and thus the dereferenced variable p itself is not present in N'.
- Determine the SSA number k that p would have had at node N' if it appeared there.
- Use the points-to set for p_k to update the annotation of $*p$ in node N of the CFG.

The updating of the annotations completes one iteration of the algorithm.

Example: Points-to analysis on IM_{SSA} (Figure 4(d)) determines that:

$$s_0 \rightarrow p$$
$$s_1 \rightarrow q$$
$$t_0 \rightarrow s$$

Note that because of the way the address-of operator is handled, if x_i is in p_k's points-to set, then x_j is in p_k's points-to set for all $j \in \{0, 1, 2, \ldots, \max\_SSA\_\#(x)\}$. Thus, the points-to sets can be represented in canonical form by using variables without subscripts.

The node in IM_{SSA} that corresponds to the node $*s = 2$ in the original CFG (Figure 4(a)) is the branch node that has $p_1 = 2$ and $q_1 = 2$ as its arms. The SSA number that s would have been given if it had appeared in that branch node is 1. Because the analysis has determined that $s_1 \rightarrow q$, the node $*s = 2$ in the original CFG is annotated with $s \rightarrow \{q\}$. Figure 5(a) shows the original program with updated annotations. □

Once the annotations have been updated, the process of creating an intermediate form, converting it to SSA form, doing pointer analysis, and obtaining better annotations can be repeated. Notice that if the annotations are the same for two different iterations, then the intermediate forms created using the annotations will be identical. Thus, when no annotations are changed during the updating stage of the algorithm (*i.e.*, the annotations are the same for two successive iterations), the algorithm has reached a fixed point (*i.e.*, no new pointer information can be discovered) and the algorithm halts. Since the results of *every* iteration are *safe*, the algorithm may also be halted after a user-specified number of iterations (just after updating the annotations), resulting in pointer information that lies somewhere in between the results from a purely flow-insensitive analysis and the results had the algorithm been run to completion.

Example: Figure 5(a) shows the CFG with its annotations updated using the results of the first iteration. Figures 5(b), (c), and (d) illustrate the start of the second iteration (the intermediate form and the two-phase conversion to SSA form). Points-to analysis on Figure 5(d) determines that:

