Relevant Context Inference*

Ramkrishna Chatterjee† **Barbara G. Ryder**† **William A. Landi**‡

†Department of Computer Science
Rutgers, The State University of NJ
110 Frelinghuysen Road
Piscataway NJ 08854-8019, USA
+1 732 445 2001
{ramkrish,ryder}@cs.rutgers.edu

‡Siemens Corporate Research, Inc.
755 College Road East
Princeton NJ 08540, USA
+1 609 734 6500
landi@scr.siemens.com

Abstract

Relevant context inference (**RCI**) is a modular technique for flow- and context-sensitive data-flow analysis of statically typed object-oriented programming languages such as C++ and Java. *RCI* can be used to analyze complete programs as well as incomplete programs such as libraries; this approach does not require that the entire program be memory-resident during the analysis. *RCI* is presented in the context of *points-to analysis* for a realistic subset of C++. The empirical evidence obtained from a prototype implementation argues the effectiveness of *RCI*.

1 Introduction

Points-to analysis [EGH94] for statically typed object-oriented programming languages (e.g., Java, C++) determines, at each program point, the objects to which a pointer may point during execution. This information is crucial to many applications, including static resolution of dynamically dispatched calls, side-effect analysis, data-flow-based testing, program slicing and aggressive compiler optimizations. The solution of *concrete type inference* [PC94], necessary for object-oriented optimizations such as method specialization and inlining, is subsumed by the solution of points-to analysis, because the concrete type of a pointer is the set of classes corresponding to the objects possibly pointed to by that pointer.

The goal of our analysis is to preserve precision as much as possible, without sacrificing scalability. Flow and context sensitivity affect both the precision and cost of analyses. A *flow-insensitive* algorithm ignores the ordering of statements within a method; by contrast, a *flow-sensitive* algorithm follows the control flow order of statements within a method, and computes different solutions for a variable at distinct program points. A *context-sensitive* algorithm considers (sometimes approximately) only interprocedurally *realizable paths* [SP81, LR91, EGH94, RHS95]: paths along which calls and returns are properly matched, while a *context-insensitive* algorithm does not make this distinction.

Existing algorithms for points-to analysis vary in their flow and context sensitivity; they compute approximate solutions which are supersets of the precise points-to solution. The least expensive, but most imprecise, are the flow- and context-insensitive approaches [Wei80, And94, Ste96, ZRL96, SH97]. In contrast, the flow- and context-sensitive techniques [LR92, CBC93, MLR+93, EGH94, Deu94, CHS95, WL95, Ruf95, PR96] are the most precise but also the most expensive (in time and memory). Although most of these have been used for alias or points-to analysis of C, they can be adapted for points-to analysis of C++without exceptions.

The precision of the solution computed for points-to analysis directly affects its utility in applications. However, flow- and context-sensitive points-to analysis of C++ is difficult, due to dynamic dispatch, objects containing pointers to subobjects (i.e., recursive types), and the many invocation contexts per method. In addition, flow- and context-sensitive algorithms are memory intensive; they frequently run out of memory while analyzing even moderately sized programs. These difficulties were addressed in the design (and implementation) of *RCI*, a modular technique explained here as applied to points-to analysis for a subset of C++. By the term *modular*, we mean a technique for whole program analysis which never requires the entire program source to be in memory at any one time. Since much object-oriented code is written as libraries, a goal of *RCI* is to analyze incomplete programs. An additional goal for *RCI* is that it maintain a sufficient degree of precision in its data-flow information for intended applications.

Intuitively, we analyze each method assuming unknown initial values for parameters and globals at method entry. The key insight is to obtain a summary function for the data-flow effect of method execution by bottom-up inference of the *relevant* conditions on the unknown initial values. These conditions capture only the relevant contexts for a method, making this approach feasible, although the summary function can be used in any context. Previous techniques [LR92, Ema93, Ghi96, WL95], have incorporated memoization of the data-flow solution associated with a particular calling context (or set of contexts).

