Demand-Driven Alias Analysis for C

Xin Zheng and Radu Rugina
Computer Science Department
Cornell University
Ithaca, NY 14853
{xinz, rugina}@cs.cornell.edu

July 2007

Abstract

This paper presents a demand-driven, flow-insensitive analysis algorithm for answering may-alias queries. We formulate the computation of alias queries as a CFL-reachability problem, and use this formulation to derive a demand-driven analysis algorithm. The analysis uses a worklist algorithm that gradually explores the program structure and stops as soon as enough evidence is gathered to answer the query. Unlike existing techniques, our approach does not require building or intersecting points-to sets.

Experiments show that our technique is effective at answering alias queries accurately and efficiently in a demand-driven fashion. For a set of alias queries from the SPEC2000 benchmarks, our analysis is able to accurately answer 97% of the queries in less than 1 millisecond per query. Compared to a demand-driven points-to analysis that constructs and intersects points-to sets on-the-fly, our alias analysis is more than two times faster.

1 Introduction

The pervasive use of pointers and references in imperative languages such as C or Java has led to a large body of research devoted to the pointer analysis problem, which aims to extract information about pointer values and aliases in programs. Such information is needed by virtually any analysis, optimization, or transformation for pointer-based programs.

Following [12], we make the distinction between *points-to analysis*, whose goal is to compute points-to relations between program variables (represented as a points-to graph), and *alias analysis*, which computes may-alias relations between program expressions. Because points-to relations compactly represent all possible aliases, they have been quickly adopted as the standard representation, and the vast majority of existing analyses (and all of the analyses developed in the past decade, to the best of our knowledge) are points-to analyses. Both exhaustive and demand-driven points-to analysis algorithms have been proposed.

When using points-to analysis to answer an alias query about two indirect memory accesses $*p$ and $*q$, the standard approach is to compute the points-to sets of p and q, and then intersect these sets. If the intersection is non-empty, then $*p$ and $*q$ may alias. However, in certain cases it is possible to answer alias queries without deriving full points-to sets. For instance, if the program assigns p to q, then the expressions $*p$ and $*q$ alias regardless of their points-to sets.

This paper presents a demand-driven, flow-insensitive analysis algorithm for C aimed at answering alias queries directly, with precision equivalent to an Andersen inclusion-based [3] points-to analysis. We formulate the alias problem as a context-free language (CFL) reachability problem over a graph representation of the assignments and pointer dereference relations in a program. In this formulation, the alias relations are described using a grammar. Non-terminals in our grammar model alias relations, not points-to relations.

We distinguish between two kinds of alias relations: *memory aliases*, representing expressions that might refer to the same memory location (this is the meaning of "may-alias" in the text above); and *value aliases*, representing expressions that might hold the same pointer value. The two relations are mutually recursive: to determine whether two expressions are memory aliases, one needs to identify whether their addresses might value-alias; and to determine whether two expressions might hold the same value, one must follow program assignments, but also consider memory aliases of the expressions being assigned. In this paper we present a context-free grammar that captures these two notions of aliases. Next, we derive a representation of the grammar using recursive state machines [2, 4], and use the state machine representation to construct a demand-driven alias analysis.

In the worst case, the proposed alias analysis algorithm might end up performing as much work as building full points-to sets for the two pointers being dereferenced. However, in many cases it can answer the queries before the two points-to sets are fully constructed. This can happen in the following two scenarios:

- If it can be seen from a small number of program assignments that the two pointers in question may hold the same value. Our approach will quickly identify such assignments and conclude that the two pointers alias, without trying to determine the possible values of these pointers. A points-to analysis will however search for all of memory locations that the two pointers may point to.

- If one of the pointers points only to a few values, none of which are pointed to by the other pointer. A points-to analysis must still compute the entire points-to set of the second pointer before concluding that the two do not alias. On the other hand, the alias analysis may discover that the few values of the first pointer cannot flow into the second, and quickly terminate the search.

To make the alias exploration more efficient, our algorithm explores deeper levels of indirection gradually, and limits the amount of exploration per query. Different exploration budgets allow users to trade precision for runtime efficiency, and vice versa.

We have implemented the alias analysis algorithm and used it to answer a set of alias queries in the SPEC2000 benchmark programs. Our experiments demonstrate that the proposed analysis can efficiently resolve a large fraction of alias queries on-demand. On average for our benchmarks, the proposed alias analysis can correctly answer (with respect to results from an exhaustive analysis) 97% of the queries in less than 1 millisecond per query, with a memory consumption of less than 50 kb. The low runtime cost and low memory demands of the analysis make it a good candidate not only for compilers, but also for interactive tools, such as program understanding or program development tools (e.g., IDEs), that have more restrictive time and space constraints.

Our experiments also show that the proposed demand-driven alias analysis is more than twice as fast as an analysis that computes points-to sets on-demand [11] and then intersects the computed sets. This shows that alias problems can indeed be solved more efficiently without constructing full points-to sets.

1.1 Contributions

This paper makes the following contributions:

- **CFL-reachability formulation of alias relations.** We present a framework for solving memory-alias and value-alias problems for C programs using CFL-reachability. To the best of our knowledge, this is the first formulation of the alias problem as a CFL-reachability problem.

- **Demand-driven alias analysis.** We present an analysis algorithm for solving alias problems in a demand-driven fashion. To the best of our knowledge, this is the first algorithm capable of answering alias queries without constructing points-to sets.

- **Experimental results.** We present an experimental evaluation showing that our method can answer alias queries accurately and efficiently.

The rest of the paper is organized as follows. We first review the related work in Section 2. Next, Section 3 presents our simplified program model. Section 4 describes the CFL-reachability formulation and presents the demand-driven analysis algorithm. A formal semantics and soundness result is given in Section 6. Finally, Section 7 presents an evaluation of the analysis and Section 8 concludes.

2 Related Work

Due to its importance, pointer analysis has been and remains a very active area of research. We limit our discussion to the most relevant pieces of related work in this area, namely alias analyses, demand-driven analyses, memory disambiguation studies, and analyses formulated as CFL-reachability problems. We refer the reader to [12] for a survey of pointer analysis techniques.

Alias analyses. Early approaches to pointer analysis have been formulated as alias analyses [16, 5]. These are inter-procedural dataflow analyses that compute a set of alias pairs at each program point. However, alias pairs are difficult to maintain and often contain redundant information. More compact representations of alias pairs have been explored [5], but eventually points-to graphs and points-to sets have emerged as the natural way of representing pointer information. The analyses developed in the past decade have all used this representation. Our analysis differs from existing alias analyses in that it is flow-insensitive and demand-driven. Both of these aspects are key to our approach: being demand-driven, we avoid the quadratic blowup of computing all alias pairs; and being flow-insensitive, we avoid computing alias pairs at each point. In the demand-driven setting, computing aliases directly is a better choice because points-to sets contain more information than needed to answer alias queries.

Demand-driven pointer analyses. Heintze and Tardieu [11] present a demand-driven Andersen-style points-to analysis for C programs, and use it to disambiguate indirect function calls. To answer a points-to query, their algorithm recursively generates point-to and pointed-by queries until the original points-to query is fully resolved. In contrast, our framework answers alias queries and does not necessarily need to build complete points-to sets.

In recent work, Sridharan et al. [24, 23] have proposed demand-driven points-to analyses for Java programs. Their algorithm extends a previous CFL-reachability formulation of the points-to analysis problem [20] to Java. A key technique in their approach is matching loads and stores on the same field via `match` edges. However, this technique is not applicable to C programs because it would amount to considering any two pointer dereferences `*p` and `*q` as aliased, which would be too conservative. They use a notion of refinement-based analysis, where `match` edges are gradually refined until the answer to the query meets the needs of the client.

Memory disambiguation. Ghyia, Lavery, and Sehr [10] present a study of memory disambiguation techniques for C programs to resolve alias queries issued by several compiler optimizations and transformations. Among other techniques, they use an Andersen inclusion-based pointer analysis for answering alias queries. Lattner et al. [17] add context sensitivity to unification-based [25] points-to analysis and use it for memory disambiguation. Their analysis uses a context-sensitive heap abstraction to distinguish between heap objects allocated at the same site, but through different calls. We use a simple malloc wrapper detector to achieve a similar effect. In another study, Das et al. [6] use a context-sensitive one-level flow analysis to resolve alias queries in several C programs. Alias pairs were generated by computing all possible expression pairs in each function. Their study indicates that tracking value flows in a context-sensitive fashion brings very little improvements to alias disambiguation in C programs. All of these analyses answer alias queries by intersecting points-to sets.

CFL-reachability formulations. CFL-reachability has become a popular technique for expressing program analysis problems [20]. Standard problems that have been expressed in this framework include points-to analysis problems [20, 24], context-sensitive inter-procedural dataflow analysis problems [19], and context-sensitive, but flow-insensitive pointer analyses [9, 6, 23].

In the CFL-reachability formulation of points-to analyses [20], the grammar expresses points-to relations. In contrast, our formulation addresses the aliasing problem and non-terminals in our grammar denote alias relations. The fact that we use a single relation for assignments A, instead of four different kinds of assignment edges (address-of, copy, load, and store) also makes the grammar simpler and more general.