$$s_0 \rightarrow p$$
$$s_1 \rightarrow q$$
$$t_0 \rightarrow s$$

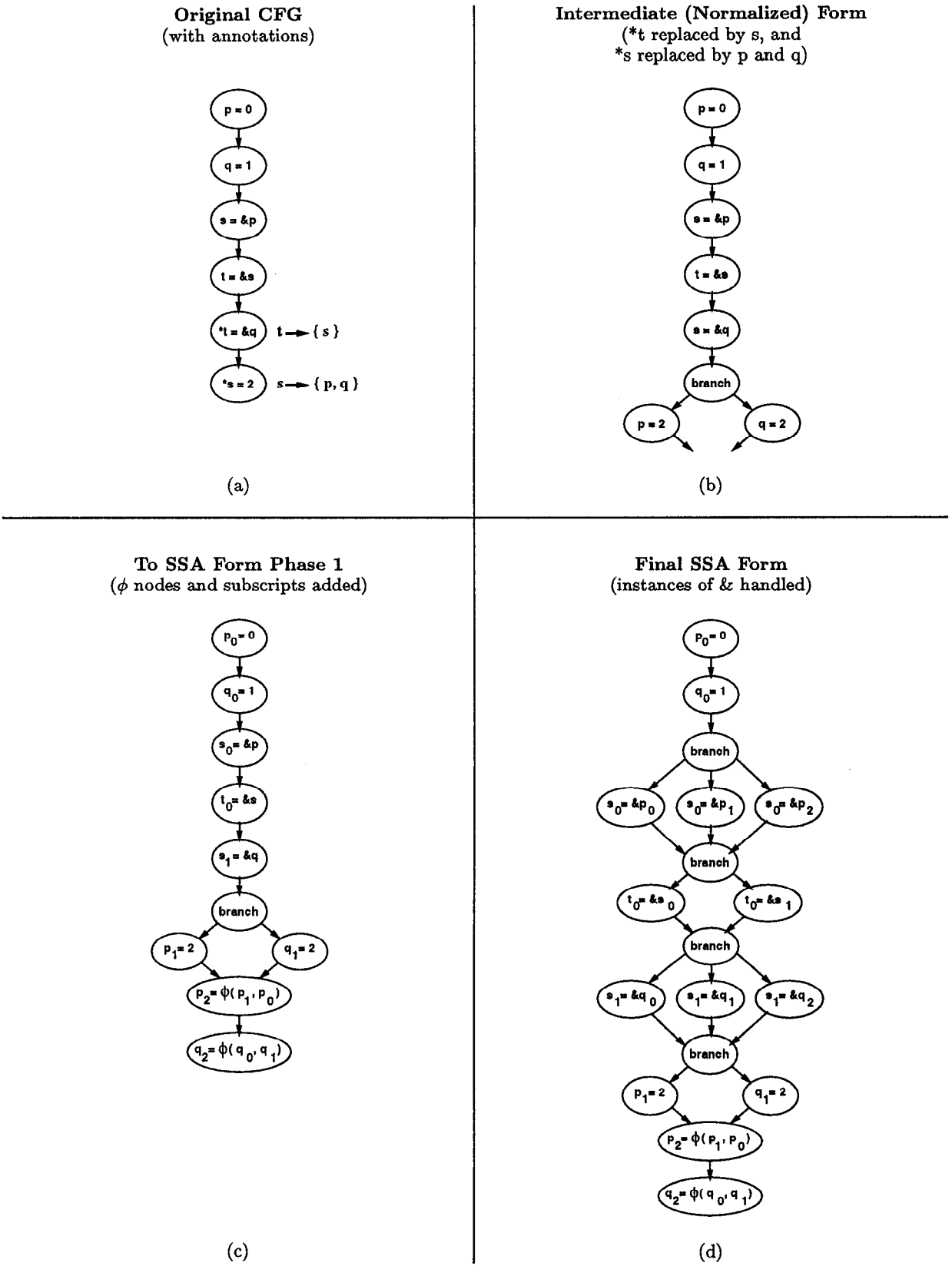

Figure 4: Translation to SSA form (first iteration)

