Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis

Guoqing Xu[1], Atanas Rountev[1], and Manu Sridharan[2]

[1] Ohio State University, Columbus, OH, USA
[2] IBM T.J. Watson Research Center, Hawthorne, NY, USA

Abstract. Pointer analyses derived from a Context-Free-Language (CFL) reachability formulation achieve very high precision, but they do not scale well to compute the points-to solution for an entire large program. Our goal is to increase significantly the scalability of the currently most precise points-to analysis for Java. This CFL-reachability analysis depends on determining whether two program variables may be aliases. We propose an efficient but less precise pre-analysis that computes context-sensitive *must-not-alias* information for all pairs of variables. Later, these results can be used to quickly filter out infeasible CFL-paths during the more precise points-to analysis. Several novel techniques are employed to achieve precision and efficiency, including a new approximate CFL-reachability formulation of alias analysis, as well as a carefully-chosen trade-off in context sensitivity. The approach effectively reduces the search space of the points-to analysis: the modified points-to analysis is more than three times faster than the original analysis.

1 Introduction

Pointer analysis is used pervasively in static analysis tools. There are dozens (or maybe even hundreds) of analyses and transformations that need information about pointer values and the corresponding memory locations. Many of these tools — e.g., software verifiers [1,2], data race detectors [3,4], and static slicers [5] — require both precision and scalability from the underlying pointer analysis. The quality of the results generated by such tools is highly sensitive to the precision of the pointer information. On the other hand, it is highly desirable for the pointer analysis to scale to large programs and to quickly provide points-to/aliasing relationships for a large number of variables. To date, existing pointer analysis algorithms have to sacrifice one of these two factors for the sake of the other, depending on the kind of client analysis they target.

Of existing pointer analysis algorithms, the family of the refinement-based analyses [6,7] derived from the Context-Free-Language (CFL) reachability formulation [8] are some of the most precise ones. They achieve precision by simultaneously approximating CFL-reachability on two axes: method calls and heap accesses. Method calls are handled context sensitively: a method's entry and exit

S. Drossopoulou (Ed.): ECOOP 2009, LNCS 5653, pp. 98–122, 2009.

are treated as balanced parentheses and are matched in order to avoid propagation along unrealizable paths (with the appropriate approximations needed to handle recursive calls). Heap accesses are handled to precisely capture the flow of pointer values through the heap: the read (load) and write (store) of a field of the same object are treated as balanced parentheses. These analyses answer particular points-to/alias queries raised from a client analysis, starting with an approximate solution and refining it until the desired precision is achieved.

Refinement-based pointer analyses may not scale well if a client analysis requires highly-refined information for a large number of variables [9]. For example, the Sridharan-Bodik analysis from [6], when using its default configuration, spent more than 1000 seconds on computing the whole-program points-to solution for a simple Java program and the large number of library classes transitively used by it. This solution was required by our static slicer to compute a program slice for a particular slicing criterion. It is important to note that the slicer requires points-to information not only for variables in the application code (i.e., the program we wrote), but also for variables in all reachable library methods; this is needed in order to compute appropriate dependence summary edges [5] at call sites. Less-refined (i.e., more approximate) points-to information, even though it can be produced quite efficiently, could introduce much more imprecision in the generated slice. For example, the generated slice contained the entire program (i.e., it was very imprecise) if we imposed a 500-second time constraint on the pointer analysis. In fact, this scalability problem prevents many similar *whole-program* analyses from obtaining highly-precise points-to information with acceptable running time. The goal of the analysis proposed in this paper is to help the analysis from [6] to generate highly-refined points-to information in a more efficient way.

Insight. The work performed by the Sridharan-Bodik analysis can be coarsely decomposed into core work that is performed to find the true points-to relationships, and auxiliary work performed to filter out infeasible points-to relationships. As analysis precision increases, *so does the ratio of auxiliary work to core work*. In fact, the increase of the amount of auxiliary work is usually much more noticeable than the expected precision improvement, which is detrimental to algorithm scalability. In order to obtain high precision while maintaining scalability, staged analysis algorithms [2,10] have been proposed. A staged analysis consists of several independent analyses. A precise but expensive analysis that occurs at a later stage takes advantage of the results of an earlier inexpensive but relatively imprecise analysis. This can reduce significantly the amount of auxiliary work that dominates the running time of the precise analysis. Our technique is inspired by this idea. We propose an analysis which efficiently pre-computes relatively imprecise results. Later, these results can be used to quickly filter out infeasible graph paths during the more precise Sridharan-Bodik analysis.

Targeted inefficiency. At the heart of the CFL-reachability formulation proposed in [6] is a context-free language that models heap accesses and method calls/returns. Given a graph representation of the program (described in Section 2), a

variable v can point to an object o if there exists a path from o to v labeled with a string from this language. Specifically, v can point to o if there exists a pair of statements $v = a.f$ and $b.f = o$ such that a and b are aliases. Deciding if a and b are aliases requires finding an object o' that may flow to both a and b. This check may trigger further recursive field access checks and context sensitivity checks (essentially, checks for matched parentheses) that can span many methods and classes. All checks transitively triggered need to be performed every time the analysis tries to verify whether a and b may be aliases, because the results may be different under different calling contexts (i.e., a method may be analyzed under different sequences of call sites starting from `main`). There could be a large number of calling contexts for a method and it may be expensive to repeat the check for each one of them.

Recent work [11,12] has identified that there exists a large number of *equivalent* calling contexts. The points-to sets of a variable under the equivalent contexts are the same; thus, distinguishing these contexts from each other is unnecessary. This observation also applies to aliasing. Suppose variables a and b may point to the same object o under two sets of equivalent contexts C_1 and C_2, respectively. Clearly, a and b may be aliases under $C_1 \cap C_2$. It is desirable for the analysis from [6] to remember the aliasing relationship of a and b for this entire set of contexts, so that this relationship needs to be computed only once for all contexts in the set. However, because in [6] the context-sensitivity check is performed along with the field access check, the context equivalence classes are not yet known when the aliasing relationship of a and b is computed. Ideally, a separate analysis can be performed to pre-compute context equivalence class information for all pairs of variables in the program. This information can be provided to the points-to analysis from [6], which will then be able to reuse the aliasing relationships under their corresponding equivalent calling contexts. In addition, this pre-analysis has to be sufficiently inexpensive so that its cost can be justified from the time saving of the subsequent points-to analysis.

Proposed approach. Since the number of calling contexts (i.e., sequences of call graph edges) in a Java program is usually extremely large even when treating recursion approximately, the proposed analysis adopts the following approach: instead of computing context equivalence classes for every pair of variables, it focuses on pairs that are not aliases *under any possible calling contexts*. This information is useful for early elimination of infeasible paths. The analysis from [6] does not have to check whether a and b are aliases if this pre-analysis has already concluded that they cannot possibly be aliases under any calling context. The pre-analysis will thus be referred to as a *must-not-alias analysis*.

The key to the success of the proposed approach is to make the must-not-alias analysis sufficiently inexpensive while maintaining relatively high precision. Several novel techniques are employed to achieve this goal:

- Aliases are obtained *directly* by performing a new form of CFL-reachability, instead of obtaining them by intersecting points-to sets [7].
- The *heap access* check of the analysis is formulated as an all-pairs CFL-reachability problem over a simplified balanced-parentheses language [13],

which leads to an efficient algorithm that has lower complexity than solving the Sridharan-Bodik CFL-reachability. The simplification of the language is achieved by computing CFL-reachability over a program representation referred to as an *interprocedural symbolic points-to graph*, which introduces approximations for heap loads and stores.
– The *context sensitivity* check is performed by combining bottom-up inlining of methods with 1-level object cloning (i.e., replicating each object for each call graph edge that enters the object-creating method). Hence, the analysis is fully-context-sensitive for pointer variables (with an approximation for recursion), but only 1-level context-sensitive for pointer targets. This approach appears to achieve the desired balance between cost and precision.

The must-not-alias analysis was implemented in Soot [14,15] and was used to pre-compute alias information for use by the subsequent Sridharan-Bodik points-to analysis. As shown experimentally, the approach effectively reduces the search space of the points-to analysis and eliminates unnecessary auxiliary work. On average over 19 Java programs, the modified points-to analysis (including the alias pre-analysis) is *more than three times faster* than the original analysis.

2 Background

This section provides a brief description of the CFL-reachability formulation of context-sensitive points-to analysis for Java [6]. It also illustrates the key idea of our approach through an example.