3 Program Representation

We formulate our analysis for C-like programs that manipulate pointers. For simplicity, we shall assume that all program values are pointers. Programs consist of sets of pointer assignments. The control-flow between assignments is irrelevant since the analysis is flow-insensitive. Assignments can execute in any order, any number of times. Program expressions and assignments are represented in a canonical form with the following minimal syntax:

$$\begin{array}{llllll} \text{Addresses} & a & \in & \mathsf{Addr} & & \\ \text{Expressions:} & e & \in & \mathsf{Expr}, & e & ::= \;\; a \;\mid\; *\,e \\ \text{Assignments:} & s & \in & \mathsf{Stat}, & s & ::= \;\; *\,e_1 := e_2 \end{array}$$

Primitive addresses $a \in \mathsf{Addr}$ model symbolic addresses of variables (e.g., `&x`) and dynamic allocation sites (e.g., `malloc()`). There is one address $a_{\texttt{malloc}}$ per allocation site, and one symbolic address a_x for each variable x. There are two kinds of expressions: primitive addresses a, and pointer dereferences $*\,e$. A pointer dereference expression $*\,e$ denotes the value of the memory location that pointer expression e points to. If `x` is a variable, then the C expression `&x` is represented in canonical form as a_x; expression `x` is represented as $*\,a_x$, and expression `*x` is represented as $*\,*\,a_x$.

In canonical form, an expression represents a memory location if and only if it is a dereference expression $*\,e$. In the rest of the paper we will also refer to such expressions as *lvalues* [13]. Program assignments are of the form $*\,e_1 := e_2$, syntactically enforcing that the left-hand side be an lvalue. Expression e_1 is the address of the memory location being updated, and e_2 is the value being written into that location. Assignments include (but are not limited to) the four standard forms of pointer assignments used in the pointer analysis literature: address-of assignments `x = &y`, represented as $*\,a_x := a_y$; copy assignments `x = y`, as $*\,a_x := *\,a_y$; loads `x = *y`, as $*\,a_x := *\,*\,a_y$; and stores `*x = y`, as $*\,*\,a_x := *\,a_y$. Our formulation does not require the program to be normalized to this form; it automatically handles more complex assignments (for instance, `*x = **y`) without introducing temporary variables.

The data structure that our algorithm operates on is the *Program Expression Graph* (PEG). This is a graph representation of all expressions and assignments in the program. The nodes of the graph represent program expressions, and edges are of two kinds:

- *Pointer dereference edges* (D): for each dereference $*\,e$, there is a D-edge from e to $*\,e$.
- *Assignment edges* (A): for each assignment $*\,e_1 := e_2$, there is an A-edge from e_2 to $*\,e_1$.

For each A and D edge, there is also a corresponding edge in the opposite direction, denoted by $\overline{A}$ and $\overline{D}$, respectively. In the rest of the paper we will also refer to graph edges as relations between the corresponding nodes. Hence, relations $\overline{A}$ and $\overline{D}$ are the inverse relations of A and D, respectively. All of the above relations (or edges) are pre-computed before the analysis.

In the remainder of the paper we will use the terms "node" and "expression" interchangeably, because of the one-to-one mapping between nodes and expressions in the PEG.

Example. Figure 1 shows an example graph. The source C program consists of the pointer assignments shown on the left (control structures are omitted because they are irrelevant to a flow-insensitive analysis). The program expression graph is shown on the right. The nodes in the graph represent all program expressions and subexpressions. For readability, node expressions are shown in C form, not in canonical form. However, the graph is constructed using their canonical representation. Solid edges represent pointer assignments, and dotted edges represent pointer dereferences. Each horizontal line corresponds to a level of pointer indirection. Expressions lower in the figure are at deeper pointer levels.

4 Alias Analysis via CFL-Reachability

The goal of the alias analysis is to compute may-alias relations between program expressions. We define two kinds of aliases:

Program:

```
s = &t;
r = &z;
y = &r;
s = r;
x = *y;
```

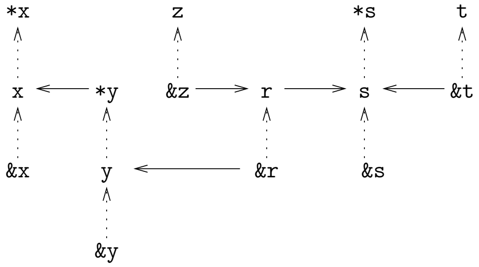


Figure 1: Program Expression Graph example. Solid, horizontal edges represent assignments (A-edges) and dotted, vertical edges represent pointer dereferences (D-edges).

$$\begin{array}{llcl}
\text{Memory aliases:} & M & ::= & \overline{D} \quad V \quad D \\[1em]
\text{Value aliases:} & V & ::= & \overline{F} \quad M? \quad F \\[1em]
\text{Flows of values:} & F & ::= & (A \quad M?)^* \\
 & \overline{F} & ::= & (M? \quad \overline{A})^*
\end{array}$$

Figure 2: Context-free grammar G for the may-alias problem. The grammar uses EBNF notation, where "?" indicates an optional term, and "*" is the Kleene star operator. Symbols D and A are terminals: D denotes dereference edges, and A denotes assignment edges.

- *Memory aliases*: two lvalue expressions are memory aliases if they might refer to the same memory location;
- *Value aliases*: two expressions are value aliases if they might hold the same pointer value.

We describe memory aliases using a binary relation $M \subseteq \mathsf{Expr} \times \mathsf{Expr}$, and value aliases using a binary relation $V \subseteq \mathsf{Expr} \times \mathsf{Expr}$. Each relation can be viewed as an edge in the program expression graph. We formulate the computation of M and V edges as a context-free grammar (CFL) reachability problem [20, 14] over the program expression graph.

The idea of CFL-reachability is as follows. Given a graph with labeled edges, a relation R over the nodes of this graph can be formulated as a CFL-reachability problem by constructing a grammar G such that nodes n and n' are in the relation R if and only if there is a path from n to n' such that the sequence of labels on the edges belongs to the language $L(G)$ defined by G. Such a formulation makes it easier to develop demand-driven algorithms in a deductive fashion.

Figure 2 shows the context-free grammar G for aliasing problems. The grammar is written using EBNF notation, where the star symbol is the Kleene star operator, and the question mark indicates an optional term. Each terminal and non-terminal represents a relation. The concatenation of terminals and non-terminals in the right-hand side of a production corresponds to relation composition.

The grammar G from Figure 2 has three non-terminals, M, V, and F; and two terminals, A and D. Terminals A and D represent assignments and dereference edges in the expression graph. Non-terminal M models memory aliasing relations, and non-terminal V represents value aliasing relations. Finally, non-terminal F describes flows of values via assignments and memory aliases. More precisely, an F edge from e to e' indicates that the execution of the program might write the value of expression e into the memory location of expression e'.

The intuition behind each production is as follows:

- Production $M ::= \overline{D} \;\; V \;\; D$ shows that two memory locations $* e_1$ and $* e_2$ are memory aliases, i.e., $M(* e_1, * e_2)$, when their addresses have the same value: $V(e_1, e_2)$. Hence, the path from $* e_1$ to $* e_1$ consists of an anti-dereference edge $\overline{D}(* e_1, e_1)$, a value alias edge $V(e_1, e_2)$, and a dereference edge $D(e_2, * e_2)$.
- Production $V ::= \overline{F} \;\; M? \; F$ shows that two expressions e_1 and e_2 are value aliases if there exist two expressions e_1' and e_2' that are memory aliases, $M(e_1', e_2')$, and whose values flow into e_1 and e_2, respectively: $F(e_1', e_1)$ and $F(e_2', e_2)$. In this production M is optional because M is not reflexive for primitive addresses $a \in \mathsf{Addr}$ (see the properties in the next subsection).
- Production $F ::= (A \; M?)^*$ means that flows of values are due to sequences of assignments and memory aliases. The production $\overline{F} ::= (M? \;\; \overline{A})^*$ describes the inverse relation, i.e., value flows in the opposite direction.

The value-flow relation F has been introduced in the grammar to make it easier to understand. However, non-terminal F can be eliminated from the productions, as follows:

$$M \;\; ::= \;\; \overline{D} \quad V \quad D \tag{1}$$

$$V \;\; ::= \;\; (M? \;\; \overline{A})^* \quad M? \quad (A \;\; M?)^* \tag{2}$$

Hence, memory aliases and value aliases are mutually recursive. Computing memory aliases requires computing value aliases for their addresses; and computing value aliases involves knowledge about memory aliases during value flows. At each step, the recursive process goes one pointer level deeper.

Example. Consider the example from Figure 1. Suppose the analysis wants to determine whether *x and *s are memory aliases. Expressions &r and y are value aliases $V(\texttt{\&r}, \texttt{y})$ because the assignment $A(\texttt{\&r}, \texttt{y})$ causes a value flow from &r to y. Therefore, the dereferences of these expressions are memory aliases: $M(\texttt{r}, \texttt{*y})$. Furthermore, the value of r flows into s, and the value of *y flows into x. Since r and *y are memory aliases, we conclude that x and s are value aliases: $V(\texttt{x}, \texttt{s})$. Therefore, *x and *s are memory aliases: $M(*\texttt{x}, *\texttt{s})$.

From the CFL-reachability perspective, the path from *x and *s that traverses nodes [*x, x, *y, y, &r, r, s, *s] corresponds to a string $\overline{D}\,\overline{A}\,\overline{D}\,\overline{A}\,D\,A\,D$. Since this string is in the language of M in the alias grammar, the two expressions *x and *y may alias.

