Unification-based Pointer Analysis with Directional Assignments

Manuvir Das
manuvir@microsoft.com

Microsoft Research
Redmond, WA 98052

Abstract

This paper describes a new algorithm for flow and context insensitive pointer analysis of C programs. Our studies show that the most common use of pointers in C programs is in passing the addresses of composite objects or updateable values as arguments to procedures. Therefore, we have designed a low-cost algorithm that handles this common case accurately. In terms of both precision and running time, this algorithm lies between Steensgaard's algorithm, which treats assignments bi-directionally using unification, and Andersen's algorithm, which treats assignments directionally using subtyping. Our "one level flow" algorithm uses a restricted form of subtyping to avoid unification of symbols at the top levels of pointer chains in the points-to graph, while using unification elsewhere in the graph. The method scales easily to large programs. For instance, we are able to analyze a 1.4 MLOC (million lines of code) program in two minutes, using less than 200MB of memory. At the same time, the precision of our algorithm is very close to that of Andersen's algorithm. On all of the integer benchmark programs from SPEC95, the one level flow algorithm and Andersen's algorithm produce either identical or essentially identical points-to information. Therefore, we claim that our algorithm provides a method for obtaining precise flow-insensitive points-to information for large C programs.

1 Introduction

Programs written in C typically make widespread use of pointer variables. In order to analyze a program that uses pointers, it is necessary to perform a pointer analysis that computes, at every dereference point in a program, a superset of the set of symbols or memory locations that may be accessed via the given dereference. The precision of a pointer analysis is determined by the sizes of these "points-to" sets. Points-to information is useful for any subsequent program analysis, whether this client analysis is an optimization algorithm, a code browsing tool, or a program transformation. Experiments have shown that the precision of points-to information can greatly impact the performance and usefulness of subsequent analyses [SH97a].

The focus of this paper is flow and context insensitive pointer analysis. Within this domain, there are algorithms such as Steensgaard's unification-based approach [Ste96b] (almost linear running-time), which scale well to large programs but produce relatively imprecise results. On the other hand, algorithms such as Andersen's subtyping-based approach [And94] (worst-case cubic running-time) produce much more precise results, but do not scale well to large programs. The challenge has been to design an analysis that has the scaling properties of Steensgaard's analysis and the precision of Andersen's analysis. While some algorithms have been proposed in the literature, it is unclear whether any of them provides the right solution (see Section 5).

Our manual studies of C programs indicate that by far the greatest use of pointers in C programs is in passing the addresses of composite objects or updateable values as arguments to procedures. In such cases, a method that produces accurate information for "top level" pointers (*i.e.*, pointers that are not themselves pointed to by other pointers) can yield the same precision as a method that is accurate for all pointers. Therefore, we have designed a new low-cost algorithm for pointer analysis that handles this common case accurately. Our one level flow algorithm improves upon Steensgaard's unification-based algorithm by avoiding unification of symbols at the top levels of pointer chains in the points-to graph. The method can also be interpreted as a restriction of Andersen's subtyping rules at assignments to one level in the type structure.

In this paper, we describe our algorithm. We make the following contributions:

- We present the one level flow algorithm, a new pointer analysis algorithm that lies between the algorithms of Steensgaard and Andersen, in terms of both running-time and precision. Its asymptotic complexity is quadratic in the size of the program.
- Our algorithm is a simple extension of Steensgaard's algorithm, and is easy to implement quickly and correctly. In can be added to any unification-based pointer analysis algorithm.
- We demonstrate that in practice, the algorithm has the scaling properties of Steensgaard's algorithm, and the precision of Andersen's algorithm.
  - On all of our test programs, including the integer programs from SPEC95 and a 1.4 MLOC pro-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI 2000, Vancouver, British Columbia, Canada.
Copyright 2000 ACM 1-58113-199-2/00/0006...$5.00.

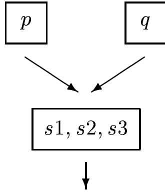
```
foo(&s1);
foo(&s2);
bar(&s3);

foo(struct s *p) { *p.a = 3; bar(p); }
bar(struct s *q) { *q.b = 4; }
```

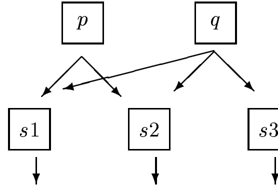
(a)

```
p = &s1;
p = &s2;
q = &s3;
q = p;
*p.a = 3;
*q.b = 4;
```

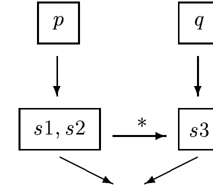
(b)



(c)



(d)



(e)

Figure 1: Two programs that illustrate the difference between the algorithms of Steensgaard and Andersen. The program in (a) above represents the common case in C programs, while the program in (b) above is a variant of the program without procedure calls. Figures (c), (d) and (e) above show the points-to graphs computed by Steensgaard's algorithm, Andersen's algorithm, and our one level flow algorithm, respectively, for the program in (b) above. The edge labeled with * is a flow edge.

gram (Microsoft Word 97), the one level flow algorithm scales linearly in both time and space requirements. It uses twice as much time and memory as Steensgaard's algorithm.

  - On all of our test programs except Word 97, to which Andersen's algorithm does not scale, the one level flow algorithm and Andersen's algorithm provide either identical or very similar precision.

- We compare our algorithm with Andersen's algorithm on real, large programs, up to 140,000 LOC (*gcc*). Beyond comparing precision in terms of the average sizes of points-to sets at static dereference points, we show that for each benchmark, the entire distributions of points-to set sizes using the two algorithms are very similar.

The rest of the paper is organized as follows: In Section 2, we present a motivating example, and we use this example to explain the main idea of our algorithm. In Section 3, we describe our one level flow algorithm in detail. In Section 4, we describe our experimental framework, and present our empirical results. We discuss related work in Section 5, and make concluding remarks in Section 6.

2 Example

Example 1 A fragment of a C program is shown in Figure 1 (a). This program represents the most common case of pointer usage across our test programs. A variant of this program without procedure calls is shown in Figure 1 (b). A flow and context insensitive pointer analysis would produce the same points-to sets for both variants of the program.

Figure 1 (c) shows the points-to information computed by Steensgaard's algorithm for the program in Figure 1 (b). The points-to graph shown contains nodes representing equivalence classes of symbols and edges representing pointer relationships. Every node contains a single pointer edge. Steensgaard's algorithm processes assignments bidirectionally: The left hand side and right hand side memory locations in an assignment are constrained to hold the same contents[1]. For the example program, the effect of the assignment from p to q is to force both p and q to point to the equivalence class containing $s1$, $s2$, and $s3$, even though p cannot point to $s3$ in any execution of the program.