In our approach, method bodies are analyzed first separately from calling context information. The effects of a method on points-to analysis information is calculated, sometimes dependent on certain conditions on the incoming unknown initial values for parameters and globals. Rather than calculate all possible conditions, the algorithm calculates only those conditions which may affect points-to infor-

*The research reported here was supported, in part, by NSF grant CCR-9501761, the Hewlett-Packard Corporation and Edison Design Group.

mation, by inferring them from the code of the method and those other methods it may invoke directly or indirectly during its lifetime.[1] Care is taken to observe those object fields actually *used* by this method directly or indirectly through calls; conditions are inferred for only those fields which are *used* in this sense. The results of these calculations are two-fold: (i) a points-to solution at each node of the method and (ii) a summary transfer function for the method, a function expressing method invocation effects on the points-to solution; both are parametrized by the unknown initial values and the conditions on these values. Summary functions for callees are calculated before those of their callers. Then, points-to information is propagated into a method from its callers, with actual-parameter bindings accounted for. Recursion requires simultaneous handling of calls in the same strongly connected component (SCC) of the program call graph. Finally, the required points-to solution at a statement is computed on demand by instantiating the unknown initial values using the points-to information at a method entry.

This entire calculation is carefully staged, so that the entire program source need not be in memory at any one time, because the calculation can be done separately on the SCC's of the program call graph.

Our algorithm is the first *modular*, flow- and context-sensitive algorithm for points-to analysis of programs written in a realistic subset of C^{++}. Moreover, although in this paper we explain *RCI* in the context of points-to analysis of C^{++} programs, the technique can be extended to handle libraries, Java programs with exceptions (but without threads), and to solve other data-flow analysis problems.

The main results in this paper are:

- An algorithm for modular points-to analysis of programs written in a realistic subset of C^{++}.
- Empirical evidence for the effectiveness of *RCI* as applied to points-to analysis of C^{++}.

Further details on this research are given in [CRL98b], including extensions to handle Java with exceptions and to analyze libraries with applications to testing.

2 Definitions

This section presents many technical definitions needed to explain the algorithm and also delimits the subset of C^{++} which is handled.

C^{++} Subset. Given the limited space available, we define a significantly restricted subset of C^{++}. In this paper, we describe our algorithm and provide analysis (e.g., complexity results) on this simple subset. This allows us to simplify the presentation while demonstrating the interesting parts of our algorithm. The subset is defined in Figure 1[2]. It includes single inheritance, dynamic dispatch, recursive types and pointer assignment statements with a single level of dereferencing.[3] It excludes multiple inheritance, an explicit address operator (i.e., pointers to the stack), reference type, function pointers, data members of structure types (note this does not exclude data members of pointers to structure types), general pointer assignment statements, arrays (array elements are mapped to a single representative element) and exceptions. Many of these constructs can be easily accommodated by extending the algorithm; we will briefly indicate these extensions where relevant.

```
Program ⇒ {Class | Proc}+
Class ⇒ class ClassName [: public ClassName]
  { {DataMember| Method}+ }
DataMember ⇒ Protection [static] Type FieldName ;
Type ⇒ ClassName * | PrimitiveType
PrimitiveType ⇒ int | char | float | bool
Method ⇒ Protection [static | virtual] [void | Type]
  MethodName ( [Param RestParam*] ) { Body }
Param ⇒ Type VarName
RestParam ⇒ , Param
Proc ⇒ {void | Type} ProcName ( [Param RestParam*] ) { Body }
Body ⇒ Decls Stmnt+
Decls ⇒ Decl*
Decl ⇒ Type VarName ;
Stmnt ⇒ AssignmentStmnt | NewStmnt | Call | If |
  While| ReturnStmnt | ;
ReturnStmnt ⇒ return VarName ;
AssignmentStmnt ⇒ Lhs = Rhs ;
Lhs ⇒ VarName | VarName→FieldName
Rhs ⇒ VarName | VarName→FieldName | 0 | Expr
Call ⇒ [VarName = ]
  {VarName→MethodName|MethodName
  |ClassName::MethodName|ProcName}
  ( [VarName RestVar*] ) ;
NewStmnt ⇒ VarName = new ClassName ( [VarName RestVar*] ) ;
RestVar ⇒ , VarName
If ⇒ if ( Expr ) {Stmnt+ } [else { Stmnt+ } ]
While ⇒ while ( Expr ) { Stmnt+ }
VarName ⇒ Name
FieldName ⇒ Name
ProcName ⇒ Name
MethodName ⇒ Name
ClassName ⇒ Name
Protection ⇒ public : | protected : | private :
```