4.1 Properties

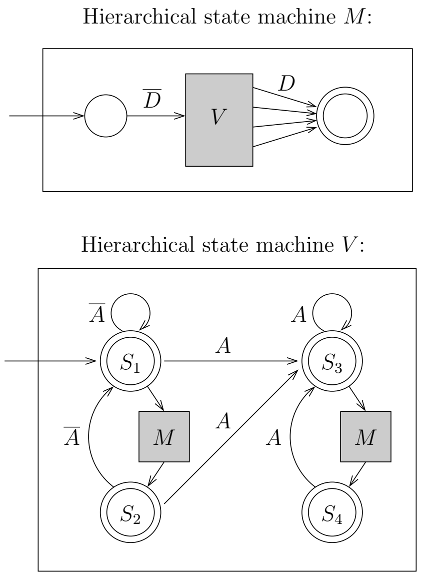
To better understand relations D, V and M, it is useful to identify their key properties. The list below enumerates and discusses these properties:

- *The dereference relation D is an injective partial map.* In other words, each program expression e has at most one dereference expression e' such that $D(e, e')$; and at most one "address-of" expression e'' such that $\overline{D}(e, e'')$. The relation $D \; \overline{D}$ (meaning the composition of D and $\overline{D}$) is the identity for primitive address expressions; and $\overline{D} \; D$ is the identity for lvalue expressions.
- *Relations V and M are symmetric.* One can show that a relation is symmetric by showing that it is equal to its inverse. In the CFL model, the inverse of a nonterminal is obtained by reversing the order of terms in the right-hand side of its production, and then inverting them. The inverses of M and V are:

  $$\overline{M} \;=\; \overline{D}\,\overline{V}\,\overline{\overline{D}} \;=\; \overline{D}\,\overline{V}\,D$$

  $$\overline{V} \;=\; \overline{F}\,\overline{M}?\,\overline{\overline{F}} \;=\; \overline{F}\,\overline{M}?\,F$$

  The above equalities can then be used to inductively show that $\overline{V} = V$ and $\overline{M} = M$, by induction on the path length of relations M, $\overline{M}$, V and $\overline{V}$ (where relations are regarded as paths, according to the CFL-reachability model).



Hierarchical state machine M:

Hierarchical state machine V:

Figure 3: The Hierarchical State Machines that describes aliasing relations: machine M recognizes memory aliases, and machine V recognizes value aliases. The two machines are mutually recursive.

- *Relation V is reflexive.* This is shown by the fact that non-terminal V is nullable (it derives the empty string ε in the grammar).

- *Relation M is reflexive only for lvalue expressions.* This is because V is nullable, so $\overline{D}\ D$ is a subset of the relations in M, and because $\overline{D}\ D$ is the identity for lvalue expression. Relation M is not reflexive for primitive addresses because such expressions don't have incoming D edges by construction.

- *Relations M and V are not necessarily transitive.* Hence, they are not equivalence relations and cannot be implemented using union-find structures.

  We illustrate this property in the example from Figure 1. Expressions `r` and `s` are value aliases, and so are `s` and `&t`; however, `r` and `&t` are not. Similarly, `*x` and `*s` are memory aliases, and so are `*s` and `t`, but `*x` and `t` refer to disjoint pieces of memory.

  In particular, the non-transitivity of M is the reason why the production for value flows F applies M at most once after each assignment. Applying M more than once might generate spurious alias relations.

4.2 Hierarchical State Machine Representation

This section shows a representation of grammar G using hierarchical state machines [2, 1, 4]. The alias analysis algorithm will be constructed from this state machine model. A hierarchical state machine is an automaton whose nodes are either states or boxes. Each box represents a "call" to another state machine, and has a set of inputs and outputs; these correspond to the start and final states of the machine being called. Transitions within each machine link states, box inputs, and box outputs. As the name implies, recursive state machines also allow recursive calls between the machines.

Figure 3 shows the hierarchical, recursive state machines for the aliasing problem. These machines are constructed from productions (1) and (2), where the F non-terminal has been eliminated. The machine for memory aliases M is shown in the upper part of the figure, and the one for value aliases V in the lower part. Each machine has one start state, indicated by the edge that crosses the box. Furthermore, machine M has one output final state, and all of the four states of machine V are final. In M's automaton, all four edges emanating from the outputs of the V box are labeled D.

The four states of the automaton V describe the current position in production (2). States 1 and 2 correspond to incoming value flows in the first portion of the production. The automaton stays in these states while traversing inverted assignment edges $\overline{A}$. Once a forward assignment A is traversed, the execution moves to states 3 and 4. These states correspond to the second portion of the production. From this point on, only A edges can be traversed. Hence, the execution can be thought of consisting of two stages, one represented by states 1 and 2, and the other represented by states 3 and 4. Each of the stages needs two states to ensure that M is never called twice in a row. After each invocation of M, the automaton follows either an A edge or an $\overline{A}$ edge, depending on the current stage.

4.3 Alias Analysis Algorithm

We now present the demand-driven algorithm for answering memory may-alias queries. Given two lvalue expressions e_1 and e_2, the algorithm determines whether $M(e_1, e_2)$ holds.

Figure 4 shows the analysis algorithm. This is a worklist algorithm that propagates CFL-reachability information through the program expression graph. The propagation of reachability facts follows the structure of the recursive state machines from Figure 3. During propagation, the algorithm uses machine call summaries to cache and reuse results of recursive calls to state machines. This mechanism is similar to the way inter-procedural dataflow analyses cache and reuse function call summaries [22, 19].

The A and D edges of the expression graph are modeled using four functions: *deref*(n) is the dereference of node n, or *null* if none exists; *addr*(n) is the address of node n, or *null* if none exists; *assignTo*(n) is the set of nodes that n is assigned into; and *assignFrom*(n) is the set of nodes that n is assigned from. Hence, functions *deref*, *addr*, *assignTo*, *assignFrom* indicate the D, $\overline{D}$, A, and $\overline{A}$ edges, respectively.

The information being propagated by the algorithm is value aliases V. The worklist elements are triples of the form $\langle n, s, c\rangle$ indicating that a value-alias reachability propagation initiated at node s (the source node) has reached node n (the current node), in automaton state c. Here, state c is one of the four states of the state machine V from Figure 3. Hence, the presence of element $\langle n, s, c\rangle$ in the worklist implies that n and s are value aliases. For each node n, the algorithm maintains a set *reach*(n) of pairs $\langle s, c\rangle$ of source nodes and machine states that have reached node n. The algorithm also maintains a set *aliasMem*(n) that indicates the currently known memory aliases of node n; these act as summaries of calls to state machine M. Both of the sets *aliasMem*(n) and *reach*(n) grow during the execution of the algorithm, but need not be fully computed before the query is answered.

The functioning of the algorithm proceeds as follows. Given the lvalue expressions e_1 and e_2, the algorithm tries to determine whether address expressions *addr*(e_1) and *addr*(e_2) are value aliases. For this, it starts the value-alias reachability automaton in state S_1, from node *addr*(e_1), as shown by the initialization at line 2. If, during the execution of the main loop, the propagation of this information reaches node *addr*(e_2), then the query returns "true", as shown in lines 10-11. Otherwise, at each iteration the algorithm performs the following tasks:

- Lines 14-19: information is propagated "up" in the graph, through dereference edges. In this part, the algorithm identifies new summaries of calls to the M machine. If the current node n has a dereference $n' \neq$ *null*, and that dereference is not in the memory alias set of s' (the source's dereference), then a new summary $M(s', n')$ is detected. In this case, the algorithm adds n' to the alias set of s', and then propagates each pair $\langle s'', c''\rangle$ that has reached s' over to n' in order to simulate the call to M. Such calls are possible only for pairs in states S_1 or S_3; the resulting states are S_2 and S_4, respectively. This models the transitions for M calls in the V machine.

- Lines 22-45: information is propagated through value flows. This part of the algorithm precisely models all of the transitions in the V automaton. For transitions that correspond to M calls, the algorithm uses the current memory-alias summaries. For the other transitions, it propagates information through the assignments or inverted assignments in the expression graph, changing the automaton state accordingly.

```
MayAlias($e_1$ : Expr, $e_2$ : Expr)

w ← { ⟨ addr($e_1$), addr($e_1$), $S_1$⟩ }
while (w is not empty)
    remove ⟨n, s, c⟩ from w
    $s' \leftarrow deref(s)$
    $n' \leftarrow deref(n)$

    /* check if the destination has been reached */
    if ($s' = e_1 \wedge n' = e_2$)
       then return true

    /* propagate information upward */
    if ( $n' \neq null \wedge n' \notin aliasMem(s')$)
       then $aliasMem(s') = aliasMem(s') \cup \{n'\}$
            for each ⟨$s''$, $c''$⟩ in $reach(s')$ :
                switch ($c''$)
                  case $S_1$ : PROPAGATE($w$, $n'$, $s''$, $S_2$)
                  case $S_3$ : PROPAGATE($w$, $n'$, $s''$, $S_4$)

    /* propagate reachability through value flows */
    switch (c)
       case $S_1$ :
            for each m in assignFrom(n) :
                PROPAGATE($w$, $m$, $s$, $S_1$)
            for each m in aliasMem(n) :
                PROPAGATE($w$, $m$, $s$, $S_2$)
            for each m in assignTo(n) :
                PROPAGATE($w$, $m$, $s$, $S_3$)
       case $S_2$ :
            for each m in assignFrom(n) :
                PROPAGATE($w$, $m$, $s$, $S_1$)
            for each m in assignTo(n) :
                PROPAGATE($w$, $m$, $s$, $S_3$)
       case $S_3$ :
            for each m in assignTo(n) :
                PROPAGATE($w$, $m$, $s$, $S_3$)
            for each m in aliasMem(n) :
                PROPAGATE($w$, $m$, $s$, $S_4$)
       case $S_4$ :
            for each m in assignTo(n) :
                PROPAGATE($w$, $m$, $s$, $S_3$)

    /* propagate information downward */
    if ($addr(n) \neq null \wedge (c = S_1 \vee c = S_3)$)
       then PROPAGATE($w$, $addr(n)$, $addr(n)$, $S_1$)

return false

PROPAGATE($w$, $n$, $s$, $c$)
if ( $\langle s, c\rangle \notin reach(n)$ )
   then $reach(n) \leftarrow reach(n) \cup \{ \langle s, c\rangle \}$
        $w \leftarrow w \cup \{ \langle n, s, c\rangle \}$
```

Figure 4: Demand-Driven Alias Analysis Algorithm.