2.1 CFL-Reachability Formulation

The CFL-reachability problem is an extension of standard graph reachability that allows for filtering of uninteresting paths. Given a directed graph with labeled edges, a relation R over graph nodes can be formulated as a CFL-reachability problem by defining a context-free grammar such that a pair of nodes $(n, n') \in R$ if and only if there exists a path from n to n' for which the sequence of edge labels along the path is a word belonging to the language L defined by the grammar. Such a path will be referred to as an L-path. If there exists an L-path from n to n', then n' is L-reachable from n (denoted by $n\ L\ n'$). For any non-terminal S in L's grammar, S-paths and $n\ S\ n'$ are defined similarly.

A variety of program analyses can be stated as CFL-reachability problems [8]. Recent developments in points-to analysis for Java [16,6] extend this formulation to model (1) context sensitivity via method entries and exits, and (2) heap accesses via object field reads and writes. A demand-driven analysis is formulated as a single-source L-reachability problem which determines all nodes n' such that $n\ L\ n'$ for a given source node n. The analysis can be expressed by CFL-reachability for language $L_{\mathsf{F}} \cap R_{\mathsf{C}}$. Language L_{F}, where F stands for "flows-to", ensures precise handling of field accesses. Regular language R_{C} ensures a degree of calling context sensitivity. Both languages encode balanced-parentheses properties.

Graph representation. L_{F}-reachability is performed on a graph representation G of a Java program, such that if a heap object represented by the abstract location o can flow to variable v during the execution of the program, there exists an L_{F} path in G from o to v. Graph G is constructed by creating edges for the following canonical statements:

- Allocation $x = \text{new } O$: edge $o \xrightarrow{\mathsf{new}} x \in G$
- Assignment $x = y$: edge $y \xrightarrow{\mathsf{assign}} x \in G$
- Field write $x.f = y$: edge $y \xrightarrow{\mathsf{store(f)}} x \in G$
- Field read $x = y.f$: edge $y \xrightarrow{\mathsf{load(f)}} x \in G$

Parameter passing is represented as assignments from actuals to formals; method return values are treated similarly. Writes and reads of array elements are handled by collapsing all elements into an artificial field *arr_elm*.

Language L_{F}. First, consider a simplified graph G with only new and assign edges. In this case the language is regular and its grammar can be written simply as $\mathit{flowsTo} \rightarrow \mathsf{new}\,(\,\mathsf{assign}\,)^*$, which shows the transitive flow due to `assign` edges. Clearly, o *flowsTo* v in G means that o belongs to the points-to set of v.

For field accesses, inverse edges are introduced to allow a CFL-reachability formulation. For each graph edge $x \rightarrow y$ labeled with t, an edge $y \rightarrow x$ labeled with $\bar{t}$ is introduced. For any path p, an inverse path $\bar{p}$ can be constructed by reversing the order of edges in p and replacing each edge with its inverse. In the grammar this is captured by a new non-terminal $\overline{\mathit{flowsTo}}$ used to represent the inverse paths for *flowsTo* paths. For example, if there exists a *flowsTo* path from object o to variable v, there also exists a $\overline{\mathit{flowsTo}}$ path from v to o.

May-alias relationships can be modeled by defining a non-terminal *alias* such that $\mathit{alias} \rightarrow \overline{\mathit{flowsTo}}\,\mathit{flowsTo}$. Two variables a and b may alias if there exists an object o such that o can flow to both a and b. The field-sensitive points-to relationships can be modeled by $\mathit{flowsTo} \rightarrow \mathsf{new}\,(\,\mathsf{assign} \mid \mathsf{store(f)}\ \mathit{alias}\ \mathsf{load(f)}\,)^*$. This production checks for balanced pairs of store(f) and load(f) operations, taking into account the potential aliasing between the variables through which the store and the load occur.

Language R_{C}. The context sensitivity of the analysis ensures that method entries and exits are balanced parentheses: $C \rightarrow \mathsf{entry(i)}\ C\ \mathsf{exit(i)} \mid C\ C \mid \epsilon$. Here entry(i) and exit(i) correspond to the i-th call site in the program. This production describes only a subset of the language, where all parentheses are fully balanced. Since a realizable path does not need to start and end in the same method, the full definition of R_{C} also allows a prefix with unbalanced closed parentheses and a suffix with unbalanced open parentheses [6]. In the absence of recursion, the balanced-parentheses language is a finite regular language (thus the notation R_{C} instead of L_{C}); approximations are introduced as necessary to handle recursive calls. Context sensitivity is achieved by considering entries and exits along a L_{F} path and ensuring that the resulting string is in R_{C}.

2.2 CFL-Reachability Example

Figure 1 shows an example with an implementation of a `List` class, which is instantiated twice to hold two objects of class `A`. One of the `List` instances is wrapped in a `ListClient` object, which declares a method `retrieve` to obtain an object contained in its list. We will use t_i to denote the variable t whose first occurrence is at line i, and o_i to denote the abstract object for the allocation site at line i. For example, s_{31} and o_{26} represent variable s declared at line 31 and the `A` object created at line 26, respectively. Literal `"abc"` is used to denote the corresponding string object. Class `A` has a string-typed field `f`, initialized to some default value in `A`'s constructor; the actual code for `A` is not shown.

The program representation for this example is shown in Figure 2; for simplicity, the inverse edges are not shown. Each entry and exit edge is also treated as an assign edge for L_F, in order to represent parameter passing and method returns. To simplify the figure, edges due to the call at line 32 are not shown. The context-insensitive points-to pairs are defined by *flowsTo* paths. For example, there exists such a path from o_{26} to s_{31}. To see this, consider that $this_{17}$ *alias* $this_{19}$ (due to o_{27}) and therefore l_{17} *flowsTo* t_{19} due to the matching store and load of field `list`. Based on this, $this_5$ *alias* $this_{11}$ due to o_{25} (note that because of the call at line 32, they are also aliases due to o_{28}). Since $this_5$ *alias* $this_7$ (due to o_{25} or due to o_{28}), it can be concluded that t_7 *alias* t_{11} (due to o_4). This leads to the *flowsTo* path $o_{26} \rightarrow t_{26} \rightarrow m_6 \rightarrow t_7 \rightarrow \ldots \rightarrow t_{11} \rightarrow p_{12} \rightarrow r_{20} \rightarrow s_{31}$.

Since the precise computation of *flowsTo* path can be expensive, the analysis from [6] employs an approximation by introducing artificial *match edges*. If, due to aliasing, there may be a path from the source of a store(f) edge to the target of a load(f) edge, a match edge is added between the two nodes. Such edges are added before the balanced-parentheses checks for heap accesses are performed. An initial approximate solution is computed using the match edges. All encountered match edges are then removed, and the paths between their endpoints are explored. These new paths themselves may contain new match edges. In the next iteration of refinement, these newly-discovered match edges

```
1 class List{                                    18    Object retrieve(){
2    Object[] elems;                             19       t = this.list;
3    int count;                                  20       Object r = t.get(0);
4    List(){ t = new Object[10];                 21       return r;
5             this.elems = t; }                  22    }
6    void add(Object m){                         23 }
7       t = this.elems;                          24 static void main(String[] args){
8       t[count++] = m;                          25    List l1 = new List();
9    }                                           26    A t = new A(); l1.add(t);
10   Object get(int ind){                        27    ListClient client = new ListClient(l1);
11      t = this.elems;                          28    List l2 = new List();
12      p = t[ind]; return p;                    29    A i = new A(); i.f = "abc";
13   }                                           30    l2.add(i);
14 }                                             31    A s = (A)client.retrieve();
15 class ListClient{                             32    A j = (A)l2.get(0);
16    List list;                                33    String str = s.f;
17    ListClient(List l){ this.list = l; }      34 }}
```

Fig. 1. Code example

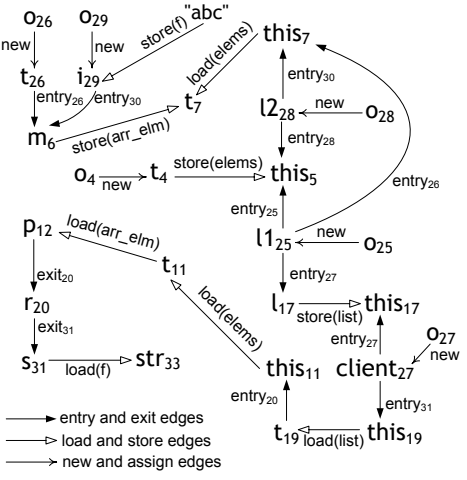

Fig. 2. Illustration of CFL-reachability

are removed, etc. Since we are interested in a highly-refined solution, the example and the rest of the paper will assume complete refinement of match edges.