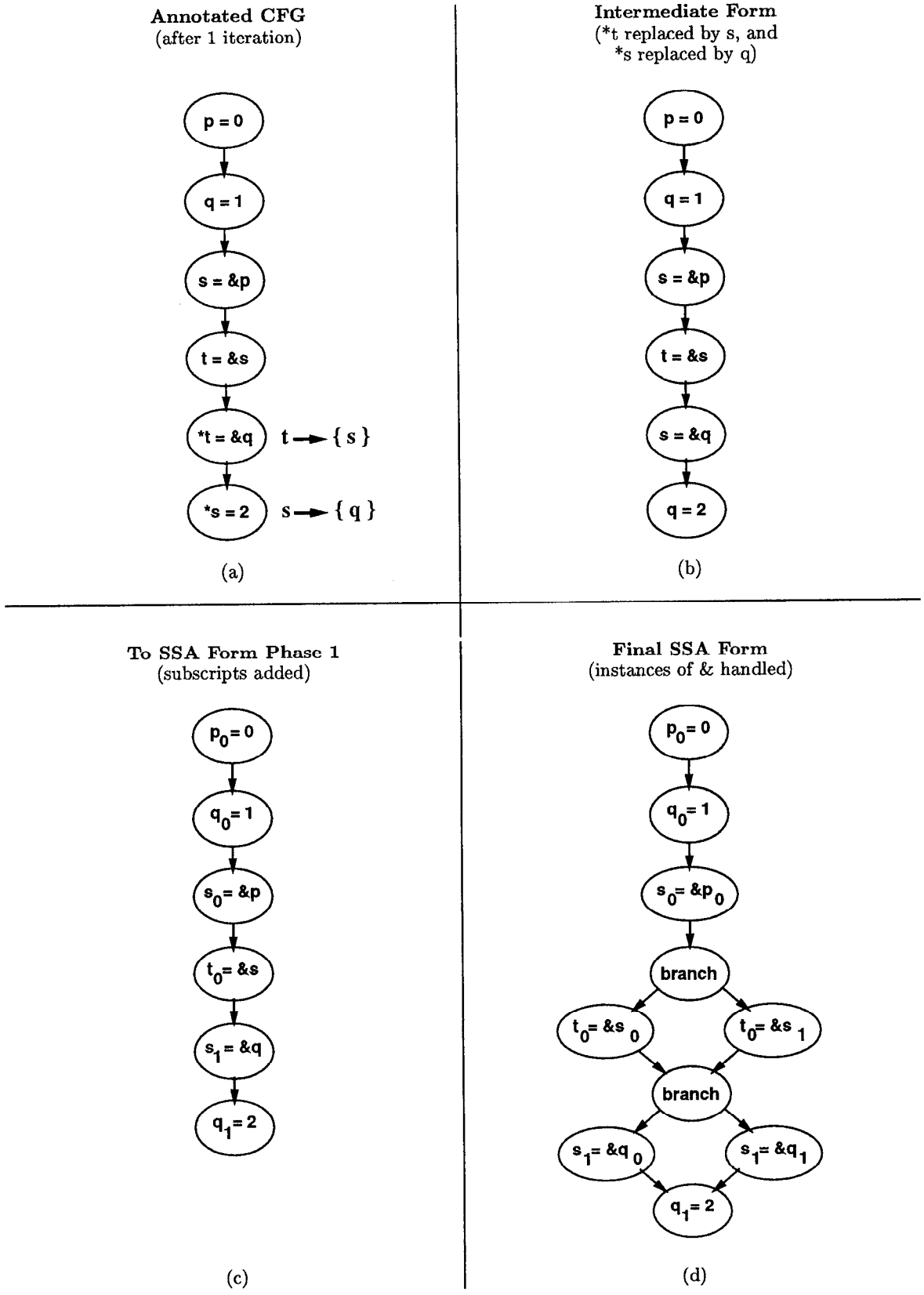

Figure 5: Translation to SSA form (second iteration)

```
int g, h;

void f()
{
  h = g;
  g = 0;
}

void main()
{
  int i;
  g = 3;
  f();
  i = g;
  if (...)
    g = 4;
  else
    f();
}
```

Figure 6: A program with multiple functions

This is the same as the information determined by the first iteration; thus, the CFG annotations do not change and the algorithm terminates after the second iteration. Note that the final results are the same as the flow-sensitive analysis on the original program. □

2.1 Handling multiple functions and function calls

A program that contains multiple functions can be represented by a set of CFGs, one for each function. However, there are problems with translating functions represented this way to SSA form when the program includes global variables. Figure 6 shows an example C program that illustrates two problems that arise when global variables are present.

One problem arises because a global variable may be used in a function before any definition of it appears in that function. For example, in the function f, global variable g is used in the assignment to h before any assignment to g. The difficulty is in determining the SSA number to give such a use. Another problem is that, because a function can modify a global variable, a use of a global variable that appears after a call may not be reached by the definition before the call. For example, in the function *main*, the value of g in the assignment to i is 0 (from the assignment $g = 0$ in f) and not 3 (from the assignment $g = 3$ before the call to f) and hence the g in $i = g$ should not have the same SSA number as the g in $g = 3$.

One way that these problems could be handled is to pass the global variables used or modified by a function as explicit parameters, and to treat the function call as an assignment to all of the global variables modified by the function.

A simpler approach is to create a supergraph[2] from the set of CFGs. The supergraph contains all nodes and edges of the original CFGs, including a call node and a return-point node for each function call. Additional edges are added from each call node to the entry node of the called function, and from the exit node of the called function to the call's return-point node. Figure 7(a) shows the CFGs for the program in Figure 6, and Figure 7(b) shows the corresponding supergraph.

Calls through function pointers are represented using a multiway branch in which the k^{th} arm of the branch contains a call to the k^{th} element of the function pointer's points-to set. This requires that pointer analysis be done before the supergraph is created. Moreover, the points-to sets for function pointers may change (*i.e.*, get smaller) during iteration, so the supergraph may need to be updated (by removing some of the arms of the multi-way branches that represent calls through function pointers) when annotations are changed. Figure 8 gives the algorithm from Figure 3 updated to handle multiple CFGs.

[2]The term *supergraph* was first used by Eugene Myers in [Mye81]. William Landi and Barbara Ryder [LR91] use the term *interprocedural control flow graph* (ICFG).

```
Given: a list L of CFGs
Do flow-insensitive pointer analysis on L
For each CFG G in L
    Annotate the dereferences in G
Create the supergraph S for L
Repeat:
    Create the intermediate form (IM) from S
    Convert IM to SSA form creating IM_SSA
    Do flow-insensitive pointer analysis on IM_SSA
    For each CFG G in L
        Update the annotations in G using
        IM_SSA and the pointer analysis results;
        update calls through function pointers in S
until there are no changes in the annotations
```

Figure 8: The algorithm updated to handle multiple CFGs

2.2 Complexity

Each iteration of our algorithm requires a transformation to SSA form and a flow-insensitive pointer analysis.