- Lines 48-49: information is propagated "down" in the graph, through inverted dereference edges. This propagation corresponds to calling automaton M, with the purpose of discovering new memory aliases of the current node n. Such calls are only possible if the automaton is currently in state S_1 or S_3, and if the current node n has an address (i.e., an inverted dereference edge $\overline{D}$). In these cases, the algorithm starts a new value-alias propagation at the address of n. As this new propagation proceeds, it will enable the algorithm to discover new memory aliases at lines 18 and 19.

The execution of the algorithm can have two outcomes. If the algorithm identifies that $addr(e_1)$ and $addr(e_2)$ are value aliases, then the worklist algorithm terminates early and reports that e_1 and e_2 are memory aliases, at line 11. Otherwise, when the worklist becomes empty, all of the value-flow paths from $addr(e_1)$ have been explored, and none of them has reached $addr(e_2)$. In this case, the algorithm reports that the two expressions are not aliased, as shown at line 44.

4.4 Analysis Example

We illustrate the functioning of the alias analysis algorithm using the example from Figure 1. Suppose we want to answer the alias query *mayAlias*(*x, *s). Figure 5 illustrates the propagation of worklist items for this query starting from expression *x. The figure shows only the relevant portion of the graph. The shaded circles shown above the nodes represent worklist items; the numbers inside items show the item state; and the dashed edges show the propagation of items through the graph.

The propagation is also shown in textual form in the table below. The table shows each step of the algorithm, the full contents of each worklist item, and the new items being generated at each step. The last column indicates the kind of propagation for each of the generated items.

Step	Current item	Added to worklist	Propagation kind
1	(x, x, S_1)	(x, *y, S_1)	flow
2	(x, *y, S_1)	(y, y, S_1)	down
3	(y, y, S_1)	(y, &r,S_1)	flow
4	(y, &r,S_1)	(x, r, S_2)	up
5	(x, r, S_2)	(x, s, S_3)	flow
6	(x, s, S_3)	return "may alias"	

The analysis starts the reachability propagation from the address of expression *x, in state S_1 of the automaton, as indicated by the first item (x, x, S_1). The analysis traverses the assignment edge $*$y → x in the opposite direction, maintaining the automaton state S_1. The new triple becomes (x, *y, S_1).

At this point there are no other incoming or outgoing assignment edges for *y, so the analysis starts looking for aliases of *y. This is done by propagating information "down" in the graph and starting a new reachability propagation at the address of *y. The starting item is (y, y, S_1). After traversing the assignment &r → y (again in the opposite direction) the analysis generates the triple (y, &r, S_1). This shows that y and &r are value aliases. Hence, the analysis concludes that their dereferences are memory aliases. This is done by propagating the information "up" in the graph: for each item that has reached *y, the analysis moves it over to r, adjusting the automaton state to indicate that a memory alias has been detected. In this example, the item (x, *y, S_1) is propagated over to (x, r, S_2). The automaton state changes to S_2 to show that the propagation has traversed a memory alias.

Finally, the analysis traverses the assignment edge r → s. Since this edge is now traversed in the forward direction, the automaton moves to state S_3, and the new item becomes (x, s, S_3). This shows that x and s are value aliases. Their dereferences, *x and *s, are the expressions in the query. Hence, the analysis reports that the two expressions may alias.

Note that the analysis would have given the same answer if the propagation of information had started from the other end, i.e., expression *s. In that case, the same path would have been traversed, but in the opposite direction. This behavior is in agreement with the fact that the alias relation M is reflexive.

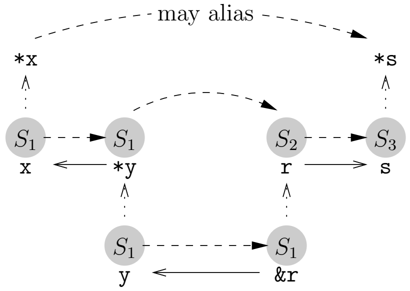


Figure 5: Functioning of the worklist algorithm for the query *mayAlias*(`*x`, `*s`). The shaded circles represent worklist items, and the dashed lines indicate the propagation of items. The automaton state is shown in each of the items.

4.5 Analysis Enhancements

We propose improving the basic alias reachability algorithm in several ways.

Gradual exploration. First, we impose an order on the exploration at different pointer levels, giving priority to reachability propagation at upper levels of the expression graph. In this way, the algorithm first explores the possibility of value aliases through assignments only. If the query is still not answered, then the algorithm searches for memory aliases and starts using them in the value flows. Furthermore, at each pointer level, the algorithm performs a breadth-first search to avoid exploring long assignment chains when short paths exist. This exploration behavior is achieved by implementing the worklist as a multi-level queue. Each insertion specifies the level where the worklist element is added. Each removal retrieves the first element in the topmost (non-empty) queue.

Concurrent exploration. The second improvement follows from the observation that memory alias relations are symmetric, but the alias algorithm is asymmetric, since it starts propagation only from the first expression. Changing the order of expressions in the query will not affect the final answer, but might impact efficiency. To address this issue, we propose an enhancement where the algorithm starts propagation from both ends, e_1 and e_2.

Conceptually, the algorithm consists of two separate, concurrent searches. This lends itself to parallelism and can take advantage of a multi-core or multi-processing system. However, our experiments with a truly concurrent implementation showed that the synchronization and thread management overheads outweigh the concurrency gains.

Thus, our algorithm uses a single-threaded implementation where the two searches use two separate worklists, and tasks from the two worklists are manually interleaved. Additionally, the search at the topmost pointer level stops as soon as the propagations from the two opposite ends connect to each other. We refer to this approach as the *two-worklist algorithm.*

Tunable exploration budget per query. To limit the time spent on alias queries, our analysis uses a parameter N that controls the amount of exploration per query. This represents the maximum number of steps (i.e., worklist iterations) that each query is allowed to run. In our experiments, N ranged between 6 and 2000 steps. For the two-worklist algorithm, the budget is equally divided between the two worklists. If the analysis does not terminate within this budget, it stops and conservatively reports that the expressions may be aliased. The exploration budget N provides a convenient way for the analysis clients to trade running time for precision, and vice-versa.

Caching. The presentation so far assumed that each query is executed from scratch. A way to improve the analysis is to cache results such that successive queries benefit from the efforts of earlier ones. In this way, the analysis will do less work, at the expense of using more memory by accumulating results. New queries will be able to reuse the memory alias information $aliasMem(n)$ stored at each node n, without exploring the deeper pointer levels.

However, one complication arises due to the following fact. When a query terminates the loop early, at line 11, it has not finished its exploration; in particular, it has not finished computing full memory alias sets for the dereference nodes traversed. A subsequent query reaching such a dereference node will not know whether the alias set is fully constructed, and will need to conservatively start a new reachability path at the child node, thus duplicating previous work. Our solution is to add a flag to each node indicating whether its alias set is fully constructed. Each query keeps track of the dereference nodes it traverses. If the query ends up with an empty worklist, returning at line 44, then it marks all of the traversed dereference nodes, indicating that it has fully built their alias sets.

4.6 Comparison to Points-to Analysis

It is useful to compare our alias analysis to points-to analyses. The key difference between the two analyses is that the alias analysis computes the alias relations V and M, whereas points-to analyses compute a points-to relation $P \subseteq \mathsf{Expr} \times \mathsf{Addr}$ between program expressions and memory addresses. Relation P can be derived from V, as follows. If R is a relation that marks all of the primitive address expressions, $R = \{\ (a, a) \mid a \in \mathsf{Addr}\}$, then:

$$P \;=\; \overline{F}\;\, R \;=\; (M?\;\, \overline{A})^*\;\; R \tag{3}$$

The above equation indicates that an address is in the points-to set of an expression if it flows into the expression. Using this equation, one could build a demand-driven points-to analysis that answers queries of the form $pointsTo(e)$. We briefly sketch such an analysis, but omit the full algorithm because it is not the focus of this paper. The algorithm would be similar to the alias analysis in Figure 4, with the following exceptions. The initialization of the worklist would mark the source with a special symbol START. Then, propagations through assignments A (at lines 29, 35, 39, and 45) will be disallowed when the source is START to model just the backward flows. The termination condition at lines 10-11 would be replaced by code that adds the current node into the points-to set being queried, provided that the source is START and the current node is an address. The final points-to set will then be returned after the loop. Essentially, this would be the demand-driven points-to analysis of Heintze and Tardieu [11]. A key observation here is that the alias analysis presented in Section 4.3 might terminate early, at line 11, whereas the points-to analysis cannot, because it must construct the full set.