Not every *flowsTo* path is feasible. Consider, for example, path $o_{29} \rightarrow i_{29} \rightarrow m_6 \rightarrow t_7 \rightarrow \ldots \rightarrow t_{11} \rightarrow p_{12} \rightarrow r_{20} \rightarrow s_{31}$. Even though this is a valid *flowsTo* path, the entry and exit edges along the path are not properly matched. To see this, consider the following subpaths. First, for the path from $this_5$ to s_{31}, the sequence of entry/exit edges is $\overline{\mathsf{entry}(25)}, \mathsf{entry}(27), \overline{\mathsf{entry}(27)}, \mathsf{entry}(31), \mathsf{entry}(20), \mathsf{exit}(20), \mathsf{exit}(31)$. Here the inverse of an entry edge can be thought of as an exit edge and vice versa. All edges except for the first one are properly matched. Second, consider the two paths from o_{29} to $this_5$: the first one goes through o_{28} and the second one goes through o_{25}. The first path contains edges $\mathsf{entry}(30), \overline{\mathsf{entry}(30)}, \mathsf{entry}(28)$ and the second path contains edges $\mathsf{entry}(30), \overline{\mathsf{entry}(26)}, \mathsf{entry}(25)$. Neither path can be combined with the path having the unmatched $\overline{\mathsf{entry}(25)}$ to form a valid string in R_{C}. On the other hand, there exists a path from o_{26} to $this_5$ with edges $\mathsf{entry}(26), \overline{\mathsf{entry}(26)}, \mathsf{entry}(25)$, which can be correctly combined.

In the example, suppose that an analysis client queries the points-to set of s_{31}. The analysis starts from the variable and reaches p_{12} after traversing back through the two exit edges. It then finds the matched `arr_elm` edges from m_6 to t_7 and from t_{11} to p_{12}. At this point, the analysis does not know whether t_7 and t_{11} can alias, and hence, it queries the points-to sets of t_7 and t_{11}. For t_7, due to the matched load and store edges for `elems`, the analysis tries to determine whether $this_5$ and $this_7$ can be aliases. Since o_{25} can flow to both variables, they indeed are aliases. (Note that o_{28} also flows to both variables, but its inclusion in the full path starting at s_{31} leads to unbalanced entry/exit edges.) Eventually, the conclusion is that t_7 and t_{11} can alias because there exists an *alias* path between them with balanced entry and exit edges. From this point, the analysis can continue with a backward traversal from m_6, which encounters o_{26} and o_{29}.

Only the path from o_{26} to s_{31} has the balanced entry/exit property, and the analysis reports that the points-to set of s_{31} is $\{o_{26}\}$.

2.3 Using Must-Not-Alias Information

The proposed must-not-alias analysis is based on a program representation we refer to as the interprocedural symbolic points-to graph (ISPG). The definition and the construction algorithm for the ISPG are presented Section 3. Using this representation, it is possible to conclude that certain variables are definitely not aliases. The description of this analysis algorithm is presented in Section 4 and Section 5. For the example in Figure 2, the analysis can conclude that i_{29} and s_{31} cannot be aliases under any calling context. This information is pre-computed before the CFL-reachability-based points-to analysis from [6] starts.

Consider a match edge — that is, the edge from z to w for a pair of matching $z \xrightarrow{\mathsf{store(f)}} x$ and $y \xrightarrow{\mathsf{load(f)}} w$. When the points-to analysis removes this edge, it normally would have to explore the paths from x to y to decide whether x *alias* y. Instead, it queries our must-not-alias analysis to determine whether x and y may be aliases. If they cannot be aliases, further exploration is unnecessary. For illustration, consider an example where a client asks for the points-to set of str_{33}. The store(f) edge entering i_{29} and the load(f) edge exiting s_{31} mean that the points-to analysis needs to determine whether i_{29} *alias* s_{31}. However, our must-not-alias information has already concluded that under no calling context these two variables can be aliases. Hence, the points-to analysis can quickly skip the check and conclude that `"abc"` does not belong to the points-to set of str_{33}. In contrast, without the must-not-alias information, the points-to analysis would have to explore further for an *alias* path between s_{31} and i_{26}, which involves traversal of almost the entire graph.

3 Program Representation for Must-Not-Alias Analysis

The must-not-alias analysis runs on the interprocedural symbolic points-to graph. This section describes the approach for ISPG construction. Section 3.1 shows the first phase of the approach, in which an SPG is constructed separately for each method. Section 3.2 discusses the second phase which produces the final ISPG. To simplify the presentation, the algorithm is described under the assumption of an input program with no static fields and no dynamic dispatch.

3.1 Symbolic Points-to Graph for a Method

The SPG for a method is an extension of a standard points-to graph, with the following types of nodes and edges:

- $\mathcal{V}$ is the domain of variable nodes (i.e., local variables and formal parameters)
- $\mathcal{O}$ is the domain of allocation nodes for `new` expressions
- $\mathcal{S}$ is the domain of symbolic nodes, which are created to represent objects that are not visible in the method

- Edge $v \rightarrow o_i \in \mathcal{V} \times \mathcal{O}$ shows that variable v points to object o_i
- Edge $v \rightarrow s_i \in \mathcal{V} \times \mathcal{S}$ shows that (1) the allocation node that v may point to is defined outside the method, and (2) symbolic node s_i is used as a placeholder for this allocation node
- Edge $o_i \xrightarrow{f} o_j \in (\mathcal{O} \cup \mathcal{S}) \times \mathit{Fields} \times (\mathcal{O} \cup \mathcal{S})$ shows that field f of allocation or symbolic node o_i points to allocation or symbolic node o_j

In order to introduce symbolic nodes as placeholders for outside objects, the CFL-reachability graph representation of a method is augmented with the following types of edges. An edge $s \xrightarrow{\mathsf{new}} \mathit{fp}$ is added for each formal parameter *fp* of the method. Here s is a symbolic node, created to represent the objects that *fp* points to upon entry into the method. A separate symbolic node is created for each formal parameter. Similarly, for each call site $v = m()$, an edge $s \xrightarrow{\mathsf{new}} v$ is created to represent the objects returned by the call. A separate symbolic node is introduced for each call site. For each field dereference expression $v.f$ whose value is read at least once, edges $s \xrightarrow{\mathsf{new}} t$ and $t \xrightarrow{\mathsf{store(f)}} v$ are created. Here symbolic node s denotes the heap location represented by $v.f$ before the method is invoked, and t is a temporary variable created to connect s and $v.f$.

The SPG for a method m can be constructed by computing intraprocedural *flowsTo* paths for all $v \in \mathcal{V}$. A points-to edge $v \rightarrow o \in \mathcal{V} \times (\mathcal{O} \cup \mathcal{S})$ is added to the SPG if o *flowsTo* v. A points-to edge $o_i \xrightarrow{f} o_j$ is added if there exists $x \xrightarrow{\mathsf{store(f)}} y$ in m's representation such that o_i *flowsTo* y and o_j *flowsTo* x.

Both symbolic nodes and allocation nodes represent abstract heap locations. A variable that points to a symbolic node n_1 and another variable that points to an allocation/symbolic node n_2 may be aliases if it is eventually decided that n_1 and n_2 could represent the same abstract location. The relationships among allocation/symbolic nodes in an SPG are ambiguous. A symbolic node, even though it is intended to represent outside objects, may sometimes also represent inside objects (e.g., when the return value at a call site is a reference to some object created in the caller). Furthermore, two distinct symbolic nodes could represent the same object — e.g., due to aliasing of two actual parameters at some call site invoking the method under analysis. Such relationships are accounted for later, when the ISPG is constructed.

The introduction of symbolic nodes is similar to pointer analyses from [17,18,19,20,12]. These analysis algorithms use symbolic nodes to compute a summary for a caller from the summaries of its callees during a bottom-up traversal of the DAG of strongly connected components (SCC-DAG) in the call graph. Unlike previous analyses that create symbolic nodes to compute the actual points-to solution, we do so to approximate the flow of heap objects in order to perform a subsequent CFL-reachability analysis on the ISPG. This reachability analysis, described in Section 4 and Section 5, identifies alias relationships among allocation and symbolic nodes, and ignores the points-to relationships involving variables. These results are used to reduce the cost of the *alias* path exploration for the points-to analysis outlined in Section 2.

$$\frac{\text{actual } a_i \in \mathcal{V}_p, \text{formal } f_i \in \mathcal{V}_m \quad a_i \to n \in SPG_p, f_i \to s \in SPG_m}{n \xrightarrow{\text{entry(e)}} s \in ISPG}$$

$$\frac{ret \in \mathcal{V}_m, r \in \mathcal{V}_p, ret \to n \in SPG_m, r \to s \in SPG_p}{n \xrightarrow{\text{exit(e)}} s \in ISPG}$$

Fig. 3. Connecting method-level SPGs

3.2 Interprocedural Symbolic Points-To Graph

In order to perform interprocedural analysis, the SPGs of individual methods are connected to build the ISPG for the entire program. The ISPG is not a completely resolved points-to graph, but rather a graph where SPGs are trivially connected. Figure 3 shows the rules for ISPG construction at a call site $r = a_0.m(a_1, \ldots, a_i, \ldots)$. Suppose the call site is contained in method p.