Although there exists a linear-time algorithm for placing ϕ nodes [SG95], the renaming phase of translation to SSA form can take cubic time in the worst case. Thus, in the worst case, the time needed to completely translate a program into SSA form (including renaming) is cubic. Moreover, the resulting program can be quadratic in the size of the original program. However, experimental evidence suggests that both the time to translate and the size of the translated program are linear in practice [CFR+91] [CC95].

Andersen [And94] gives a flow-insensitive pointer-analysis algorithm that computes the greatest fixed point of the set of equations (2) given in Section 1.1. Andersen's algorithm is cubic in the worst case. Experimental evidence intended to evaluate the algorithm's performance in practice[SH97] is inconclusive: on small programs (up to about 10,000 lines) its performance is very similar to that of Steensgaard's (essentially) linear-time algorithm[Ste96]; however, lines of code alone does not seem to be a good predictor of runtime (for example, one 6,000 line program required over 700 CPU seconds, while several 7,000 line programs required only 3 seconds). Note that our algorithm could make use of a fast algorithm like Steensgaard's. However, Steensgaard's algorithm does *not* always compute the greatest fixed point of the set of equations (2). Therefore, while the final result produced by our algorithm would still be an improvement over a purely flow-insensitive analysis, it is unlikely that it would be as good as a flow-sensitive analysis that computes the greatest fixed point of the set of equations (1).