Equation (3) also indicates that the computation of points-to sets requires information about memory aliases. As we know, the memory alias relation M and value alias relation V are mutually recursive. Hence, points-to analyses, either exhaustive or demand-driven, must compute relations M and V, or some approximation thereof. Conservative approximations can be obtained via transitive closures (recall that M and V are symmetric, but not transitive). This leads to the following classification of three important points-to analyses by the kinds of alias relation approximations they use:

- Andersen's inclusion algorithm [3]: M and V are not transitive. This is the case for the relations computed by the algorithm in this paper, as defined by equations (1) and (2).
- Steensgaard's unification algorithm [25]: both M and V are approximated by their transitive closures.
- Das's one-level flow algorithm [7]: approximates M by its transitive closure, but V remains non-transitive.

The unification-based analyses are efficient both in time and in space. These analyses run in almost linear time in the program size, and they efficiently represent computed relations using union-find data structures. Steensgaard's analysis provides an approximation of both M and V, whereas the unification phase of Das's algorithm provides an approximation of M only (computing V or P still requires performing value flows).

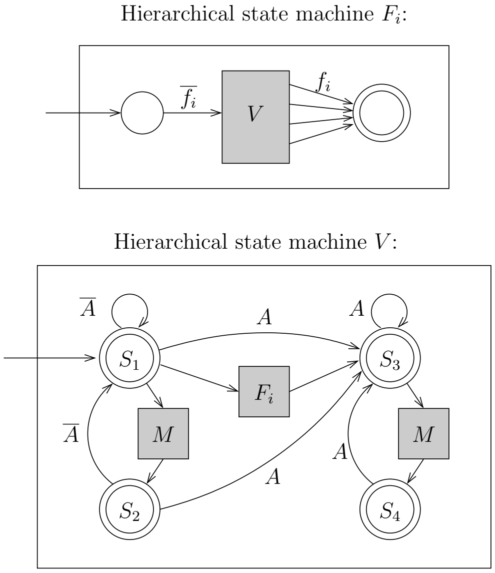


Figure 6: Hierarchical state machines for field-sensitive grammar G_f. State machine M is the same as in Figure 3.

5 Field-Sensitive Analysis

The CFL formulation of alias analysis from Section 4 can be extended to distinguish between different structure fields. We extend the syntax for expression to model fields:

$$e ::= a \mid *e \mid e + f$$

If e is a pointer expression and f a structure field, then expression $e + f$ denote the address of field f in the structure pointed to by e. Standard C expressions such as `x.f`, `&(x->f)`, or `x->f` are represented as $*(a_x + f)$, $*a_x + f$, and $*(*a_x + f)$, respectively. In the presence of structure fields, the PEG contains field edges between nodes, in addition to assignment and dereference edges. In the PEG, there is an f edge from e to $e + f$, and an inverse edge $\overline{f}$ in the opposite direction.

We now augment the grammar for alias relations to deal with the presence of structure fields. The grammar G_f for field-sensitive alias analysis is as follows:

$$\begin{array}{lrcl}
\text{Memory aliases:} & M & ::= & \overline{D} \;\; V \;\; D \\
\text{Value aliases:} & V & ::= & \overline{F} \;\; V \;\; F \;\mid\; \overline{f_i} \;\; V \;\; f_i \;\mid\; M? \\
\text{Flows of values:} & F & ::= & (A \;\; M?)^* \\
 & \overline{F} & ::= & (M? \;\; \overline{A})^*
\end{array}$$

The above grammar reflects the fact that two field expressions are value aliases if and only if they refer to the same field and their base expressions are value aliases. This is shown in the production $V ::= \overline{f_i}\ V\ f_i$, where f_i is a structure field. There is one such production for each field f_i in the program. The language of grammar G_f subsumes the language of the field-insensitive grammar G.

The hierarchical state machines for grammar G_f are shown in Figure 6. State machine M is the same as before and is omitted. Additionally, there is one state machine F_i for each field f_i. Machine F_i is aimed at matching field address expressions with the same field name. Note that transitions into the F_i machine occur only in the first stage (states S_1 or S_2); and transitions out of F_i lead to the second stage (state S_3).

```
FieldSensitiveMayAlias($e_1$ : Expr, $e_2$ : Expr)

$w \leftarrow \{ \langle addr(e_1),\ addr(e_1),\ S_1,\ `*` \rangle \}$

while ($w$ is not empty)
    Remove $\langle n, s, c, l \rangle$ from $w$
    $s' \leftarrow upNode(s, l)$
    $n' \leftarrow upNode(n, l)$

    /* check if the destination has been reached */
    if ( $s' = e_1 \ \wedge\ n' = e_2 \ \wedge\ isDeref(l)$ )
       then return true

    /* propagate information up through a dereference */
    if ( $isDeref(l) \ \wedge\ n' \neq null \ \wedge\ n' \notin aliasMem(s')$ )
       then $aliasMem(s') = aliasMem(s') \ \cup\ \{n'\}$
            for each $\langle s'',\ c'',\ l'' \rangle$ in $reach(s')$ :
                switch ($c''$)
                  case $S_1$ : PROPAGATE($w,\ n',\ s'',\ S_2,\ l''$)
                  case $S_3$ : PROPAGATE($w,\ n',\ s'',\ S_4,\ l''$)

    /* propagate information up through a field */
    if ( $isField(l) \ \wedge\ n' \neq null \ \wedge\ n' \notin aliasField(s')$ )
       then $aliasField(s') = aliasField(s') \ \cup\ \{n'\}$
            for each $\langle s'',\ c'',\ l'' \rangle$ in $reach(s')$ :
                if ($c'' = S_1$)
                  then PROPAGATE($w,\ n',\ s'',\ S_3,\ l''$)

    /* propagate reachability through value flows */
    switch ($c$)
       case $S_1$ :
            for each $m$ in $aliasMem(n)$ :
                PROPAGATE($w,\ m,\ s,\ S_2,\ l$)
            for each $m$ in $aliasField(n)$ :
                PROPAGATE($w,\ m,\ s,\ S_3,\ l$)
            for each $m$ in $assignTo(n)$ :
                PROPAGATE($w,\ m,\ s,\ S_3,\ l$)
       case $S_2$ :
            for each $m$ in $assignFrom(n)$ :
                PROPAGATE($w,\ m,\ s,\ S_1,\ l$)
            for each $m$ in $assignTo(n)$ :
                PROPAGATE($w,\ m,\ s,\ S_3,\ l$)
       case $S_3$ :
            for each $m$ in $aliasMem(n)$ :
                PROPAGATE($w,\ m,\ s,\ S_4,\ l$)
       case $S_4$ :
            for each $m$ in $assignTo(n)$ :
                PROPAGATE($w,\ m,\ s,\ S_3,\ l$)

    /* propagate information downward */
    if ( $downNode(n) \neq null \ \wedge\ (c = S_1 \ \vee\ c = S_3)$ )
       then $label \leftarrow downLabel(n)$
            $node \leftarrow downNode(n)$
            PROPAGATE($w,\ node,\ node,\ S_1,\ label$)

return false
```

```
PROPAGATE($w,\ n,\ s,\ c,\ l$)
if ( $\langle s,\ c,\ l \rangle \notin reach(n)$ )
   then $reach(n) \leftarrow reach(n) \ \cup\ \{ \langle s,\ c,\ l \rangle \}$
        $w \leftarrow w \ \cup\ \{ \langle n,\ s,\ c,\ l \rangle \}$
```

Figure 7: Demand-Driven Field-Sensitive Alias Analysis.

The reason is that field address expressions cannot be assigned into. A field expression can only be the last in a sequence of inverse assignments $\overline{A}$, or the first in a sequence of direct assignments A.

Figure 7 shows the demand-driven field-sensitive alias analysis algorithm. The algorithm augments the algorithm from Figure 4 with support for fields. The key point of the algorithm is that it uses summaries for the execution of each state machine F_i, which matches f_i edges. These summaries are similar to those for machine M, which matches dereference edges. In general, the field-sensitive analysis tends to treat field edges and dereference edges uniformly.

For each node n, the analysis maintains a set of aliases describing state machine summaries. If n is an lvalue expression, then $aliasMem(n)$ is a set of memory aliases, as before. If n is a field address expression, then $aliasField(n)$ is a set of value aliases for that field. These sets represent the summaries of machines M and F_i, respectively.

The algorithm augments the worklist elements with one additional item: l, representing a label. A label is either a field name f_i, or the special label "*" that describes dereferences. The label indicates whether the propagation has been started at a dereference, or at a field expression. It is used by the analysis to match a corresponding dereference or field with the same name. This is done at lines 6 and 7, where the function $upNode(n, l)$ looks for an edge that emanates from n and matches label l. If such an edge is found, $upNode(n, l)$ returns its target; otherwise, it returns $null$. Predicates $isDeref(l)$ and $isField(l)$ indicate whether label l is the dereference symbol, or a field symbol.