Figure 1 (d) shows the points-to information computed by Andersen's algorithm. In this case, every node is associated with a single symbol and contains a set of pointer values. Assignments are processed directionally: The contents of the right hand side location are copied to the left hand side. This method produces more accurate points-to sets as a result. In the example program, p is not required to point to $s3$.

Figure 1 (e) shows the points-to information computed by our one level flow algorithm. Our points-to graph is similar to that of Steensgaard, but includes special "flow" edges between nodes. Flow edges are introduced at assignments, one level below (in the points-to graph) the symbols involved in the assignment. In the example program, the assignment from p to q introduces a flow edge from the pointer target node of p to the pointer target node of q. A flow edge from one node to another indicates that all of the symbols associated with the first node must be associated with the second. Thus, the pointer target node of p contains $s1$ and $s2$, whereas the pointer target node of q contains both of

[1] A limited form of subtyping allows an exception to this rule, in cases where the right hand side does not hold any pointer value.

those symbols, as well as $s3$. All of the nodes related by flow edges are constrained to have equal contents; that is, they are required to hold pointers to the same node. If any of $s1$, $s2$ or $s3$ point to a given symbol, our algorithm will require that all of $s1$, $s2$ and $s3$ point to that symbol, thereby losing precision compared to Andersen's algorithm.

Andersen's algorithm achieves directionality in assignments by using subtyping rules. In order to accommodate directionality, it is necessary to allow unlimited fanout, or outdegree, in the points-to graph. This leads to trees with fanout at all levels. The algorithm is expensive because of the work required to track the subtyping relations induced at all levels of the points-to graph. Steensgaard's algorithm uses type equality rules to merge (unify) equivalence classes of symbols at assignments, leading to nodes with single outdegree in the points-to graph. The use of type equality rules makes Steensgaard's analysis inexpensive, but approximate.

The difference between the two algorithms can be viewed in terms of the final points-to graph, or in terms of the analysis itself. Shapiro and Horwitz viewed the difference in terms of the final points-to graph. Therefore, they designed an algorithm that allows k-limited outdegree at all levels of the points-to graph [SH97b]. We believe the interesting difference is really in the analysis itself. Therefore, our algorithm allows some amount of directionality (using subtyping) at assignments. In terms of the final points-to graph, our algorithm can be viewed as allowing unlimited outdegree in a few critical areas of the graph, while restricting outdegree to single edges everywhere else.

3 One level flow algorithm

In this section, we present our one level flow algorithm. We first describe the algorithm informally as an extension of Steensgaard's unification-based approach, and then present the algorithm formally as a set of non-standard type inference rules over a pointer language. We show how our algorithm can be viewed as a restriction of Andersen's algorithm. Finally, we discuss correctness and asymptotic complexity.

3.1 Algorithm description

A pointer analysis can be thought of as an abstract computation that models memory locations. Every location τ is associated with a label or set of symbols φ and holds some contents α (an abstract pointer value) (Figure 2). A location "points-to" another if the contents of the former is a pointer to the latter. Information about locations can be encoded as a points-to graph, in which nodes represent memory locations and edges represent points-to relationships.

In Steensgaard's algorithm, the effect of an assignment from y to x is to equate the contents of the locations associated with y and x. This is achieved by unifying (*i.e.*, equating their labels and contents) the locations pointed-to by y and x into one representative location (Figure 2). This unification may further require the pointed-to locations of the pointed-to locations to be merged, and so on, resulting in chains in the points-to representation. The unification is represented declaratively through the type rule in Figure 2 (d). The rule says that the program is correctly typed (that is, symbols are associated with correct points-to sets) if and only if the pointed-to locations of y and x are identical. Steensgaard's algorithm is extremely efficient, because pointed-to locations can be found and unified in close to constant time (amortized). It has a linear space requirement, and almost linear running-time (in the size of the program).

Our one level flow algorithm extends Steensgaard's algorithm by pushing the effect of assignment processing one level down the chains in the points-to graph (Figure 3). The effect of an assignment from y to x is to introduce a special "flow" edge from the pointed-to location of y to the pointed-to location of x, and to equate *only the contents* of the two pointed-to locations (Figure 3 (b)). The flow edge relates the labels of the pointed-to locations: For every flow edge, all of the symbols in the label of the source of the edge must be included in the label of the target of the edge. Assignment processing is represented declaratively through the type rule in 3 (d). This rule says that the program is correctly typed if and only if the pointed-to locations of y and x have the same contents, and if the label of the pointed-to location of y is a subset of the label of the pointed-to location of x. The rule allows a degree of subtyping in the labels of the pointed-to locations. Therefore, our algorithm is directional up to one level below symbols related by assignments. It has the same effect as Steensgaard's algorithm at all lower levels in the points-to representation.

3.2 Declarative specification

We present the algorithm as a set of non-standard type inference rules over a simple language of pointer related assignments (Figure 4). These rules are similar to the rules used by Steensgaard [Ste96b, pp. 35]. Every rule in Figure 4 extends the corresponding rule from [Ste96b] by introducing a new form of type inequality at the assignment level, as illustrated in Figure 3. The domains for the type inference described by these rules are as in Figure 3 (d).

The rule for $x = \&y$ says that the pointed-to location of x must include the symbol y in its label, and this pointed-to location must have the same contents as the location of y. The rule for $x = *y$ says that the pointed-to location of y and the location of x are related by an assignment, while the rule for $*x = y$ says that the location of y and the pointed-to location of x are related by an assignment.

The rules shown in Figure 4 operate on a toy pointer language. However, they represent the essence of the pointer analysis. The complete algorithm includes rules for function definitions and calls, function pointers, and indirect function calls. The type rules are extended to handle these features exactly as in Steensgaard's analysis. Function calls give rise to implicit assignments from arguments to formal parameters and from return expressions to calling contexts. These implicit assignments are processed with one level of flow, in exactly the same manner as explicit assignments. We omit details of the extensions for brevity.