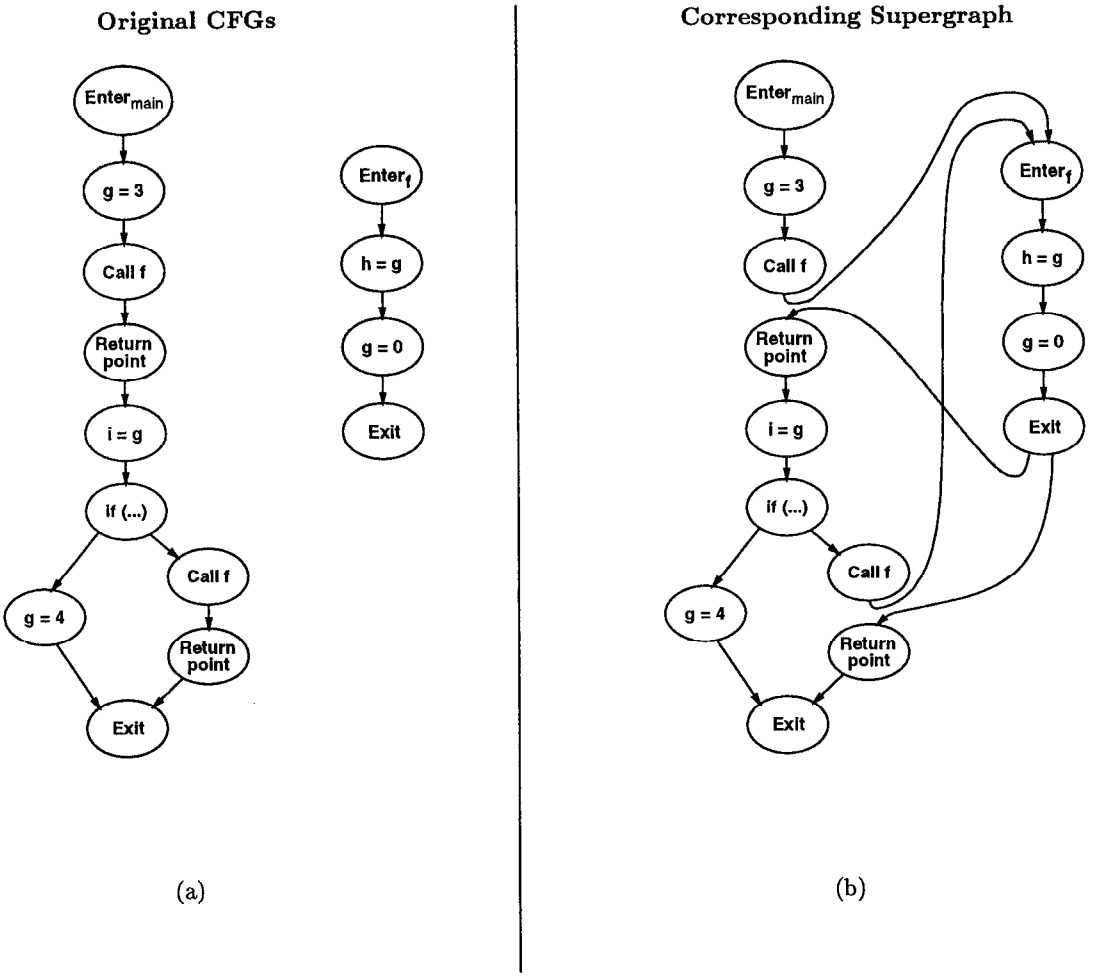

Figure 7: The CFGs and supergraph corresponding to the code in Figure 6

3 Related Work

A program representation similar to the intermediate form described here was used by Cytron and Gershbein in [CG93], where they give an algorithm for incrementally incorporating points-to information into SSA form. Our intermediate representation is essentially an in-lined version of Cytron and Gershbein's *IsAlias* function. However, their algorithm requires pre-computed may-alias information and incorporates points-to information as needed into a partial SSA form while solving another dataflow problem (constant propagation, in their paper).

Lapkowski and Hendren [LH96] also discuss the problems with SSA form in the presence of pointers. However, they abandon SSA form and develop instead a related analysis called SSA Numbering.

Others have worked on improving the precision of flow-insensitive alias analysis. In [BCCH94] Burke et al. develop an approach that involves using pre-computed *kill* information, although an empirical study by Hind and Pioli [HP97] does not show it to be more precise in practice than a flow-insensitive analysis. Shapiro and Horwitz [SH97] give an algorithm that can be 'tuned' so that its precision as well as worst-case time and space requirements range from those of Steensgaard's (almost linear, less precise flow-insensitive) algorithm to those of Andersen's (cubic worst-case but more precise flow-insensitive) algorithm.

4 Conclusions

We have presented a new iterative points-to analysis algorithm that uses flow-insensitive pointer analysis, a normalized intermediate form, and translation to SSA form. The results after just one iteration are generally better than those of a purely flow-insensitive analysis (on the original program) and if the algorithm is run until the fixed point is reached, the results may be as good as those of a flow-sensitive analysis.

We are currently working on implementations of our algorithm using the flow-insensitive pointer analyses defined in [And94], [Ste96], and [SH97]. We plan to use the implementations to explore how our algorithm compares to flow-sensitive points-to analysis in practice.

5 Acknowledgement

Thanks to Charles Consel, whose question about using SSA form in pointer analysis inspired this work.

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).

[BCCH94] M. Burke, P. Carini, J.D. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In K. Pingali, U. Banerjee, D. Galernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing: Proceedings of the 7th International Workshop*, volume 892 of *Lecture Notes in Computer Science*, pages 234–250, Ithaca, NY, August 1994. Springer-Verlag.

[CC95] C. Click and K.D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, 1995.

[CFR+91] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CG93] R. Cytron and R. Gershbein. Efficient accommodation of may-alias information in SSA form. *SIGPLAN Conference on Programming Language Design and Implementation*, 28(6):36–45, June 1993.

[Hor97] S. Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, January 1997.

[HP97] M. Hind and A. Pioli. An empirical comparison of interprocedural pointer alias analyses. IBM Research Report RC 21058, IBM Research Division, December 1997.

[Kil73] G.A. Kildall. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206, January 1973.

[LH96] C. Lapkowski and L.J. Hendren. Extended SSA numbering: Introducing SSA properties to languages with multi-level pointers. ACAPS Technical Memo 102, School of Computer Science, McGill University, Montréal, Canada, April 1996.

[LR91] W. Landi and B.G. Ryder. Pointer induced aliasing: A problem classification. In *ACM Symposium on Principles of Programming Languages*, pages 93–103, 1991.

[Mye81] E.W. Myers. A precise inter-procedural data flow algorithm. In *ACM Symposium on Principles of Programming Languages*, pages 219–230, 1981.

[SG95] V.C. Sreedhar and G.R. Gao. A linear time algorithm for placing ϕ-nodes. In *ACM Symposium on Principles of Programming Languages*, pages 62–73, 1995.

[SH97] 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.