Returning from state machine calls is performed between lines 14-19 for dereference expressions, and between lines 22-26 for field expressions. The former corresponds to the state transitions for calls to the M machine, whereas the latter models state transitions for calls to F_i. The value flows are augmented with the loops at lines 33-34, where the analysis reuses existing summaries of calls to F_i.

Finally, at lines 50-52, the analysis calls either the M machine, or the F_i machine. This is done by starting a fresh new reachability query from the expression that represents the child of n. This expression is computed using $downNode(n)$, defined below. The function $downLabel(n)$ returns the label of the corresponding edge. These functions are formally defined as follows:

$$\begin{array}{lclclcl}
downNode(a) & = & null & \quad & downLabel(a) & = & null \\
downNode(*\,e) & = & e & & downLabel(*\,e) & = & '*' \\
downNode(\texttt{\#}f\,e) & = & e & & downLabel(\texttt{\#}f\,e) & = & f
\end{array}$$

This completes the presentation of the field-sensitive alias analysis algorithm.

6 Semantics and Soundness

We briefly state our soundness result as follows. Consider the following semantic domains:

$$\begin{array}{rcl}
\sigma \in \mathsf{Store} & = & \mathsf{Addr} \to \mathsf{Value} \\
\nu \in \mathsf{Value} & = & \mathsf{Addr} \cup (\mathsf{Field} \to \mathsf{Value}) \cup \{\bot\}
\end{array}$$

where $a \in \mathsf{Addr}$ ranges over addresses, $f \in \mathsf{Field}$ over structure fields, and $\bot$ represents uninitialized values. The denotational semantics of statements and expressions is:

$$\begin{array}{rcl}
[\![a]\!]\,\sigma & = & a \\
[\![*\,e]\!]\,\sigma & = & \sigma([\![e]\!]\,\sigma) \\
[\![e + f]\!]\,\sigma & = & [\![e]\!]\,\sigma\,f \\
[\![*\,e_1 := e_2]\!]\,\sigma & = & \sigma[[\![e_1]\!]\sigma \mapsto [\![e_2]\!]\sigma]
\end{array}$$

A program is a set of statements $\mathsf{prog} = \{s_1, ..., s_n\}$. The execution of the program consists of executing statements s_i any number of times, in any order, starting from an initial store σ_0 where all memory locations are uninitialized: $\sigma_0 = \lambda a.\bot$.

Definition 1 *We say that relations M, V, and F are sound approximations of store σ, written $(M, V, F) \approx \sigma$, if:*

$$\begin{array}{rclcl} \forall e, e' & : & [\![e]\!]\sigma = [\![e']\!]\sigma \neq \bot & \Rightarrow & V(e, e') \\ \forall e, e' & : & [\![e]\!]\sigma = [\![e']\!]\sigma \neq \bot & \Rightarrow & M(*e, *e') \\ \forall e & : & [\![e]\!]\sigma = a \in \mathsf{Addr} & \Rightarrow & F(a, e) \end{array}$$

Theorem 1 *The relations V, M, and F defined using CFL-reachability over the program expression graph are sound approximations of all stores that might arise during the execution of the program.*

Proof. We will show that $(M, V, F) \approx \sigma$ for all σ that can arise in a program by strong induction on program execution.

Base case. Trivially true for σ_0.

Inductive case. Let σ be the store after execution of statement s. Write s as $*e_1 := e_2$. Inductively assume that

$$(V, M, F) \approx \sigma' \text{ for all } \sigma' \text{ before execution of } s.$$

Consider an arbitrary expression e such that $[\![e]\!]\sigma = a$. We will show that $F(a, e)$ holds.

If $[\![e]\!]\sigma' = a$ for any σ' that existed before the execution of s, then $F(a, e)$ already holds by the inductive hypothesis.

Otherwise, we must have $[\![e]\!]\sigma \neq [\![e]\!]\sigma'$ where σ' is the store right before executing s. That is, statement s modified the value of expression e, and as such, s must have written into at least one memory location represented by a subexpression of e.

Write e as $*\cdots*a_e$, and let e' be the smallest subexpression of e whose value changed as a result of executing s. Statement s must have updated the memory location where e' resides.

Write e' as $*e''$. Since s explicitly updated $*e_1$, we must have $[\![e'']\!]\sigma' = [\![e_1]\!]\sigma'$. This implies $V(e'', e_1)$ and $M(e', *e_1)$ by our inductive hypothesis.

Let expression e' be updated to have value a' by statement s. We must have $[\![e_2]\!]\sigma' = a'$ and thus $F(a', e_2)$ holds. Statement s itself implies $A(e_2, *e_1)$.

Altogether, the facts $F(a', e_2)$, $A(e_2, *e_1)$, and $M(*e_1, e)$ imply $F(a', e')$.

- Case 1: If $e' = e$, we would have $a = a'$ and $F(a, e)$, and we are done.
- Case 2: If e' is some strict subexpression of e, then we have the following.

  Let $[\![*e']\!]\sigma = a''$, and let s' be the statement that updated the memory location where $*e'$ resides by writing a'' into it. Note that s' can be any statement executed before s or even possibly s itself. (Our inductive assumption will still hold even if $s' = s$, since we will only need information about the store *before* the execution of s'.)

  Write s' as $*e'_1 := e'_2$, and let σ'' be the store before the execution of s'. We thus have $[\![e'_1]\!]\sigma'' = a'$ and $[\![e'_2]\!]\sigma'' = a''$. By our induction hypothesis, we get $F(a', e'_1)$ and $F(a'', e'_2)$.

  The facts $F(a', e')$ and $F(a', e'_1)$ imply $V(e', e'_1)$ and $M(*e', *e'_1)$. Statement s' gives us $A(e'_2, *e'_1)$.

  We have thus inferred $F(a'', e'_2)$, $A(e'_2, *e'_1)$, and $M(*e'_1, *e')$. These facts imply $F(a'', *e')$.

  Repeat this process to show value flows into $**e', \ldots, *\cdots*e' = e$, and we have $F(a, e)$ as desired.

Now consider two arbitrary expressions e and e' such that $[\![e]\!]\sigma = [\![e']\!]\sigma = a$. We have shown that $F(a, e)$ and $F(a, e')$ hold, which implies $V(e, e')$ and $M(*e, *e')$. This completes the proof.

Program	Code size	PEG size		Alias
	(KLOC)	Nodes	Edges	queries
164.gzip	7.8	4767	3226	34
175.vpr	17.0	11242	9833	91
176.gcc	205.7	112341	168484	1086
177.mesa	50.2	51766	271863	955
179.art	1.3	1226	659	4
181.mcf	1.9	1303	1040	4
183.equake	1.5	1716	967	54
186.crafty	19.5	10929	7238	17
188.ammp	13.3	13526	9203	59
197.parser	10.9	9538	8753	99
253.perlbmk	61.8	48703	52964	304
254.gap	59.5	58915	809665	656
255.vortex	52.6	50322	65125	784
256.bzip2	4.6	3523	1681	22
300.twolf	19.7	14057	9977	120

Figure 8: Benchmark programs.

7 Evaluation

This section presents an evaluation of the algorithms proposed in this paper, as well as the enhancements discussed in Section 4.5. These algorithms were implemented in Crystal [21], a program analysis infrastructure for C written in Java. The alias analysis is implemented in Java and is publicly available as part of the latest release of the Crystal infrastructure.

We ran our experiments on the SPEC2000 suite of C benchmark programs. Figure 8 lists these benchmarks and their sizes, both in terms of number of lines of source code, and in terms of the size of their Program Expression Graphs. The experiments were conducted on a dual-processor 3.8 GHz Pentium 4 machine with 2 GB of memory, running Windows XP.

7.1 Program Representation and C Language Features

The Crystal program analysis infrastructure uses an intermediate representation of expressions and statements similar to the canonical form presented in Section 3. The program expression graph (PEG) is constructed by scanning all of the statements in the Crystal representation, and this takes on average 0.5 seconds for our benchmarks. Our front-end handles all of the C language, translating all expressions and assignments into their canonical forms.

In addition to primitive addresses and dereference expressions, the intermediate representation contains two other kinds of canonical expressions: field expressions and arithmetic expressions. Array expressions are automatically translated into pointer arithmetic and dereference expressions. In the PEG, each pointer arithmetic expression $e + i$ is mapped to the same node as its base expression e, so the analysis does not distinguish between the two. This also implies that the analysis does not distinguish between different array elements.

In the PEG, each field expression $e + f$ is mapped to the same node as its base expression e, meaning that the analysis is field-insensitive. We have also experimented with a field-sensitive formulation of our analysis (presented in Section 5), but found that the benefits of field sensitivity are extremely small (less than 1%) to justify the additional complexity in the analysis. Therefore, all of the results in this section use a field-insensitive analysis.

Finally, function calls are handled in a context-insensitive manner, as assignments from actual arguments to parameters, and from the returned expression to the expression being assigned at the call. Allocation wrappers are automatically detected by our system using a simple intra-procedural flow-insensitive analysis. The analysis identifies as allocation wrappers those functions whose return value exclusively come from allocation points (either `malloc` or other allocation wrappers) via assignments to local variables whose addresses have not been taken. The alias analysis treats calls to allocation wrappers as distinct allocation sites.