In the figure, e denotes the call graph edge that goes from p to m through this particular call site. The callee m is assumed to have an artificial local variable *ret* in which the return value of the method is stored. For a formal-actual pair (f_i, a_i), an entry edge is added between each object/symbolic node n that a_i points to in caller and the symbolic node s created for f_i in the callee. The second rule creates an exit edge to connect the returned object/symbolic nodes n_1 from the callee and the symbolic node s created for r at the call site. Similarly to the entry and exit edges in the CFL-reachability formulation from Section 2, the entry and exit edges in the ISPG are added to represent parameter passing and value return. The annotations with call graph edge e for these ISPG edges will be used later to achieve context sensitivity in the must-not-alias analysis.

Figure 4 shows part of the ISPG built for the running example, which connects SPGs for methods `main`, `retrieve`, `add`, and `get`. Symbolic nodes are represented by shaded boxes, and named globally (instead of using code line numbers). For example, in method `add`, S_1 is the symbolic object created for `this`, S_2 is created due to the read of `this.elems`, and the ISPG contains an edge from S_1 to S_2. Name S_3 represents the object to which formal `m` points; due to `t[count++]=m`, the SPG contains an edge from S_2 to S_3 labeled with *arr_elm*. Due to the calls to `add` at lines 26 and 30, entry edges connect O_{25} and O_{28} with S_1, and O_{26} and O_{29} with S_3.

The *backbone* of an ISPG is the subgraph induced by the set of all allocation nodes and symbolic nodes. Edges in the backbone are either field points-to edges $o_i \xrightarrow{f} o_j$ computed by the intraprocedural construction described in Section 3.1, or entry/exit edges created at call sites, as defined above. Variable points-to edges (e.g., $this_7 \to S_1$ from above) are not included in the backbone. Section 4 and Section 5 show how to perform CFL-reachability on the backbone of an ISPG to compute the must-not-alias information.

Why use the ISPG? The benefits of performing the alias analysis using the ISPG backbone are two-fold. First, this graph abstracts away variable nodes, and partitions the heap using symbolic and allocation nodes (essentially, by defining

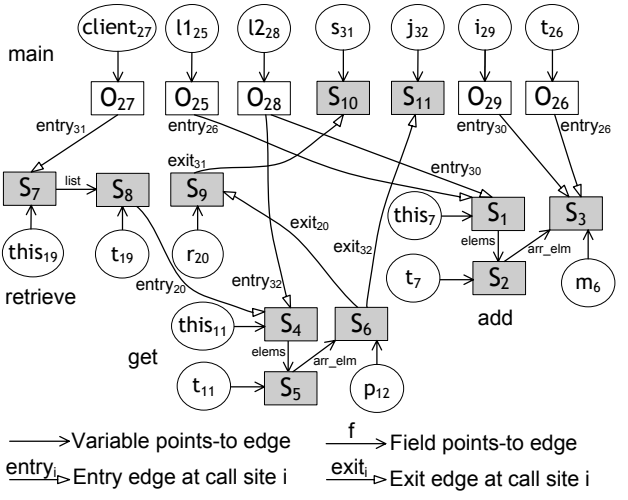

Fig. 4. Illustration of the ISPG for the running example

equivalence classes for these nodes). Hence, the backbone of an ISPG contains fewer nodes and edges than the graph representation for CFL-reachability points-to analysis from Section 2. Second, the ISPG allows simple modeling of the alias computation — the CFL-reachability used to formulate the context-insensitive version of the problem, as described in the next section, is restricted to a language *memAlias* which is simpler to handle than the more general CFL-reachability for context-insensitive points-to analysis [6,7].

4 Context-Insensitive Memory Alias Formulation

This section defines a CFL-reachability formulation of a context-insensitive version of the must-not-alias analysis. The context sensitivity aspects are described in the next section. Hereafter, the term "node" will be used as shorthand for "allocation or symbolic node in the ISPG", unless specified otherwise.

Two nodes o_1 and o_2 are memory aliases if they may denote the same heap memory location. We describe the memory aliases using relation $memAlias \subseteq (\mathcal{O} \cup \mathcal{S}) \times (\mathcal{O} \cup \mathcal{S})$. This relation is reflexive, symmetric, and transitive, and therefore is an equivalence relation. The computation of *memAlias* is formulated as a CFL-reachability problem over the backbone of the ISPG. The relation has the following key property: for any pair of variables v_1 and v_2 in relation *alias* computed by the points-to analysis from Section 2, there must exist ISPG edges $v_1 \rightarrow o_1$ and $v_2 \rightarrow o_2$ such that the ISPG backbone contains a *memAlias* path from node o_1 to node o_2 (and also from o_2 to o_1). For a variable pair (v_1, v_2) for which such a pair (o_1, o_2) does *not* exist, the points-to analysis from Section 2 does not need to explore *alias* paths between v_1 and v_2, since all such work is guaranteed to be wasted. This section presents an efficient algorithm for solving

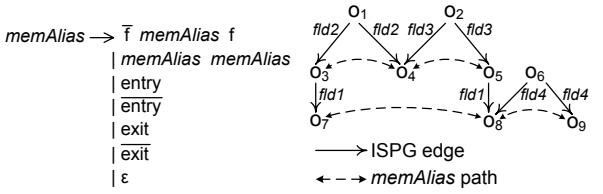

Fig. 5. Language *memAlias*

the all-pairs *memAlias*-path problem. We first assume that the backbone of the ISPG is free of recursive data structures. The approximation for recursive data structures is addressed later in the section.

Figure 5 shows the grammar for language *memAlias* and an example illustrating several such paths. An edge label f shows the field name for an ISPG edge $o_i \xrightarrow{f} o_j$. As before, $\bar{f}$ denotes the inverse of the edge labeled with f. The existence of a *memAlias* path from o_1 to o_2 also means that there is a *memAlias* path from o_1 to o_2. For this reason, the figure uses double-headed arrows to show such paths. In this example, o_7 *memAlias* o_9 because of path o_7 $\overline{fld1}$ o_3 $\overline{fld2}$ o_1 $fld2$ o_4 $\overline{fld3}$ o_2 $fld3$ o_5 $fld1$ o_8 $\overline{fld4}$ o_6 $fld4$ o_9.

Example. For illustration, consider the ISPG shown in Figure 4. Some of the *memAlias* pairs in this graph are (S_7, O_{27}), (S_1, S_4), (S_2, S_5), (S_3, S_6), (S_6, O_{29}), (S_{11}, O_{29}), (S_9, O_{29}), and (S_{10}, O_{29}).

Production *memAlias* → *memAlias memAlias* encodes the transitivity of the relation. The productions for entry/exit edges and their inverses allow arbitrary occurrences of such edges along a *memAlias* path; this is due to the context insensitivity of this version of the analysis. Production *memAlias* → $\bar{f}$ *memAlias* f says that if x and y are reachable from the same node z through two paths, and the sequences of fields along the paths are the same, x and y may denote the same memory location. This is an over-approximation that is less precise than the alias information computed by the analysis from Section 2.

Consider the sources of this imprecision. Suppose x, y, z and w are nodes in the ISPG and variables v_x, v_y, v_z and v_w point to them. If w and z are memory aliases, and there exist two points-to edges $x \xleftarrow{f} z$ and $w \xrightarrow{f} y$ in the ISPG, x and y are memory aliases based on our definition. The existence of these two edges can be due to four combinations of loads and stores in the program:

- $v_x = v_z.f$ and $v_y = v_w.f$: in this case, x and y are true memory aliases
- $v_x = v_z.f$ and $v_w.f = v_y$: x and y are true memory aliases because there exists a *flowsTo* path from v_y to v_x.
- $v_z.f = v_x$ and $v_y = v_w.f$: again, x and y are true memory aliases
- $v_z.f = v_x$ and $v_w.f = v_y$: this case is handled imprecisely, since x and y do not need to be aliases. Our approach allows this one source of imprecision in order to achieve low analysis cost.