3.3 Operational algorithm

Every symbol referenced in the program is associated with a unique location on demand. The program is processed one assignment at a time, including implicit assignments generated by function calls. At every assignment, locations are unified as necessary to satisfy the requirements imposed by the rules in Figure 4. We delay processing of the subset constraints between labels by introducing flow edges between locations, as shown in Figure 3 (c). The location resulting

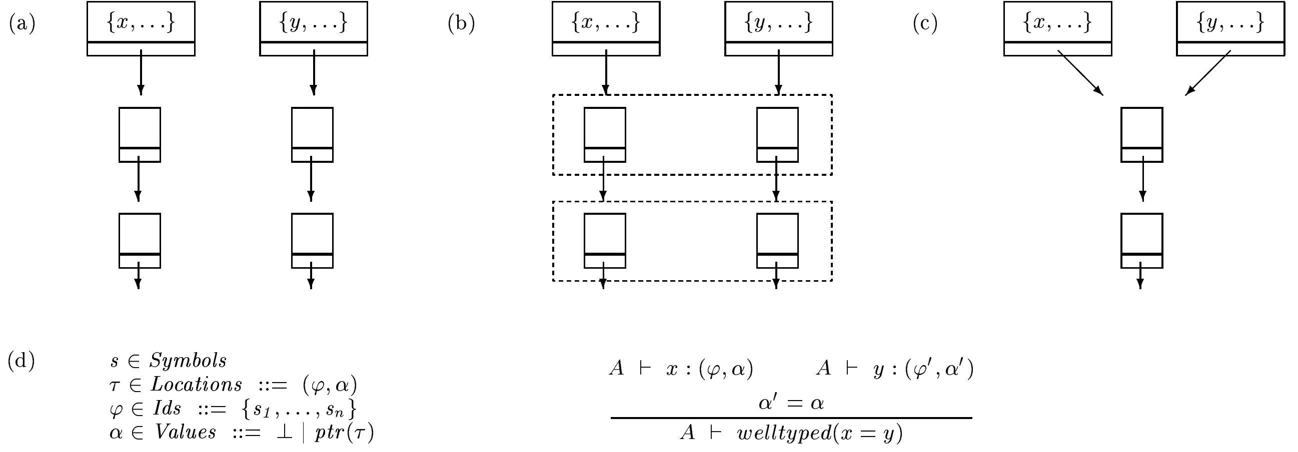


Figure 2: Assignment processing in Steensgaard's algorithm. The graphs in (a) and (c) above represent points-to information before and after processing $x = y$. Figure (b) depicts the unification actions, shown using dashed boxes, triggered by the assignment. The type rule in (d) above describes assignment processing declaratively.

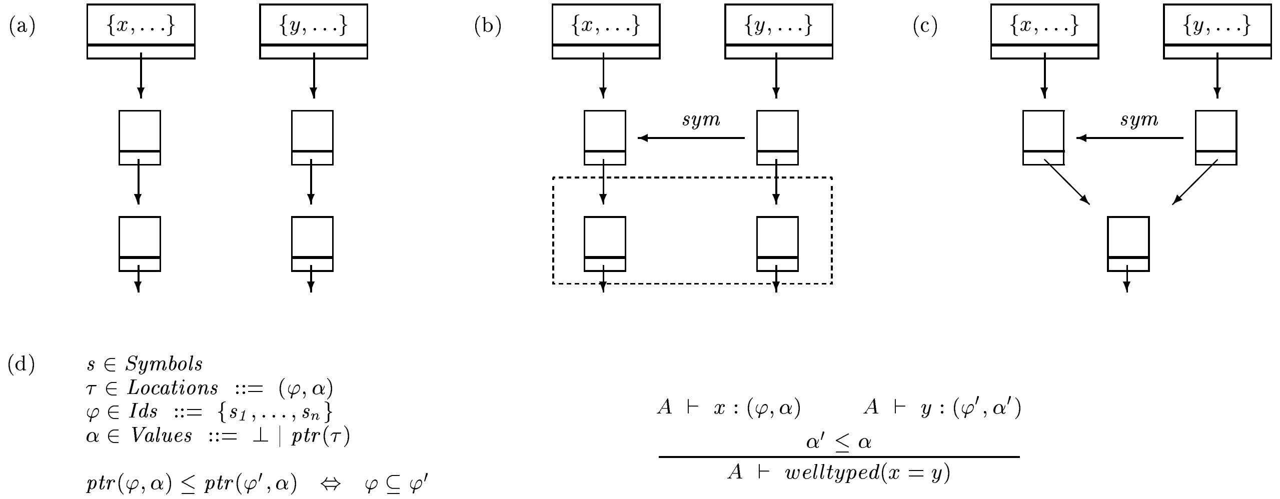


Figure 3: Assignment processing in the one level flow algorithm. The graphs in (a) and (c) above represent points-to information before and after processing $x = y$. Figure (b) depicts the actions triggered by the assignment. The edge labeled *sym* is a flow edge relating labels of locations. The domains and type rule in (d) above provide a declarative specification of assignment handling.

$$\frac{A \vdash x : (\varphi, \alpha) \qquad A \vdash y : (\varphi', \alpha') \qquad \alpha' \leq \alpha}{A \vdash \mathit{welltyped}(x = y)}$$

$$\frac{A \vdash x : (\varphi, \alpha) \qquad A \vdash y : \tau \qquad ptr(\tau) \leq \alpha}{A \vdash \mathit{welltyped}(x = \&y)}$$

$$\frac{A \vdash x : (\varphi, \alpha) \qquad A \vdash y : (\varphi', ptr(\tau)) \qquad \tau = (\varphi'', \alpha'') \qquad \alpha'' \leq \alpha}{A \vdash \mathit{welltyped}(x = *y)}$$

$$\frac{A \vdash x : (\varphi', ptr(\tau)) \qquad A \vdash y : (\varphi, \alpha) \qquad \tau = (\varphi'', \alpha'') \qquad \alpha \leq \alpha''}{A \vdash \mathit{welltyped}(*x = y)}$$

Figure 4: The one level flow algorithm, expressed as a set of type inference rules over a simple language of pointer related assignments.

from the unification of two locations inherits the flow edge successors of both the unified locations. Once processing of the entire program is complete, labels are computed for all of the locations in the points-to representation by a flow processing step: Every symbol is associated with every location that is reachable, via flow edges, from the location initially associated with the symbol.

3.4 Comparison with Andersen's algorithm

The one level flow algorithm can be viewed as a restriction of Andersen's algorithm, in the following way: We introduce subtyping rules in the label component of locations. Andersen's algorithm introduces subtyping in the contents component of locations as well. The effect of an assignment from y to x in Andersen's algorithm is shown in Figure 5. The rule in Figure 5 (a) says that the program is correctly typed if and only if the label of the pointed-to location of y is a subset of the label of the pointed-to set of x, and the contents of the pointed-to location of y is a subtype of the contents of the pointed-to location of x. The structural subtyping relations introduced by Andersen's algorithm make it expensive, but also lead to greater precision. His algorithm has a worst-case quadratic space requirement, and worst-case cubic running time.