7.2 Evaluation Methodology

We ran our analysis against a static set of queries using different budgets N to judge its precision. To generate a more realistic set of queries, we performed a standard available expressions analysis, a dataflow analysis commonly used for partial redundancy elimination. From this set we excluded those queries that can be trivially answered without pointer analysis, or that require techniques beyond pointer analysis. These include the following:

- Pairs of memory locations corresponding to memory blocks with different names, e.g., different variables or different allocation sites. In canonical form, these are dereference expressions whose base expressions are distinct primitive addresses. Such expressions cannot alias.
- Pairs where one of the expressions is a variable whose address has not been taken. In canonical form, the primitive addresses of these variables have no outgoing assignment edges. Such variables cannot alias any other expression.
- Pairs referring to different elements of the same array, e.g., `a[i]` and `a[j]`. These expressions map to the same PEG node. Disambiguating such expressions would require techniques beyond pointer analysis, for instance, array dependence analysis [18]. We conservatively assume that such expressions may alias, and we exclude them from our statistics.
- Pairs of expressions that access different fields of a structure, for instance `a.f` and `b.g`. Regardless of whether expressions `a` and `b` are the same, the two field accesses never alias. These pairs are also excluded.

The last column in Figure 8 shows the number of remaining queries from each benchmark.

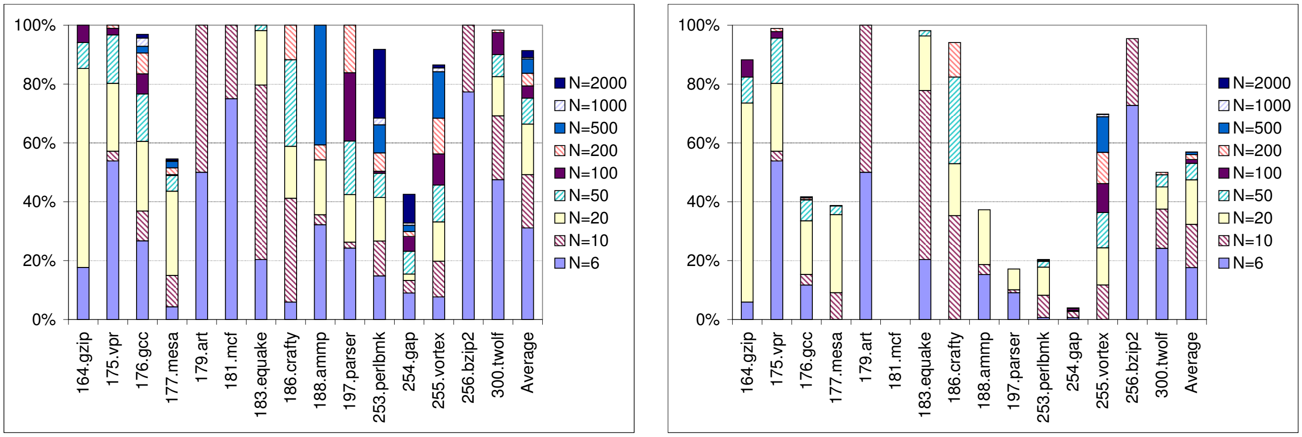
We evaluate and compare the following three algorithms for answering alias queries:

1. **Demand-Driven Alias Analysis (DDA)**. This is the analysis algorithm proposed in this paper.
2. **Exhaustive Points-to Analysis (EXH)**. This is an exhaustive Andersen-style points-to analysis [15]. The analysis pre-computes points-to sets for all variables in the program, and then answers alias queries by intersecting their points-to sets. We use the results of this analysis as an upper bound for determining the accuracy of our analysis.
3. **Demand-Driven Points-to Analysis (DDPT)**. This is the demand-driven points-to analysis discussed in Section 4.6. For each alias query, the analysis computes points-to sets on-demand and then intersects the computed sets. We compare both the precision and the efficiency of this approach against ours.

Section 7.3 presents an evaluation of DDA alone. Next, Sections 7.4 and 7.5 compare DDA against EXH and DDPT, respectively.

7.3 Evaluating DDA Alone

We evaluate the demand-driven alias analysis algorithm presented in Section 4. The implementation of the analysis includes all of the extensions presented in Section 4.5: the gradual exploration of deeper pointer



(a)

(b)

Figure 9: (a) Percentage of queries completed by the demand-driven alias analysis (DDA) for exploration budgets N between 6 and 2000 steps. (b) Percentage of same queries answered as unaliased.

levels, the two-worklist algorithm, and the tunable analysis budgets. The exploration budget parameter N ranges from 6 iterations to 2000 iterations per query, and all budgets are divided equally between the two worklists of the algorithm. We evaluate both a version that caches query results (DDA-C) and one without caching (DDA-NC) that discards all of the results after each query.

Running time. Figure 10 shows the time to answer an alias query on-demand, averaged over all queries from all benchmarks, for different values of the analysis budget N. Both of the axes (runtime and analysis budget) are logarithmic. The upper line in the figure corresponds to the non-caching version DDA-NC, and the lower line to the caching version DDA-C.

The results indicate that both versions take less than 1 millisecond per query for all budgets less than 500. For lower budgets, the analysis takes as little as 0.2 milliseconds per query. The numbers also indicate that caching can improve the runtime per query, especially for larger budgets. For those, the cached version runs more than twice as fast. For budgets in the middle of the range (e.g., $N = 100$), caching makes the analysis run 1.5 times faster.

Precision. Figure 9 shows an evaluation of the precision of the demand-driven alias analysis. The figure shows the percentage of queries completed and the percentage of queries answered as unaliased for different values of the exploration budget N.

The numbers in Figure 9(a) show that the analysis completes and thus fully resolves (either with a positive or with a negative answer) about 90% of the queries on average using a budget of 2000 iterations. Almost 50% of the queries can be answered in 10 iterations or less. The figure shows that for increasing exploration sizes, the analysis yields diminishing returns. In particular, there is little benefit in running the analysis with budgets larger than 200 iterations. For that budget, the analysis resolves more than 80% of the queries. The numbers of Figure 9(b) show that the number of queries answered as unaliased follow the same trends, and represent slightly more than half of the total number of queries.

As we will show in Section 7.4, most of the unfinished queries correspond to expressions that may alias according to the exhaustive analysis. Since we conservatively answer that expressions may-alias when a query exceeds its budget, it means that most of our conservative answers are in fact correct. This situation occurs in applications such as *mesa* and *gap* that use complex, custom memory management that leads to long, eventually unsuccessful searches. Setting a low exploration budget thus becomes useful for avoiding such expensive queries in these cases, while providing the correct answer in most cases.

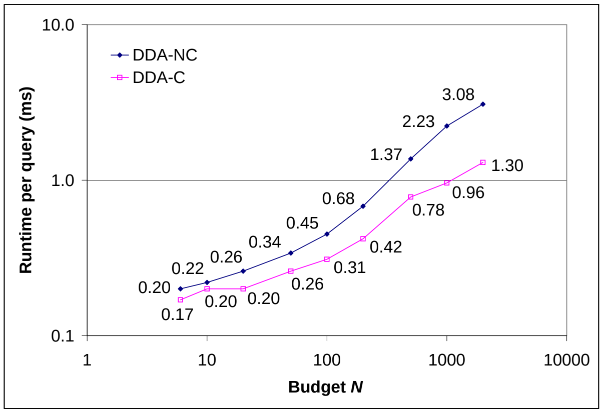


Figure 10: Average runtime per query (in milliseconds) for the demand-driven alias analysis (DDA) under various budgets. DDA-NC is the version with no caching, and DDA-C is the version with caching. Both axes are logarithmic.

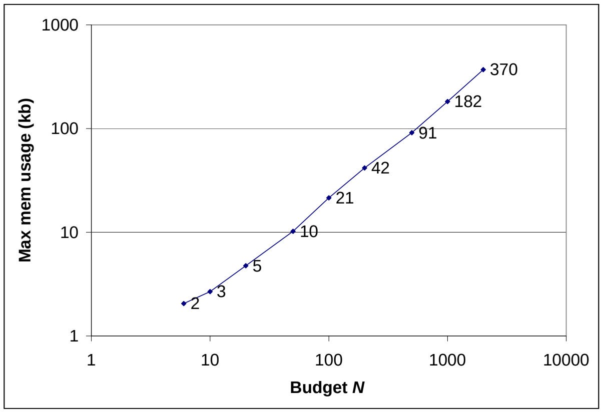


Figure 11: Maximum memory consumption (in kilobytes) of DDA-NC across all queries in all benchmarks for different budgets. Both axes are logarithmic.

Memory consumption. Since our implementation is in Java and this is a garbage collected language, it is difficult to monitor precise memory usage. As such, we measure memory demand by identifying the total number of objects created during each query and estimating their memory consumption per object. The latter is done by separately creating a large number of them, inserting them into the same Java collection structures as the algorithm (e.g., HashSets), and then deriving the average memory consumption per object.

Figure 11 shows the estimated memory consumption numbers for the analysis version without caching (DDA-NC). These numbers confirm that our demand-driven analysis has very low memory requirements. For a budget of 100, only 21 kb is needed. Even for the largest budget, the analysis never uses more than 400 kb.

In addition, for the caching version (DDA-C) we have also estimated the amount of memory being used for the cache. This cache size increases both with the budget N and with the number of queries. The maximum cache size observed in our experiments was 2 Mb for *gap* with a budget of $N = 2000$. For all other benchmarks and all budgets, the cache never exceed 1 Mb. For a budget of 100, the cache size for all benchmarks was less than 200 kb. Note that it would be possible to implement a more fine-grained cache eviction policy to ensure control on the cache memory size, thus balancing memory usage and performance.