Precision improvement. It is important to note that two allocation nodes (i.e., non-symbolic nodes) are never memory aliases even if there exists a *memAlias* path between them. Hence, in the final solution computed by the analysis, node x is considered to not be an alias of y if (1) there does not exist a *memAlias* path between them, or (2) all *memAlias* paths between them are of the form $x \xrightarrow{\overline{f_i \cdots f_0}} o_i \ memAlias \ o_j \xrightarrow{f_0 \cdots f_i} y$, where o_i and o_j are distinct allocation nodes.

Soundness. The formulation presented above defines a sound must-not-alias analysis. Consider any two variables v_1 and v_2 such that v_1 *alias* v_2 in the approach from Section 2. It is always the case that there exist ISPG edges $v_1 \rightarrow o_1$ and $v_2 \rightarrow o_2$ such that o_1 and o_2 are declared to be memory aliases by our analysis. Here one of these nodes is a symbolic node, and the other one is either a symbolic node or an allocation node. The proof of this property is not presented in the paper due to space limitations.

4.1 Solving All-Pairs *memAlias*-Reachability

Solving CFL-reachability on the mutually-recursive languages *alias* and *flowsTo* from [6] yields $O(m^3k^3)$ running time, where m is the number of nodes in the program representation and k is the size of L_{F}. As observed in existing work [21,22,13], the generic bound of $O(m^3k^3)$ can be improved substantially in specific cases, by taking advantage of certain properties of the underlying grammar. This is exactly the basis for our approach: the algorithm for *memAlias*-reachability runs in $O(n^4)$ where n is the number of nodes in the ISPG backbone. The value of n is smaller than m, because variable nodes are abstracted away in the ISPG; Section 6 quantifies this observation. This algorithm speeds up the computation by taking advantage of the symmetric property of *memAlias* paths. This subsection assumes that the ISPG is an acyclic graph; the extension to handle recursive types is presented in the next subsection.

The pseudocode of the analysis is shown in Algorithm 1. The first phase considers production $memAlias \rightarrow \bar{f}\ memAlias\ f$. Strings in the corresponding language are palindromes (e.g., *abcddcba*). Once a *memAlias* path is found between nodes a and b, pair (a, b) is added to set *memAlias* (line 9). The *memAlias* set for this particular language can be computed through depth-first traversal of the ISPG, starting from each node n (line 4-13). The graph traversal (line 5) is implemented by function ComputeReachableNodes, which finds all nodes n' that are reachable from n, and their respective sequences $l_{n,n'}$ of labels along the paths from which they are reached (n' could be n). Sequence $l_{n,n'}$ will be referred to as the *reachability string* for a path between n and n'. Due to space limitations, we omit the detailed description of this function. Also, for simplicity of presentation, we assume that there exists only one path between n and n'. Multiple paths and their reachability strings can be handled in a similar manner. The function returns a map which maps each pair (n, n') to its reachability string $l_{n,n'}$. A similar map *cache* accumulates the reachability information for all nodes (line 7). For any pair of nodes (a, b) reachable from n such that $l_{n,a} = l_{n,b}$, there exists a *memAlias*

Algorithm 1. Pseudocode for solving all-pairs *memAlias*-reachability.

```
SolveMemAliasReachability (ISPG backbone IG)
 1: Map cache // a cache map that maps a pair of nodes (n, a) to their reachability string l_{n,a}
 2: List endNodes // a worklist containing pairs of nodes that are two ends of a sequence of edges
    that forms a memAlias path
 3: /* phase 1: consider only production memAlias → f̄ memAlias f */
 4: for each node n in IG do
 5:    Map m ⟵ ComputeReachableNodes(n)
 6:    for each pair of entries [(n, a), l_{n,a}] and [(n, b), l_{n,b}] in m do
 7:       cache ⟵ cache ∪ [(n, a), l_{n,a}] ∪ [(n, b), l_{n,b}] // remember the reachability information
 8:       if l_{n,a} = l_{n,b} then
 9:          memAlias ⟵ memAlias ∪ (a, b) // a memAlias path exists between a and b
10:          endNodes ⟵ endNodes ∪ (a, b)
11:       end if
12:    end for
13: end for
14: /* phase 2: consider production memAlias → memAlias memAlias */
15: /* a worklist-based algorithm */
16: while endNodes ≠ ∅ do
17:    remove a pair (a, b) from endNodes
18:    if (a, b) has been processed then
19:       continue
20:    else
21:       mark (a, b) as processed
22:    end if
23:    for each (c, a) in memAlias do
24:       for each (b, d) in memAlias do
25:          memAlias ⟵ memAlias ∪ (c, d)
26:          endNodes ⟵ endNodes ∪ (c, d)
27:       end for
28:    end for
29:    for each [(a, c), l_{a,c}] in cache do
30:       for each [(b, d), l_{b,d}] in cache do
31:          if l_{a,c} = l_{b,d} then
32:             /* add to the worklist all pairs of nodes with a memAlias path between them */
33:             memAlias ⟵ memAlias ∪ (c, d)
34:             endNodes ⟵ endNodes ∪ (c, d)
35:          end if
36:       end for
37:    end for
38: end while
```

path between them (line 9). This pair is added to relation *memAlias* and to a list *endNodes* for further processing.

Phase 1 complexity. Each graph traversal at line 5 takes time $O(m)$, where m is the number of edges in the ISPG backbone. The *for* loop at lines 6-12 takes $O(m^2)$. Note that when a reachability string is generated, the hashcode of the string is computed and remembered. Hence, line 8 essentially compares two integers, which takes constant time. Since there are $O(m)$ nodes in the program, the complexity of computing all *memAlias* paths in this phase is $O(m^3)$.

The second phase employs a worklist-based iteration which considers the entire *memAlias* language. As usual, the phase computes a closure by continuously processing pairs of nodes between which there exists a *memAlias* path. Such pairs of nodes are contained in list *endNodes*, and are removed from the list upon processing. Lines 23-28 update the transitive closure of *memAlias*. Next, all nodes reachable from a or from b are retrieved from *cache*, together with their reachability strings (line 29 and 30). Due to the caching of reachability information,

graph traversal is no longer needed. If reachability strings $l_{a,c}$ and $l_{b,d}$ match, a *memAlias* path exists between c and d. Hence, pair (c, d) is added to *memAlias* and to worklist *endNodes*.

Phase 2 complexity. Each *while* iteration (lines 17-37) takes $O(m^2)$ time. The worst case is that all possible pairs of nodes (a, b) in the ISPG backbone have been added to *endNodes* and processed. There are $O(m^2)$ such pairs; hence, the worst-case time complexity for the entire algorithm is $O(m^4)$.

Although a slightly modified algorithm is used in the actual context-sensitive version of the analysis (presented in the next section), the description from above illustrates the key to computing *memAlias*-reachability. It is important to note again that the design of this efficient algorithm is due to the specific structure of the grammar of language *memAlias*. Since the grammar is symmetric and self-recursive (instead of being mutually-recursive), the finite number of open field parentheses can be computed a priori (i.e., stored in the cache). Thus, at the expense of the single source of imprecision discussed earlier, this approach avoids the cost of the more general and expensive CFL-reachability computation described in Section 2.

4.2 Approximation for Recursive Data Structures

Existing analysis algorithms have to introduce *regularity* into the context-free language and approximate either recursive calls [16,6], or recursive data structures [23] over the regularized language. Because *memAlias*-reachability is performed over the backbone of an ISPG, we focus on the handling of cycles consisting of field points-to edges caused by recursive data structures.

The key to the handling of a recursive type is to collapse an ISPG SCC caused by the recursion. For any path going through a SCC, a *wildcard* (*) is used to replace the substring of the path that includes the SCC nodes. The wildcard can represent an arbitrary string. During the string comparison performed by Algorithm 1, two paths match as long as the regular expressions representing them have non-empty intersection. Figure 6 shows an example of reachability checking in the presence of recursive types. In this example, it is necessary to check whether the two regular expressions $fld1 * fld2\ fld3\ fld6$ and $fld1 * fld6$ have non-empty intersection.