Thus, the one level flow algorithm obtains the precision of Andersen's algorithm for top level pointers, while it obtains the same precision as Steensgaard's algorithm for all other pointers. The one level flow algorithm is not always less precise than Andersen's algorithm on pointed-to pointers. Consider the program in Example 1. The one level flow algorithm forces $s1$, $s2$, and $s3$ to hold the same contents. Suppose $s1$, $s2$, and $s3$ are locals. Then they can be referenced within *foo* and *bar* only indirectly, via $*p$ and $*q$. Even a fully directional analysis such as Andersen's algorithm must treat accesses of $*p$ and $*q$ identically, because updates of $*p$ must be reflected in $*q$, and vice versa. Thus, there is no further loss of precision in this case because of unification at lower levels. On the other hand, if $s1$ is updated explicitly to hold a pointer value, Andersen's algorithm will avoid associating this value with explicit references to the values of $s2$ and $s3$, thereby gaining precision over the one level flow algorithm. Our empirical evidence suggests that this case occurs infrequently in C programs.

3.5 Correctness

We claim that the type rules in Figure 4 represent a correct flow-insensitive pointer analysis. This follows from the observation that every rule in Figure 4 can be viewed as a restriction of the corresponding rule from a type system describing Andersen's algorithm, which we assume to be correct. The operational algorithm records flow edges at assignments to represent the subset relationships on labels imposed by the type rules. When the entire program is processed, every subset constraint is represented by some flow edge in the points-to graph. The only exception is when two nodes initially connected by a flow edge are unified into a single node, in which case the subset constraint is trivially satisfied by the resulting node. The final flow step is implemented as reachability on flow edges. This ensures that after all labels are computed, the labels at every pair of nodes connected with a flow edge satisfy the subset constraint represented by the corresponding flow edge.

3.6 Complexity

The algorithm has two steps: An assignment processing step, which produces a points-to graph with flow edges, and a flow propagation step. The first step has the same complexity as Steensgaard's algorithm: Linear space, and almost linear running-time (in the size of the input program). The number of nodes created during the first step is proportional to the number of assignments in the program. Every node has a single pointer edge. Finally, every assignment introduces a single flow edge. Therefore, the space requirement for the first step is linear. The running-time is determined by the cost of assignment processing. Every assignment requires lookup of four pointed-to locations, the addition of a single flow edge, and a unification step. A single unification, which takes constant time, may trigger other unifications, but the total number of unifications is bounded by the number of distinct nodes. The amortized cost of N lookup operations and M unification operations using a fast-union-find structure is $O(N\alpha(M, N))$ where α is the inverse Ackermann's function [Tar83]. Therefore, the initial phase of the algorithm has almost linear running time.

The flow step involves reachability over the linear graph structure, for every symbol associated with a node in the

(a)

$$s \in \mathit{Symbols}$$
$$\tau \in \mathit{Locations} ::= (\varphi, \alpha, \overline{\alpha})$$
$$\varphi \in \mathit{Ids} ::= \{s_1, \ldots, s_n\}$$
$$\alpha \in \mathit{Values} ::= \bot \mid \mathit{ptr}(\tau)$$

$$\mathit{ptr}(\varphi, \alpha, \overline{\alpha}) \leq \mathit{ptr}(\varphi', \alpha', \overline{\alpha}') \Leftrightarrow \varphi \subseteq \varphi' \wedge \alpha \leq \alpha' \wedge \overline{\alpha}' \leq \overline{\alpha}$$

$$\frac{A \vdash x : (\varphi, \alpha, \overline{\alpha}) \qquad A \vdash y : (\varphi', \alpha', \overline{\alpha}') \qquad \alpha' \leq \overline{\alpha}}{A \vdash \mathit{welltyped}(x = y)}$$

(b)

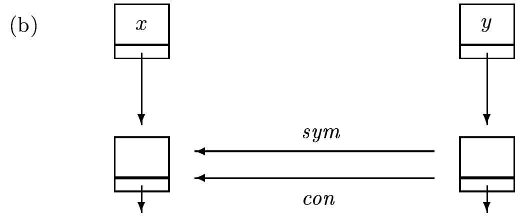


Figure 5: Assignment processing in Andersen's algorithm. The domains and type rule in (a) above, taken from [FFSA98], provide a declarative specification of assignment handling. The graph in (b) above represents the effect of the assignment action, which is to induce subtyping relationships between the pointed-to locations of y and x. These are represented pictorially by edges labeled *sym* and *con*, which are flow edges relating labels and contents, respectively. Note that the graph in (b) is a simplification of the type rule in (a), which induces flow edges on contents in both positive and negative directions in the points-to representation.

graph. In the worst case, each symbol is added to the set of symbols (using constant-time append operations) at every node. Therefore, this step has worst-case quadratic time and space requirements. The flow step can also be performed in a demand-driven manner. The points-to set at a single dereference point can be obtained by a single linear walk of the points-to representation.

4 Experiments

In this section, we first describe our implementation of the one level flow algorithm, and our experimental setup. We then present our results.

4.1 Implementation and experimental setup

We have produced a modular implementation of Steensgaard's algorithm using the AST Toolkit [AST], which is itself an extension of the Microsoft Visual C compiler. Our implementation, which includes a client and a linker, handles all of the features of C. The client analyzes individual compilation units, and produces object files of a special kind. Each object file contains a partial points-to graph, and a table that associates symbols and functions with nodes in the graph. The linker produces global points-to information by unifying locations corresponding to the same symbol or function from different object files. Local and static symbols, which are also passed on to the linker, are assigned globally unique names. We model library functions for which source code is not available using hand-coded stub functions that simulate the pointer effects of the corresponding library functions. We treat library functions polymorphically, by cloning the partial points-to graphs corresponding to these functions at call sites. In order to model allocator functions, we automatically create a dummy symbol representing a unique memory location at each call to an allocator function. We do not model constant strings.

Our implementation includes a garbage collector, which periodically releases memory associated with locations that are forwarded by unification. Garbage collection is an important feature of our scalable pointer analysis; it allows us to process a 1.4 MLOC program without disk thrashing on a machine with 256MB of memory. Our base implementation includes a variety of precision-enhancing features (including pending lists, structure layout mapping, k-limited outdegree, and filtering of indirect calls using lengths of argument lists), which can be enabled using command line switches. In order to measure the effectiveness of our technique accurately, we did not enable any of these features (except pending lists) in our experiments.

We implemented the one level flow algorithm by extending the assignment rule in our base analysis, and adding a flow computation step to the link phase. These changes took less than a day to implement. Apart from all of the usual testing, we verified the correctness of our implementation in three ways: First, we compared our implementation against an independent implementation of Steensgaard's analysis produced by our group [FRD00], and against an implementation of Steensgaard's analysis produced using the BANE framework at UC Berkeley [FFSA98]. All three implementations produce identical points-to information for all of the SPECINT95 benchmark programs. Second, we tested our implementation of the flow computation step by treating flow edges bi-directionally and verifying that we obtain exactly the same points-to information as with Steensgaard's algorithm. Third, the striking similarity between the points-to information computed by the one level flow algorithm and Andersen's algorithm provides a strong indication of the correctness of the one level flow algorithm.