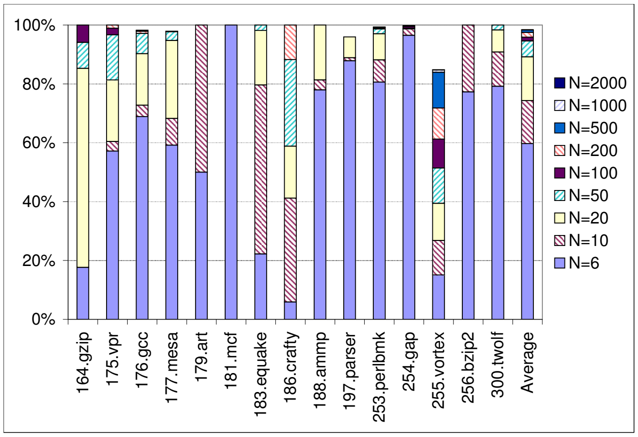


Figure 12: Percentage of queries where our demand-driven alias analysis (DDA) gave the same answer as the exhaustive analysis (EXH).

Budget	Runtime (ms)	
N	DDA	DDPT
6	0.20	0.20
10	0.22	0.23
20	0.26	0.30
50	0.34	0.53
100	0.45	0.83
200	0.68	1.44
500	1.37	2.78
1000	2.23	4.26
2000	3.08	9.76

Figure 13: Average runtime per query (in milliseconds) for the demand-driven alias analysis (DDA) and the demand-driven points-to analysis (DDPT).

7.4 Comparison to Exhaustive Points-to Analysis (EXH)

We compare our analysis to Banshee [15], an exhaustive state-of-the-art Andersen-style points-to analysis that pre-computes points-to sets for all variables in the program. Alias queries are then answered by intersecting these sets. We refer to this analysis as EXH. Since the precision of EXH is an upper bound for the precision of our analysis, the purpose of comparing the two analyses is to determine how close our analysis gets to the ideal answers.

Figure 12 shows the result of this comparison. The bars in this figure show the percentage of queries where DDA gave the same results as EXH. The results indicate that our analysis is highly accurate, even for low analysis budgets. For an analysis budget of $N = 200$ the analysis correctly answers more than 97% of the queries. Even with as little as 50 steps, DDA disambiguates over 90% of the queries that EXH does. Comparing Figures 12 and 9(a), we conclude that the majority of queries where our demand-driven analysis does not finish are cases where expressions may alias. As mentioned earlier, our conservative answer becomes the correct answer in these cases.

We have also computed and compared the memory consumption of the exhaustive analysis. The computed memory size excludes the memory needed for parsing the programs (estimated by a run where points-to analysis was disabled). On average, EXH requires 35 Mb per benchmark, and up to 150 Mb for *gcc*. We see

Budget	Completed		Unaliased	
N	DDA	DDPT	DDA	DDPT
6	31%	14%	18%	14%
10	49%	31%	32%	30%
20	66%	40%	48%	38%
50	75%	51%	53%	49%
100	79%	53%	54%	50%
200	84%	54%	56%	51%
500	89%	55%	57%	51%
1000	89%	55%	57%	51%
2000	91%	55%	57%	52%

Figure 14: Accuracy comparison between the demand-driven alias (DDA) and demand-driven points-to (DDPT): percentage of queries completed and percentage found unaliased for different budgets.

that an exhaustive analysis can be very expensive in terms of the memory needed, especially when compared with the memory demands of DDA.

Although comparing running times is less meaningful, as the exhaustive analysis pre-computes all of the results and amortizes the cost over all queries, we briefly report these numbers for the sake of completeness. On average, EXH takes 0.95 seconds to pre-compute the points-to sets. The runtime per query is 1 ms, but most of this time is spend building points-to sets for complex expressions (such as `**p`) from the points-to sets of individual variables. The actual set intersection operation is a small fraction of this time.

7.5 Comparison to Demand-Driven Points-To Analysis (DDPT)

Finally, we compare DDA against our own implementation of demand-driven points-to analysis (DDPT), as described in Section 4.6. This points-to analysis is similar to the one proposed by Heintze and Tardieu [11]. Although an implementation in C + SML of that analysis is publicly available, we chose to use our Java implementation for two reasons: 1) to ensure that DDA and DDPT are implemented in the same language and same framework; and 2) to provide tunable exploration budgets for DDPT. These aspects make the comparison more meaningful.

Figure 13 shows the runtimes of both the demand-driven alias analysis and the points-to analysis under various budgets. The version with no caching is used for both analyses. Similar differences between the two are seen with caching.

The alias analysis is consistently faster than the points-to analysis, and for budgets beyond 100, the difference is about a factor of 2. We observe that when both analyses are able to resolve a query under the same budget, the points-to analysis takes on average twice as many exploration steps, and this is reflected in the runtimes.

Figure 14 compares the precision of the demand-driven alias analysis against the demand-driven points-to analysis given the same exploration budgets. We see that the alias analysis can complete and answer more queries than the points-to analysis for all budgets. Beyond a budget of 200, the points-to analysis is unable to complete many more queries even with an exponential increase in the budget. The data is even more striking when one considers the differences in budget needed to achieve the same results. DDA can complete more queries in 20 steps than DDPT can in 2000 steps. The number of queries that is decided as unaliased is about the same when DDA is given a budget of 50 and when DDPT is given one of 2000.

These results show that the demand-driven alias analysis is more precise (for the same budget) and significantly more efficient (for the same accuracy) in answering alias queries compared to the demand-driven points-to analysis.

7.6 Summary of Findings

We summarize the main conclusions of our experiments as follows:

- A demand-driven alias analysis can effectively resolve memory-alias queries with the same accuracy as an exhaustive analysis in very little time and using no pre-computed information.
- The memory consumption of our demand-driven analysis is extremely low and orders of magnitude less than that of a state-of-the-art exhaustive analysis.
- For answering may-alias queries, a demand-driven alias analysis will do less work than a demand-driven points-to analysis and can answer more queries in less time given the same budget.

8 Conclusions

We have presented a novel demand-driven algorithm that answers may-alias queries. The analysis is designed to answer the alias queries without attempting to compute or intersect points-to sets. We have described the alias relation using a context-free grammar, and then formulated the alias problem as a CFL-reachability problem over a graph representation of the program. Our results show that the technique is very effective at accurately resolving alias queries on-demand. The analysis can accurately answer a very large fraction of the queries with small time and space consumption, making the approach attractive not only for compilers, but also for more constrained environments such as program development and interactive tools.

A possible direction of future work will concern investigating the applicability of existing optimizations proposed for inclusion-based points-to analysis, such as on-line cycle elimination [8], to the alias analysis problem. Such optimizations have proved to be very effective for exhaustive points-to analyses, but it is unclear if the same applies to alias analyses or to demand-driven analyses. We plan to investigate this problem in the future.

9 Acknowledgements

The authors would like to thank Manu Sridharan for insightful comments on earlier drafts of this paper. We also thank Jeff Foster for useful discussions about this analysis. This work was supported in part by National Science Foundation grants CCF-0541217 and CNS-0406345.

References

[1] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Proceedings of the 13th International Conference on Computer Aided Verification*, Paris, France, July 2001.

[2] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *Proceedings of the 6th ACM SIGSOFT international Symposium on the Foundations of Software Engineering*, Lake Buena Vista, FL, November 1998.

[3] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994.

[4] M. Benedikt, P. Godefroid, and T. Reps. Model checking of unrestricted hierarchical state machines. In *Proceedings of the Twenty-Eighth Interantional Colloquium on Automata, Languages, and Programming*, Crete, Greece, July 2001.

[5] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual Symposium on Principles of Programming Languages*, Charleston, SC, January 1993.

[6] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Proceedings of the International Static Analysis Symposium*, Paris, France, July 2001.

[7] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, Vancouver, Canada, June 2000.

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

[9] M. Fahndrich, 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*, Vancouver, Canada, June 2000.

[10] R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, Snowbird, UT, June 2001.

[11] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, Snowbird, UT, June 2001.

[12] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the SIGPLAN-SIGSOFT '01 Workshop on Program Analysis for Software Tools and Engineering*, Snowbird, UT, June 2001.

[13] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.

[14] J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, Washington, DC, June 2004.

[15] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *In Proceedings of the International Static Analysis Symposium*, London, UK, September 2005.

[16] W. Landi and B. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992.

[17] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitve points-to analysis with heap cloning practical for the real world. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2007.

[18] D. Maydan, J. Hennessy, and M. Lam. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, June 1991.

[19] T. Reps, S. Horowitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd Annual ACM Symposium on the Principles of Programming Languages*, San Francisco, CA, January 1995.

[20] Thomas Reps. Program analysis via graph reachability. In *Proceedings of the International Logic Programming Symposium*, Port Jefferson, NY, October 1997.

[21] Radu Rugina, Maksim Orlovich, and Xin Zheng. Crystal: A program analysis system for C. URL: http://www.cs.cornell.edu/projects/crystal.

[22] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice Hall Inc, 1981.

[23] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, Ottawa, Canada, June 2006.

[24] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, San Diego, CA, October 2005.

[25] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg Beach, FL, January 1996.