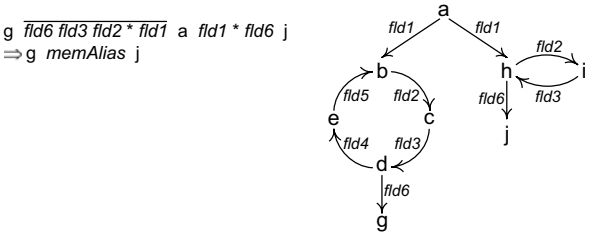

Fig. 6. Handling of recursive data structures

The handling of recursive types requires modifications to the comparison of reachability strings. If neither string contains wildcards, no changes are needed. If at least one string contains wildcards, it is necessary to consider the corresponding finite automata and to check if there exists a common sequence of state transitions that can lead both automata to accepting states. Although deciding whether two general-form regular expressions have non-empty intersection is a non-trivial problem, the alias analysis needs to handle a significantly simplified version of the problem in which the regular expressions do not contain general closure. In practice, the cost of this processing is insignificant.

Formulation of points-to analysis from *memAlias***.** Although we use language *memAlias* to formulate an alias analysis, a points-to relation *pointsTo* can be easily derived from the following production: $pointsTo \rightarrow \mathsf{var\_pts}\ memAlias$. Here var_pts is the label on an ISPG edge from a variable node to a (symbolic or allocation) node. The set of all such edges forms the complement of the backbone edge set. All var_pts edges are constructed in the intraprocedural phase of ISPG construction, as described in Section 3.1. For example, in Figure 4, $j_{32}\ pointsTo\ O_{29}$ because $j_{32}\ \mathsf{var\_pts}\ S_{11}$ and $S_{11}\ memAlias\ O_{29}$ hold.

Due to the approximations described earlier and the limited context sensitivity (discussed shortly), the points-to information derived from *memAlias* is less precise than the solution computed by the analysis from Section 2. However, as shown in our experimental results, a large number of infeasible alias pairs can be eliminated early, leading to considerable overall performance improvement.

5 Context-Sensitive Must-Not-Alias Analysis

A context-sensitivity check can be performed along with the heap access check to guarantee that a *memAlias* path contains balanced entry and exit edges. Previous work [24,11] has shown that heap cloning (i.e., context-sensitive treatment not only of pointer variables, but also of pointer targets) is one of the most important factors that contribute to the precision of the analysis. Existing analysis algorithms achieve heap cloning primarily in two ways: (1) they maintain a push-down automaton to solve CFL-reachability over language R_{C} described in Section 2, or (2) they explicitly clone pointer variables and pointer targets (i.e., allocation nodes) for each distinct calling context [11,25,12], so that the cloned nodes are automatically distinguished. In order to achieve both efficiency and precision, we develop a hybrid algorithm that combines both approaches.

5.1 Analysis Overview

This subsection gives a high-level overview of the proposed approach; the detailed definitions are presented in the next subsection. The analysis uses bottom-up propagation on the call graph to ensure context sensitivity for pointer variables, with appropriate approximations for recursion. This enables a summary-based approach to propagate reachability strings from callees to callers, which yields efficiency. By composing summary functions (i.e., reachability strings for nodes

that parameters and return variables point to in the ISPG) at call sites, reachability strings for nodes in callees are concatenated with reachability strings for nodes in callers. At each call graph edge e, if the analysis enters the callee through edge entry(e), it has to exit through edge $\overline{\text{entry(e)}}$ or exit(e). This type of context sensitivity corresponds to the classical *functional* approach [26]. However, functional context sensitivity does not automatically enforce heap cloning. A (symbolic or allocation) node in a method may represent different objects if the method is inlined to a caller through different call chains. If these objects are not differentiated, imprecise *memAlias* paths may be derived.

Our proposal is to perform lightweight cloning for (symbolic and allocation) nodes when composing summary function at a call site. In this cloning, there may be several "clones" (copies) of an ISPG node, each annotated with a different calling context. The level of cloning, of course, has an impact on the analysis precision. Since the primary concern is efficiency, the level of cloning is restricted to 1, and thus, each symbolic or allocation node in the analysis has only one call graph edge associated with it. In fact, for some programs in our benchmark set, increasing the level of cloning to 2 (i.e., a chain of two call graph edges) makes the alias analysis too expensive compared to the cost reduction for the subsequent points-to analysis.

5.2 Analysis Details

This subsection defines the context-sensitive alias analysis using the rules shown in Figure 7. The analysis state is represented by cache map *cache*, worklist *endNodes*, and relation *memAlias*, whose functions are similar to those defined in Algorithm 1. In the rules, ϵ denotes the empty string and operator $\circ$ represents string concatenation.

The first rule describes the intraprocedural analysis with a rule of the form $endNodes, cache, memAlias \Rightarrow endNodes', cache', memAlias'$ with unprimed and primed symbols representing the state before and after an SPG field points-to edge $p \xrightarrow{f} o$ is traversed. The intraprocedural analysis performs backward traversal of the SPG for the method being processed, and updates the state as described above. When edge $p \xrightarrow{f} o$ is traversed backwards, the reachability from p to n is established for any n already reachable from o: that is, for any $[(o, n), l_{o,n}] \in cache$ where $l_{o,n}$ is a reachability string. Given the updated *cache*, it is necessary to consider the set of pairs (a, b) of nodes reachable from p such that the corresponding reachability strings $l_{p,a}$ and $l_{p,b}$ have non-empty intersection (represented by predicate OVERLAP); this processing is similar to the functionality of lines 6-12 in Algorithm 1.

The second rule describes the processing of an entry edge from o to s, corresponding to a call graph edge e. In this rule, p^e denotes the clone of node p for e. Here o represents a symbolic or allocation node to which an actual parameter points, s represents a symbolic node created for the corresponding formal parameter in the callee, and p^c is a node reachable from s. Node p^c may represent a node in the callee itself (i.e., when c is the empty string ϵ), or a node in a method deeper in the call graph that is cloned to the callee due to a previously-processed call. When

[Intraprocedural state update]
$[(o,n), l_{o,n}] \in cache$
$cache' = cache \cup [(p,n), f \circ l_{o,n}] \cup [(p,p), \epsilon]$
$pairs = \{\, (a,b) \mid [(p,a), l_{p,a}], [(p,b), l_{p,b}] \in cache' \wedge \textsc{Overlap}(l_{p,a}, l_{p,b}) \,\}$
$memAlias' = memAlias \cup pairs$
$endNodes' = endNodes \cup pairs$

$$endNodes, cache, memAlias \Rightarrow^{p \xrightarrow{f} o} endNodes', cache', memAlias'$$

[Method call]
$[(s, p^c), l_{s,p}] \in cache$
$p^x = p^e$ if $c = \epsilon$ and $p^x = p^c$ otherwise
$triples = \{\, [(o, p^x), l_{s,p}] \,\} \cup \{\, [(o, s^e), \epsilon] \,\} \cup$
$\{\, [(o,q), l_{s,p} \circ l_{n,q}] \mid (p^c \xrightarrow{\mathsf{exit(e)}} n \vee p^c \xrightarrow{\overline{\mathsf{entry(e)}}} n) \wedge [(n,q), l_{n,q}] \in cache \}$
$cache' = cache \cup triples$
$pairs = \{\, (a,b) \mid [(p,a), l_{p,a}], [(p,b), l_{p,b}] \in cache' \wedge \textsc{Overlap}(l_{p,a}, l_{p,b}) \,\}$
$memAlias' = memAlias \cup pairs$
$endNodes' = endNodes \cup pairs$

$$endNodes, cache, memAlias \Rightarrow^{o \xrightarrow{\mathsf{entry(e)}} s} endNodes', cache', memAlias'$$

Fig. 7. Inference rules defining the context-sensitive alias analysis algorithm

a call site is handled, all nodes that are reachable from a symbolic node created for a formal parameter, and all nodes that can reach a node pointed-to by a return variable, are cloned from the callee to the caller. All edges connecting them are cloned as well. The algorithm uses only 1-level cloning. If the existing context c of node p is empty, it is updated with the current call graph edge e; otherwise, the new context x remains c. Note that multiple-level cloning can be easily defined by modifying the definition of p^x.

In addition to updating the cache with (o, p^x) and (o, s^e), it is also necessary to consider any node n in the caller's SPG that is connected with p^c either through an $\mathsf{exit(e)}$ edge (i.e., p^c is a symbolic node pointed-to by the return variable), or through an $\overline{\mathsf{entry(e)}}$ edge (i.e., p^c is a symbolic node created for another formal parameter). The analysis retrieves all nodes q in the caller that are reachable from n, together with their corresponding reachability strings $l_{n,q}$, and updates the state accordingly. Now q becomes reachable from o, and the reachability string $l_{o,q}$ is thus the concatenation of $l_{s,p}$ and $l_{n,q}$.

After all edges in the caller are processed, the transitive closure computation shown at lines 16-38 of Algorithm 1 is invoked to find all *memAlias* pairs in the caller as well as all its (direct and transitive) callees. This processing is applied at each call graph edge e, during a bottom-up traversal of the call graph.