4.1.1 Comparison with Andersen's algorithm

Although we have implemented Steensgaard's algorithm and the one level flow algorithm, we have not implemented Andersen's algorithm. Therefore, we used an implementation of Andersen's algorithm based on the BANE framework to obtain precision measurements for Andersen's algorithm on all of our benchmark programs (except Microsoft Word, to which Andersen's algorithm does not scale). The BANE framework does not include an implementation of the one level flow algorithm. Further, BANE uses a different front-end than our infrastructure. This raises a natural question about the validity of our experiments, which compare the points-to information produced by all three algorithms on a set of benchmark programs. In order to answer this

Program	LOC	AST nodes	Analysis time (secs)		Average thru-deref size		
			Ste96	Flow	Ste96	And94	Flow
compress	1,904	2,234	0.03	0.05	2.1	1.22	1.22
li	7,602	23,379	0.43	0.67	287.7	185.62	185.62
m88ksim	19,412	65,967	0.79	1.22	86.3	3.19	3.29
ijpeg	31,215	79,486	0.97	1.51	17.0	11.76	11.78
go	29,919	109,134	0.89	1.42	45.2	14.79	14.79
perl	26,871	116,490	1.21	2.12	36.1	22.22	22.22
vortex	67,211	200,107	3.35	5.66	1,064.5	45.54	59.30
gcc	205,406	604,100	5.70	9.45	245.8	7.71	7.72
Word97	2,150,793	5,961,129	61.34	126.83	27,258.6		11,219.5

Table 1: Experimental data. For each benchmark program, the table above shows the lines of code and AST node count, the running time of Steensgaard's algorithm (Ste96) and the one level flow algorithm (Flow), and the average size of points-to sets at static dereference points for Steensgaard's algorithm (Ste96), Andersen's algorithm (And94), and the one level flow algorithm (Flow).

question, we spent several months synchronizing the two infrastructures so that they produce identical points-to information using Steensgaard's analysis. This process involved making several changes to the two infrastructures to ensure that library functions are treated polymorphically, symbols are counted similarly, etc. Because the two infrastructures have now been "normalized" to produce identical points-to information for Steensgaard's analysis, it is possible to directly compare the precision of the one level flow algorithm with that of Andersen's algorithm.

4.1.2 Benchmark programs

Table 1 shows our benchmark programs, consisting of the integer benchmarks from SPEC95, and a version of Microsoft Word. For each benchmark, we list the total lines of code in source files (including comments and blank lines), as well as the number of AST nodes, which we feel is a more accurate measure of program size.[2] Each benchmark was analyzed using Steensgaard's analysis, the one level flow algorithm, and Andersen's algorithm. We report the analysis time, averaged over 5 runs, for Steensgaard's analysis and the one level flow algorithm. Analysis time includes time to analyze each compilation unit (not including parse time), time to write out object files, and time taken by the link phase. Link phase time includes time to read in all of the object files, perform unifications, flow symbols, and compute points-to sets exhaustively at all static dereference points in the program. All of our experiments were conducted on a Dell 610 desktop PC running Windows NT 4, with 512MB RAM and a single 450Mhz Intel Xeon processor. We report the average size of points-to sets at static dereference points using all three algorithms. As mentioned earlier, we are unable to run Andersen's algorithm on Microsoft Word. We do not consider array accesses (which are determined using the declared types in the program) as dereference points. We omit data for points-to sets at indirect call sites, although the one level flow technique can also be applied to function pointers.

4.2 Results

In this subsection, we present results that support the following claim: The one level flow algorithm has the scaling properties of Steensgaard's algorithm, and the precision of Andersen's algorithm.

4.2.1 Performance vs Steensgaard's algorithm

In Figure 6, we chart the running times from Table 1 for Steensgaard's algorithm and the one level flow algorithm. We use the ratio of running-time (milliseconds) to program size (AST node count). The chart shows that for both algorithms, this ratio is fairly steady as program size grows, indicating that the analyses scale linearly with program size. The one level flow algorithm has two different overheads when compared with Steensgaard's algorithm: The overhead of reading and writing larger object files (fewer unifications are performed), and the overhead of the flow step. The one level flow algorithm requires roughly twice as much time as Steensgaard's algorithm over all of the test programs.

We reduced the time required for the flow step using a simple factoring technique: We identify the node with the greatest number of outgoing flow edges, via a single linear scan of the points-to graph. We then perform reachability from this node exactly once, appending the list of symbols that reach this node to the list of symbols at every node reachable from this node. This optimization eliminates repeated scans of a large portion of the points-to graph. It is possible to extend this idea to more than a single node, but we did not find that necessary.

To save space, we do not present data on space consumption. The one level flow algorithm requires more memory than Steensgaard's algorithm, because fewer locations are reclaimed by the garbage collector during the link phase, and because symbols may be associated with multiple locations after the flow step. Space consumption is still very low. For instance, the one level flow algorithm requires less than 200MB of space for Word97.

4.2.2 Precision vs Andersen's algorithm

In Figure 7, we chart the average sizes of points-to sets at dereference points, using Andersen's algorithm and the one level flow algorithm. The size of the points-to set at a dereference point is measured as the number of program symbols, including dummy symbols produced at allocation sites, in the points-to set of the dereferenced expression. Because the averages vary greatly across the test programs, we have normalized the averages against Steensgaard's algorithm to

[2] Word 97 and gcc contain roughly 1.4 MLOC and 140,000 LOC respectively, excluding comments and blank lines.

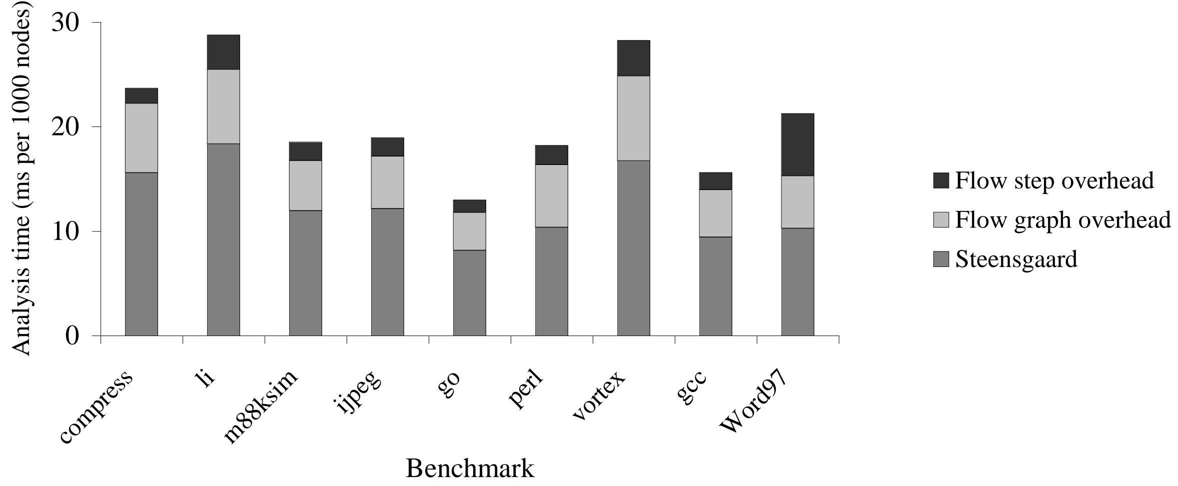


Figure 6: Scaling of the one level flow algorithm. For each benchmark, the figure shows analysis time, in milliseconds per 1000 AST nodes. The lowest band of each bar is the analysis time for Steensgaard's algorithm, while the middle band is the overhead of computing the points-to graph containing flow edges, and the upper band is the overhead of the final flow computation step.

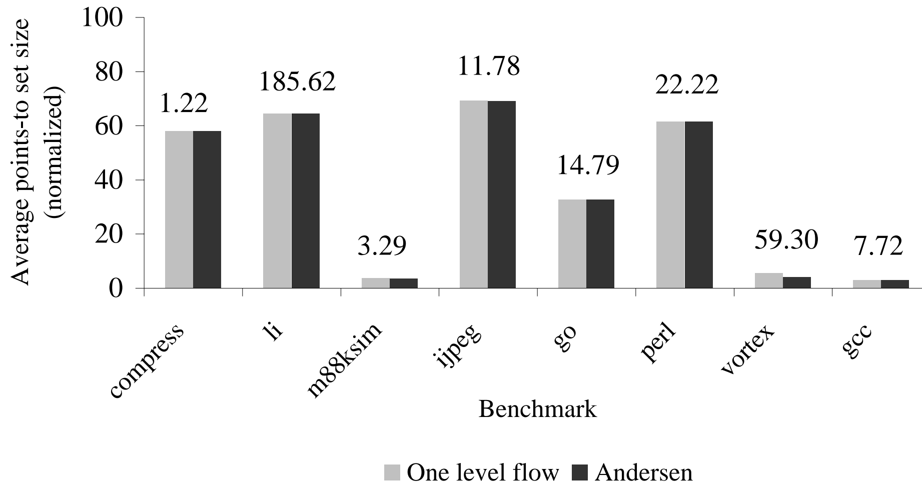


Figure 7: Precision of the one level flow algorithm, relative to Andersen's algorithm. For each benchmark, the figure shows the average size of points-to sets at static dereference points, using the one level flow algorithm and Andersen's algorithm. In order to indicate the extent to which the one level flow algorithm bridges the gap in precision between Steensgaard's algorithm and Andersen's algorithm, we have normalized the average sizes to a scale where the average using Steensgaard's algorithm is 100 for every benchmark. For reference, we have labeled the columns for the one level flow algorithm with the actual sizes from Table 1.

	0	1	2	3
And94	3	44	6	5
Flow				

compress

	0	1	2	3	4	5	6	8	9	10	11	12	15	17	30
And94	15	630	553	202	83	9	37	202	123	0	7	0	2	1	6
Flow		-7	-17	+1	-2				+6	+17		+2			

m88ksim

	0	1	2	4	6	7	117	186	201	202
And94	7	32	1	2	26	2	5	1	847	6
Flow										

li

	0	1	2	4	5	25	26	36
And94	102	921	130	1	1	8782	10	1
Flow								

perl

	0	1	2	3	5	8	16	19	24	29	73	100
And94	2	26	21	12	17	1	6	8	9	1	7	5
Flow												

go

	0	1	2	3	4	20	21
And94	2075	1211	78	9	2	4583	58
Flow			-5			+5	

ijpeg

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
And94	2958	3946	721	113	79	23	13	29	14	39332	63	54	17	16	1	54	3	21
Flow	-66	-8	+4			+1				+69			-3					

	19	20	21	23	24	25	26	29	30	31	33	35	36	38	40	47	48
And94	23	3	15	14	1	1	8	8	13	5	3	5	3	1	4	3	8
Flow			+3														

gcc

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
And94	1651	3212	1611	212	107	89	55	29	42	18	8	512	17	19	21	12	12	4	5	4	2
Flow	-1	-7	-1424	-27											-2						

	21	22	23	24	25	26	27	28	29	32	35	36	39	44	46	49	50-100	100-150	150+
And94	2	21	61	6	20	19	1	11	1	2	2	3	1	13	17	1	28	4795	104
Flow		-1			-17	-11		-11	-1								-5	+1502	+5

vortex

Figure 8: Distributions of points-to set sizes at dereferences. Each table above shows, for a given benchmark, the number of static dereference sites associated with points-to sets of given sizes by Andersen's algorithm and the one level flow algorithm. Entries for the one level flow algorithm are shown as deviations from the entries for Andersen's algorithm. Only non-zero deviations are shown. The distribution for vortex contains too many different sizes for them all to be listed individually.

	0	1	2	3	4	5	6-10	11-100	100-1000	15740-16076	16751-19094	29591
Ste96	5558	4990	1683	950	708	349	1041	736	301	0	0	177844
Flow	12670	21571	8502	2842	2234	1240	3423	4259	120	136386	11	0

Word97 – all static dereferences

	0	1	2	3	4	5	6-10	11-100	100-1000	15740-16076	16751-19094	29591
Ste96	1004	2030	844	498	362	141	423	540	38	0	0	33458
Flow	2722	7290	2745	977	790	366	1169	1638	33	21348	2	0

Word97 – updates through static dereferences

Figure 9: Distributions of points-to set sizes at dereferences in Word97. The first table above shows the number of static dereference sites in Word97 associated with points-to sets of given sizes by Steensgaard's algorithm and the one level flow algorithm. The second table includes only dereferences representing updates.