Termination. To ensure termination, the following approximation is adopted: when a call-graph-SCC method m is processed, edge $a \xrightarrow{f} b$ (which is reachable from m's formal parameter) is not cloned in its caller n if an edge $a^e \xrightarrow{f} b^e$ (where e is the call graph edge for the call from n to m) already exists in n.

Table 1. Java benchmarks

Benchmark	#Methods	#Statements	#SB/ISPG Nodes	#SB/ISPG Edges
compress	2344	43938	18778/10977	18374/3214
db	2352	44187	19062/11138	18621/3219
jack	2606	53375	22185/12605	21523/15560
javac	3520	66971	23858/14119	23258/3939
jess	2772	51021	22773/13421	21769/4754
mpegaudio	2528	55166	22446/12774	21749/4538
mtrt	2485	46969	20344/11878	19674/3453
soot-c	4583	71406	31054/18863	29971/5010
sablecc-j	8789	125538	44134/26512	42114/9365
jflex	4008	25150	31331/18248	30301/4971
muffin	4326	80370	33211/19659	32497/5282
jb	2393	43722	19179/11275	18881/3146
jlex	2423	49100	21482/11787	20643/3846
java_cup	2605	50315	22636/13214	21933/3438
polyglot	2322	42620	18739/10950	18337/3128
antlr	2998	57197	25505/15068	24462/4116
bloat	4994	79784	38002/23192	35861/5428
jython	4136	80067	34143/19969	33970/5179
ps	5278	84540	39627/23601	38746/5646

The processing of a SCC method stops as soon as the analysis determines that no more nodes need to be cloned to this method during the interprocedural propagation.

6 Experimental Evaluation

The proposed approach was implemented using the Soot 2.2.4 analysis framework [14,15]. The analyses included the Sun JDK 1.3.1_20 libraries, to allow comparison with previous work [11,6]. All experiments were performed on a machine with an Intel Xeon 2.8GHz CPU, and run with 2GB heap size. The experimental benchmarks, used in previous work [12], are shown in Table 1. Columns *Methods* and *Statements* show the number of methods in the original context-insensitive call graph computed by Soot's Spark component [27], and the number of statements in these methods. The ISPG was constructed using this call graph. Columns *#SB/ISPG Nodes (Edges)* shows the comparison between the number of nodes (edges) in the Sridharan-Bodik (SB) graph representation of the program, and the number of nodes (edges) in the corresponding ISPG backbone. On average, the numbers of ISPG backbone nodes and edges are 1.7× and 5.6× smaller than the numbers of SB nodes and edges, respectively.

The rest of this section presents an experimental comparison between the optimized version and the original version of the Sridharan-Bodik analysis. Specifically, queries were raised for the points-to set of each variable in the program.

6.1 Running Time Reduction

Table 2 compares the running times of the two analysis versions. Since the analysis cannot scale to compute fully-refined results, it allows users to specify a

Table 2. Analysis time (in seconds) and precision comparison

Benchmark	Original	Optimized				Speedup	Precision				
	SB	ISPG	Alias	SB′	Total		Casts	Ins	mA	1H	SB
compress	1101	59	20	203	282	3.9	6	0	0	0	2
db	1180	62	10	198	270	4.4	24	0	6	6	19
jack	1447	223	37	241	501	2.9	148	14	40	42	49
javac	1727	86	20	339	445	3.9	317	0	38	40	55
jess	1872	92	51	228	371	5.0	66	6	8	8	38
mpegaudio	866	56	20	185	261	3.3	13	1	4	4	4
mtrt	873	67	16	192	275	3.2	10	0	4	4	4
soot-c	3043	159	64	672	895	3.4	797	7	72	89	142
sablecc-j	4338	445	59	2350	2854	1.5	327	6	35	30	62
jflex	3181	151	43	1148	1342	2.4	580	1	12	2	43
muffin	3378	232	50	599	891	3.8	148	2	20	21	69
jb	802	58	9	287	354	2.3	38	0	3	2	24
jlex	833	54	14	237	305	2.7	47	1	4	3	14
java_cup	1231	73	10	342	425	2.9	460	24	24	24	372
polyglot	707	48	15	208	271	2.6	9	0	2	1	4
antlr	1211	87	13	453	553	2.2	77	7	4	3	28
bloat	3121	139	80	1655	1874	1.7	1298	80	91	80	148
jython	1576	83	64	415	562	2.8	458	11	29	30	167
ps	2676	236	73	1226	1535	1.7	667	17	41	189	49

threshold value to bound the total number of refinement passes (or nodes visited) — once the number of passes (or nodes visited) exceeds this value, the analysis gives up refinement and returns a safe approximate solution.

We inserted a guard in the points-to analysis code that returns immediately after the size of a points-to set becomes 1. If the points-to set contains multiple objects, the refinement continues until the maximum number of passes (10) or nodes (75000) is reached. Because the same constraint (i.e., #passes and #nodes) is used for both the original version and the optimized version, the optimized version does not lose any precision — in fact, it could have higher precision because it explores more paths (in our experiments, this affects only `ps`, in which two additional downcasts are proven to be safe).

The running time reduction is described in Table 2. Column *SB* shows the running time of the original version of the Sridharan-Bodik analysis. Columns *ISPG*, *Alias*, and *SB′* show the times to build the ISPG, run the must-not-alias analysis, and compute the points-to solution. Column *Speedup* shows the value of *Refine*/*Total*. On average, using the must-not-alias information provided by our analysis, the points-to analysis ran more than 3 times faster, and in some case the speedup was as large as five-fold.

The smallest performance improvement (1.6×) is for `sablecc-j`. We inspected the program and found the reason to be a large SCC (containing 2103 methods) in Spark's call graph. The fixed-point iteration merged a large number of symbolic/object nodes in the SCC methods, resulting in a large reachability map and limited filtering of un-aliased variables. In general, large SCCs (containing thousands of methods) are well known to degrade the precision and performance of context-sensitive analysis. Large cycles may sometimes be formed due to the existence of a very small number of spurious call graph edges. Based on this observation, we employ an optimization that uses the original (un-optimized)

version of the points-to analysis to compute precise points-to sets for the receiver variables at call sites that have too many call graph edges in Spark's call graph. This is done if the number of outgoing call graph edges at a call site exceeds a threshold value (e.g., the current value is 10). Hence, the approach pays the price of the increased ISPG construction time to reduce the cost and imprecision of the *memAlias* computation.

Note that our analysis does not impose heavy memory burden — once the must-not-alias analysis finishes and relation *memAlias* is computed, all reachability maps are released. The only additional needed memory is to hold the relation and the ISPG nodes. Both the reachability analysis and the subsequent points-to analysis ran successfully within the 2GB heap limit. We also performed experiments with 2-level-heap cloning (defined in Section 5). Due to space limitations, these results are not included in the paper. For some programs in the benchmark set, the analysis ran out of memory; for others, the *memAlias* computation became very slow. Thus, 1-level heap cloning appears to strike the right balance between cost and precision.

6.2 Analysis Precision

Column *Precision* in Table 2 shows a precision comparison between a points-to solution derived from relation *memAlias* (as described at the end of Section 4) and those computed by other analyses. The table gives the number of downcasts that can be proven safe by context-insensitive points-to analysis (*Ins*), our analysis (*mA*), object-sensitive analysis with 1-level heap cloning (*1H*), and the Sridharan-Bodik analysis (*SB*). From existing work, it is not surprising that *Ins* and *SB* have the lowest and highest precision, respectively. Analyses *1H* and *mA* have comparable precision. Although *mA* is fully context-sensitive for pointer variables, this does not have a significant effect on precision. The reason is that heap cloning is more important than context-sensitive treatment of pointer variables [24,11,6]. Even though *mA* can prove a much smaller number of safe casts than *SB*, it does prune out a large number of spurious aliasing relationships. For example, the points-to set of a variable computed by *mA* can be much smaller than the one computed by *Ins*. Thus, the analysis proposed in this paper could either be used as a pre-analysis for the Sridharan-Bodik points-to analysis (in which case it significantly reduces the overall cost without any loss of precision), or as a stand-alone analysis which trades some precision for higher efficiency (e.g., since it is significantly less expensive than *1H*).

7 Related Work

There is a very large body of work on precise and scalable points-to analysis [28,11]. The discussion in this section is restricted to the analysis algorithms that are most closely related to our technique.