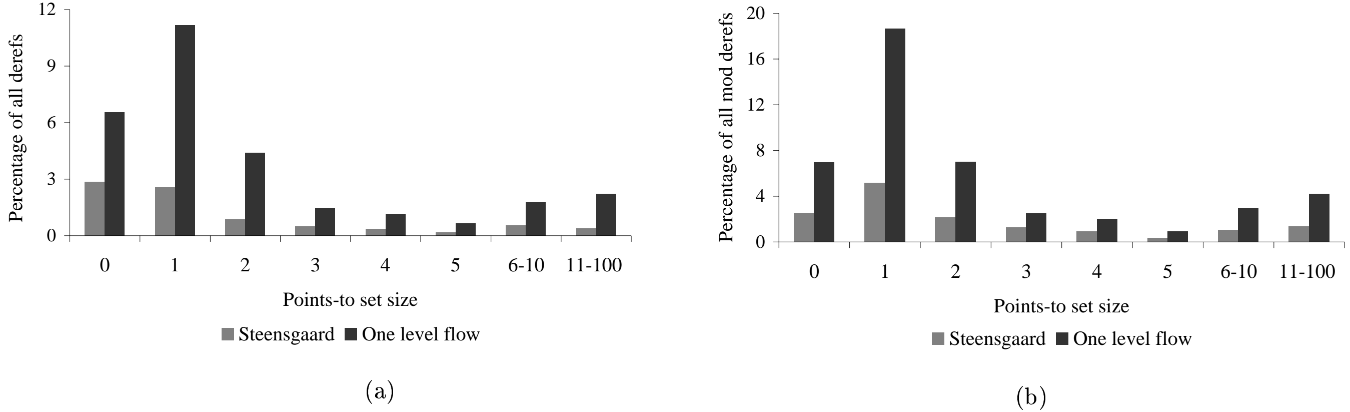


Figure 10: Distributions of dereferences with small points-to sets in Word97. The chart in (a) above shows the percentage of static dereference sites in Word97 associated with points-to sets of various small sizes by Steensgaard's algorithm and the one level flow algorithm. The chart in (b) above includes only dereferences representing updates.

represent the relative improvements obtained by the two algorithms. All three analyses were run with the same settings, including polymorphic library functions, and pending lists (not needed for Andersen's analysis). The chart shows that the precision of the one level flow algorithm is very close to that of Andersen's algorithm for all of our test programs.

Figure 8 shows the distribution of dereferences, using the two algorithms, for all of the SPECINT95 benchmark programs. For six of these programs (*compress*, *m88ksim*, *li*, *perl*, *go*, *ijpeg*), the distributions are either identical or almost identical. For *gcc*, the only difference is that Andersen's algorithm computes more empty points-to sets than our algorithm. Empty points-to sets are produced at dereference points because of two primary reasons: First, the benchmarks contain functions that are never called. Some of these functions have pointer parameters that are dereferenced. Second, we do not model constant strings.

The implementation of Andersen's algorithm in BANE is lazy: If a pointer p has no points-to set, and its points-to target $*p$ is assigned a pointer value, the assigned pointer value is not propagated to dereferences $**p$. Our algorithm sometimes propagates such values eagerly via unifications, and produces fewer empty points-to sets as a result. This effect is seen in the distribution for *gcc*.

Intuitively, there are two reasons why the results of the one level flow algorithm so closely mimic the results of Andersen's algorithm: First, our algorithm is as precise as Andersen's algorithm for top level pointers (*i.e.*, pointers that are not themselves pointed to by other pointers). Top level pointers dominate in real code, because of parameter passing. Second, even in cases where there are pointed-to pointers, these often arise because addresses of pointer-valued objects (or structures with pointer-valued fields) are passed to functions. As pointed out in Section 3.4, Andersen's algorithm must treat accesses through dereferences of these pointers bi-directionally.

The only benchmark with non-trivial differences in the distributions is *vortex*. This is because *vortex* contains several instances of pointers to pointers where the target symbols of the top level pointers are both updated and referenced directly.

4.2.3 A closer look at Word97

The distributions in Figure 8 show that the one level flow algorithm represents a method for obtaining essentially the precision of Andersen's algorithm in a manner that scales easily to large programs. However, it may be argued that for the only very large program in our test suite, the average obtained using our algorithm is too large for the analysis to be useful, even if this average is much lower than that obtained using Steensgaard's algorithm.

Figure 9 shows the distribution of dereferences for Word 97, using Steensgaard's algorithm and the one level flow algorithm. We show the distribution for all dereferences and updates through dereferences. The figure shows that our algorithm has two effects: It reduces the size of the dominant points-to set, and it shifts the distribution to include more small-sized points-to sets. The latter effect shows the value of the one level flow algorithm. The shift in distribution is shown visually in Figure 10. For instance, the distribution of updates through dereferences using our algorithm is bimodal: Over forty percent of these updates have very small points-to sets. A client static analysis can take advantage of the precise information at these updates, while making the usual conservative assumptions that are necessary when no pointer information is available at the remaining updates.

5 Related work

In recent years, there has been much work in the design of pointer analysis algorithms. Techniques described in the literature come in a variety of flavours: Flow-sensitive analyses, such as [LR92, CBC93], take into account the order in which statements are executed, while flow-insensitive analyses, such as [And94, Ste96b], assume that statements can execute in any order. Similarly, context-sensitive or polymorphic analyses [EGH94, WL95] keep different call sites to the same callee function apart in the analysis, while context-insensitive analyses do not. Each of these choices introduces a tradeoff between scaling and precision. In addition, the precision of any pointer analysis can be improved through the use of orthogonal techniques such as structure layout

mapping [Ste96a, YHR99, RFT99], or shape analysis of heap allocated data structures [GH96, SRW99].

Our goal is an algorithm that scales to large programs, while providing precision comparable to Andersen's algorithm. The previous work closest to ours is the multiple outdegree algorithm of Shapiro and Horwitz [SH97b], which attempts to bridge the gap between Steensgaard's and Andersen's analyses by k-limiting the outdegree of nodes in the points-to graph. Because the *average case* memory requirement of their algorithm is k times that of Steensgaard's algorithm, only small values of k can be used for larger programs. However, large programs require large fanout in the points-to graph, as the same utility functions are called from more call sites. To overcome this problem, Shapiro and Horwitz propose a scheme in which the k-limited algorithm is run multiple times. This scheme relies on performing intersections of points-to sets from multiple runs, which can be expensive. It is also critically dependent on the manner in which symbols are partitioned. We believe that our algorithm is more practical than their multiple run algorithm because our algorithm appears to scale better to larger programs, both in terms of time and space requirements and in terms of precision.

The program in Figure 1 (a) is an example of a common case of pointer usage involving procedure calls. An alternative method for improving precision in such cases is to use a context-sensitive version of unification-based pointer analysis. One approach, exemplified by Rehof's implementation from [FRD00], is to use polymorphic type inference to build an instantiation graph that records flow information. This approach has been found to scale to large programs in practice. Liang and Harrold have developed a second approach that operates on an explicit call graph [LH99]. However, they have only tested this approach on very small programs without indirect calls. We have found that in larger programs, which contain many indirect calls and large points-to sets for function pointers, call graphs become unmanageably large. We have found this to be true even after using the filtering techniques proposed in [AG98]. Therefore, we believe that polymorphic type inference is the best option for scalable context-sensitive pointer analysis.

The drawback of a context-sensitive extension of Steensgaard's analysis is that it introduces directionality only at the implicit assignments induced by function calls, while explicit assignments are treated bi-directionally. This may limit precision improvement. For instance, we have found that polymorphism reduces the average size of points-to sets at dereferences in *gcc* from SPEC95 to approximately 100, as compared to the average of 8 produced by the one level flow algorithm. On the other hand, there are situations where a polymorphic analysis would provide precision gains not provided by the one level flow algorithm. A natural question that arises is whether the two techniques can be combined for maximum benefit. One solution is to use the technique of polymorphic flow analysis proposed in [FRD99].

Much work has also been done from the other direction, namely on improving the scaling behaviour of Andersen's algorithm. The implementation of Andersen's analysis developed at UC Berkeley uses on-line cycle elimination [FFSA98] and projection merging [SFA00] to speed up the analysis substantially. Rountev and Chandra have used a pre-processing step to further improve performance by a factor of two [RC00]. In spite of these efforts, Andersen's algorithm does not yet scale to programs beyond 500KLOC. We believe that our use of a highly efficient unification engine at all but the critical regions of the points-to graph avoids most of the "wasted" work done by Andersen's algorithm.

We arrived at our one level flow algorithm by examining large amounts of code, and developing a succession of algorithms that each went one step further than the previous algorithm in terms of improving precision over Steensgaard's approach. The one level flow technique that resulted from this exercise is an example of subtyping restricted to one level in the type structure. This idea has been previously used in other contexts. It is called "simple closure analysis" in [Hen92] and is the basis of efficient binding-time analysis [Hen91], closure analysis and dynamic type inference algorithms. In a subtype setting, the restriction to top level is characterized by the absence of "induced" or "deep" subtyping rules. Our work shows that this restriction of subtyping has a useful application in the domain of pointer analysis.

6 Conclusions

We believe that our one level flow algorithm provides a practical method for obtaining precise flow and context insensitive points-to information on large C programs. However, this is only the first step in our long term agenda of improving the precision of scalable pointer analysis. There are several logical next steps in this direction:

- Combining the one level flow technique with polymorphic pointer analysis, using the technique of polymorphic flow analysis [FRD99]
- Combining the one level flow technique with orthogonal precision enhancements such as structure field disambiguation and call filtering
- Extending the one level flow technique by introducing approximations to flow-sensitivity such as SSA form

Our algorithm is a technique that exploits one common case of pointer usage in C programs. There are likely many remaining opportunities for improving the precision of pointer analysis.

The one level flow technique provides a simple method for improving the precision of any unification-based analysis. We have found that pointer analysis of C programs is one setting in which this approach provides substantial benefit; there are probably many others.

Acknowledgements

We would like to thank Daniel Weise, Manuel Fahndrich, Jakob Rehof, and Bjarne Steensgaard of Microsoft Research for many useful discussions on pointer analysis. Jeff Foster of UC Berkeley provided his implementation of Andersen's algorithm. Suan Yong of UWisconsin spent several months helping us improve the precision of Steensgaard's algorithm. The Semantics-Based Tools group at Microsoft Research provided the AST Toolkit. We thank the anonymous referees for their careful reviews of the original draft of this paper.

References

[AG98] D. Atkinson and W. Griswold. Effective whole-program analysis in the presence of pointers. In

Proceedings of the ACM SIGSOFT 98 Symposium on the Foundations of Software Engineering, 1998.

[And94] L. Andersen. *Program analysis and specialization for the C programming language.* PhD thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

[AST] AST Toolkit documentation. //research.microsoft.com/sbt/asttoolkit/ast.htm

[CBC93] J. D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side-effects. In *Conference Record of the Twenty-Fourth ACM Symposium on Principles of Programming Languages*, 1993.

[EGH94] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 94 Conference on Programming Language Design and Implementation*, 1994.

[FFSA98] M. Fähndrich, J. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN 98 Conference on Programming Language Design and Implementation*, 1998.

[FRD99] M. Fähndrich, J. Rehof, and M. Das. From polymorphic subtyping to CFL Reachability: Context-sensitive flow analysis using instantiation constraints. Technical Report MSR-TR-99-84, Microsoft Research, Redmond, WA, November 1999.

[FRD00] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.

[GH96] R. Ghiya and L. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Conference Record of the Twenty-Third ACM Symposium on Principles of Programming Languages*, 1996.

[Hen91] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In *Proc. Conf. on Functional Programming Languages and Computer Architecture (FPCA), Cambridge, Massachusetts*, pages 448–472. Springer, August 1991. Lecture Notes in Computer Science, Vol. 523; also DIKU Semantics Report D-88.

[Hen92] Fritz Henglein. Simple closure analysis. DIKU Semantics Report D-193, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark, March 1992.

[LH99] D. Liang and M. Harrold. Efficient points-to analysis for whole program analysis. In *Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1999.

[LR92] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN 92 Conference on Programming Language Design and Implementation*, 1992.

[RC00] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.

[RFT99] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Conference Record of the Twenty-Sixth ACM Symposium on Principles of Programming Languages*, 1999.

[SFA00] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Conference Record of the Twenty-Seventh ACM Symposium on Principles of Programming Languages*, 2000.

[SH97a] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *LNCS 1302, 4th International Symposium on Static Analysis, 1997.* Springer-Verlag, September 1997.

[SH97b] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of the Twenty-Fourth ACM Symposium on Principles of Programming Languages*, 1997.

[SRW99] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Conference Record of the Twenty-Sixth ACM Symposium on Principles of Programming Languages*, 1999.

[Ste96a] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *LNCS 1060, Proceedings of the 1996 International Conference on Compiler Construction*, pages 136–150. Springer-Verlag, 1996.

[Ste96b] B. Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the Twenty-Third ACM Symposium on Principles of Programming Languages*, 1996.

[Tar83] R. E. Tarjan. Data structures and network flow algorithms. *Regional Conference Series in Applied Mathematics*, CMBS 44, 1983.

[WL95] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 95 Conference on Programming Language Design and Implementation*, 1995.

[YHR99] S. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *Proceedings of the ACM SIGPLAN 99 Conference on Programming Language Design and Implementation*, 1999.