CFL-reachability. Early work by Reps et al. [8,29,30,31,32] proposes to model realizable paths using a context-free language that treats method calls and re-

turns as pairs of balanced parentheses. Based on this framework, Sridharan et al. defined a CFL-reachability formulation to precisely model heap accesses, which results in demand-driven points-to analyses for Java [16,6]. Combining the CFL-reachability formulations of both heap accesses and interprocedural realizable paths, [6] proposes a context-sensitive analysis that achieves high precision by continuously refining points-to relationships. The analysis is the most precise one among a set of context-sensitive, field-sensitive, subset-based points-to analysis algorithms, and can therefore satisfy the need of highly-precise points-to information. However, the high cost associated with this precision is an obstacle for the practical real-world use of the analysis, which motivates our work on reducing the cost while maintaining the precision.

Zheng and Rugina [7] present a CFL-reachability formulation of alias analysis and implement a context-insensitive demand-driven analysis for C programs. The key insight is that aliasing information can be directly computed without having to compute points-to information first. Similarly to computing a points-to solution, this analysis also needs to make recursive queries regarding the aliasing relationships among variables. Hence, our pre-computed must-not-alias information could potentially be useful to improve the performance of this analysis.

Must-not-alias analysis. Naik and Aiken [33] present a conditional must-not-alias analysis and use it to prove that a Java program is free of data races. The analysis is conditional, because it is used to show that two objects can not alias under the assumption that two other objects can not alias. If it can be proven that any two memory locations protected by their respective locks must not alias as long as the two lock objects are distinct, the program cannot contain potential data races. Our analysis uses the must-not-alias relationship of two memory locations to disprove the existence of *alias* paths between two variables. These two analyses are related, because both use must-not-alias information to disprove the existence of certain properties (data race versus *alias* path).

Improving the scalability of points-to analysis. Similarly to our technique, there is body of work on scaling of points-to analysis. Rountev and Chandra [34] present a technique that detects equivalence classes of variables that have the same points-to set. The technique is performed before the points-to analysis starts and can speed up context-insensitive subset-based points-to analysis by a factor of two. Work from [35] observes that such equivalence classes still exist as points-to sets are propagated, and proposes an online approach to merge equivalent nodes to achieve efficiency. A number of other approaches have employed binary decision diagrams and Datalog-based techniques (e.g., [36,37,38,39,40]) to achieve high performance and precision. Our previous work [12] identifies equivalence classes of calling contexts and proposes merging of equivalent contexts. This analysis has strong context sensitivity — it builds a symbolic points-to graph for each individual method, and clones all non-escaping SPG nodes from callees to callers. Although the technique proposed in this paper uses a variation of this ISPG as program representation, it is fundamentally different from this previous analysis. In this older work, which is not based on CFL-reachability, both variable and field points-to edges are cloned

to compute a complete points-to solution. The technique in this paper formulates a new alias analysis as a CFL-reachability problem, uses a completely different analysis algorithm, employs a different form of context sensitivity, and aims to reduce the cost of a subsequent points-to analysis that is based on a more general and expensive CFL-reachability algorithm.

8 Conclusions

The high precision provided by CFL-reachability-based pointer analysis usually comes with great cost. If relatively precise alias information is available at the time heap loads and stores are matched, many irrelevant paths can be eliminated early during the computation of CFL-reachability. Based on this observation, this paper proposes a must-not-alias analysis that operates on the ISPG of the program and efficiently produces context-sensitive aliasing information. This information is then used in the Sridharan-Bodik points-to analysis. An experimental evaluation shows that the points-to analysis is able to run 3× faster without any precision loss. This technique is orthogonal to existing CFL-reachability-based points-to analysis algorithms — it does not aim to compute precise points-to information directly, but rather uses easily computed aliasing information to help the points-to analyses quickly produce a highly-precise solution. These scalability results could directly benefit a large number of program analyses and transformations that require high-quality points-to information at a practical cost, for use on large programs in real-world software tools.

Acknowledgments. We would like to thank the ECOOP reviewers for their valuable and thorough comments and suggestions. This research was supported in part by the National Science Foundation under CAREER grant CCF-0546040.

References

1. Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 57–68 (2002)
2. Fink, S., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. In: ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 133–144 (2006)
3. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 308–319 (2006)
4. Voung, J.W., Jhala, R., Lerner, S.: RELAY: Static race detection on millions of lines of code. In: ACM SIGSOFT International Symposium on the Foundations of Software Engineering, pp. 205–214 (2007)
5. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems 12(1), 26–60 (1990)
6. Sridharan, M., Bodik, R.: Refinement-based context-sensitive points-to analysis for Java. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 387–400 (2006)

7. Zheng, X., Rugina, R.: Demand-driven alias analysis for C. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 197–208 (2008)
8. Reps, T.: Program analysis via graph reachability. Information and Software Technology 40(11-12), 701–726 (1998)
9. Sridharan, M. (2006), http://www.sable.mcgill.ca/pipermail/soot-list/2006-January/000477.html
10. Kahlon, V.: Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 249–259 (2008)
11. Lhoták, O., Hendren, L.: Context-sensitive points-to analysis: Is it worth it? In: International Conference on Compiler Construction, pp. 47–64 (2006)
12. Xu, G., Rountev, A.: Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In: ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 225–235 (2008)
13. Kodumal, J., Aiken, A.: The set constraint/CFL reachability connection in practice. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 207–218 (2004)
14. Soot Framework, http://www.sable.mcgill.ca/soot
15. Vallée-Rai, R., Gagnon, E., Hendren, L., Lam, P., Pominville, P., Sundaresan, V.: Optimizing Java bytecode using the Soot framework: Is it feasible? In: International Conference on Compiler Construction, pp. 18–34 (2000)
16. Sridharan, M., Gopan, D., Shan, L., Bodik, R.: Demand-driven points-to analysis for Java. In: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 59–76 (2005)
17. Chatterjee, R., Ryder, B.G., Landi, W.: Relevant context inference. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 133–146 (1999)
18. Wilson, R., Lam, M.: Efficient context-sensitive pointer analysis for C programs. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 1–12 (1995)
19. Cheng, B., Hwu, W.: Modular interprocedural pointer analysis using access paths. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 57–69 (2000)
20. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for Java programs. In: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 187–206 (1999)
21. Melski, D., Reps, T.: Interconvertibility of a class of set constraints and context-free-language reachability. Theoretical Computer Science 248, 29–98 (2000)
22. Rehof, J., Fähndrich, M.: Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 54–66 (2001)
23. Kodumal, J., Aiken, A.: Regularly annotated set constraints. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 331–341 (2007)
24. Nystrom, E., Kim, H., Hwu, W.: Importance of heap specialization in pointer analysis. In: PASTE, pp. 43–48 (2004)
25. Lattner, C., Lenharth, A., Adve, V.: Making context-sensitive points-to analysis with heap cloning practical for the real world. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 278–289 (2007)

26. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Muchnick, S., Jones, N. (eds.) Program Flow Analysis: Theory and Applications, pp. 189–234. Prentice-Hall, Englewood Cliffs (1981)
27. Lhoták, O., Hendren, L.: Scaling java points-to analysis using SPARK. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 153–169. Springer, Heidelberg (2003)
28. Hind, M.: Pointer analysis: Haven't we solved this problem yet? In: PASTE, pp. 54–61 (2001)
29. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 49–61 (1995)
30. Horwitz, S., Reps, T., Sagiv, M.: Demand interprocedural dataflow analysis. In: ACM SIGSOFT International Symposium on the Foundations of Software Engineering, pp. 104–115 (1995)
31. Reps, T.: Solving demand versions of interprocedural analysis problems. In: Fritzson, P.A. (ed.) CC 1994. LNCS, vol. 786, pp. 389–403. Springer, Heidelberg (1994)
32. Reps, T., Horwitz, S., Sagiv, M., Rosay, G.: Speeding up slicing. In: ACM SIGSOFT International Symposium on the Foundations of Software Engineering, pp. 11–20 (1994)
33. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 327–338 (2007)
34. Rountev, A., Chandra, S.: Off-line variable substitution for scaling points-to analysis. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 47–56 (2000)
35. Hardekopf, B., Lin, C.: The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 290–299 (2007)
36. Berndl, M., Lhoták, O., Qian, F., Hendren, L., Umanee, N.: Points-to analysis using BDDs. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 103–114 (2003)
37. Lhoták, O., Hendren, L.: Jedd: A BDD-based relational extension of Java. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 158–169 (2004)
38. Whaley, J., Lam, M.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 131–144 (2004)
39. Zhu, J., Calman, S.: Symbolic pointer analysis revisited. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 145–157 (2004)
40. Bravenboer, M., Smaragdakis, Y.: Doop framework for Java pointer analysis (2009), `doop.program-analysis.org`