Figure 1: Simpler subset of C^{++}

If the algorithm is understood fully for the simple subset, then handling of most of C^{++} requires handling of details but not any changes to the fundamental ideas of the algorithm. The subset of C^{++} that can easily be handled by the modular points-to analysis algorithm excludes arbitrary casting, uninstantiated templates, pointers to data members and pointers to member methods (which are different from ordinary function pointers). Up-cast of a derived class to a base class and down-cast of a base class to a derived class can be handled. Our implementation of the algorithm is consistent with this bigger subset, only excluding in addition, multiple inheritance and exceptions.

Precision and Safety. At each program point, points-to analysis calculates the set of objects to which a pointer may point during some execution. Any static analysis technique for points-to analysis needs to represent a potentially infinite number of heap-allocated, run-time objects with a finite number of names. Following [LR92, CBC93, MLR$^+$93, WL95, Ste96], we represent all the run-time objects created

[1]This approach is analogous to the way pointer aliases are calculated in [LR92]. The possible reaching aliases (RA's) used in the aliasing algorithm are those which actually are propagated to procedure entry, rather than all those aliases which possibly might be propagated to the entry. Likewise, *RCI* solves for conditions that potentially affect the data-flow solution, rather than for all conditions.

[2]*Expr* is any side-effect-free expression that does not have any function call and that can be ignored for points-to analysis. *[pattern]* means at most one occurrence of the *pattern*. $\{pattern\}^+$ means one or more occurrences of the *pattern*. $\{pattern\}^*$ means zero or more occurrences of the *pattern*. $\{a \mid b\}$ means a or b. The terminal symbols are underlined.

[3]lhs = rhs, where lhs is p or p→f1 and rhs is q or q→f2.

at a program point n with the single name $object_n$, a **heap-name**. A pair $\langle ptr, object_n \rangle$ belongs to the *precise* solution of points-to analysis at a program point m if and only if there exists an execution path from the *start* node of the program to m, such that when this path is executed, at m the pointer *ptr* points to an object created at program point n. A **safe** solution of points-to analysis is one that is a superset of the *precise* solution. Our modular points-to analysis algorithm calculates a safe solution.

Data Representations. *RCI* stores the data-flow facts together with corresponding alias and type context conditions (inferred by the analysis) in data-flow elements or **dfelms**. The initial phase of the algorithm examines each method, presuming unknown initial values for each of its parameters and any global variables. Depending on the pointer assignments in that method, these unknown initial values may appear as variable names in the points-to relations and/or in corresponding conditions associated with such relations.

We use var_{init} to represent the unknown initial value for the global or parameter *var*. Note that var_{init} denotes the unknown initial object to which *var* points and not the address of that object. $var_{init}.next_{init}$ represents the unknown initial value to which $var \rightarrow next$ points. $var_{init}.next_{init}.next_{init}$ represents the unknown initial value to which $var \rightarrow next \rightarrow next$ points. Obviously, in the presence of recursive types, the number of unknown initial values accessed in a method could be unbounded. To overcome this problem, unknown initial values are mapped into a finite number of sets. All the elements in a set are represented by a single, representative name. *RCI* uses patterns in the *access paths* [Deu94] of unknown initial values to form these sets[CRL98b].

For points-to analysis, each *dfelm* propagated by *RCI* has the form:

$\langle$*relevant context, points-to*$\rangle$,

where *points-to* represents a pair of the form:

$\langle$*var, object*$\rangle$.

var is a local pointer variable, a global pointer variable, an unknown initial value's field of pointer type or a heap-name's field of pointer type; and *object* is an unknown initial value or heap-name. *object* can also be *null*, which is treated as a special heap-name.

A *relevant context* has the form:

$\langle$*alias context, type context*$\rangle$.

An *alias context* is *empty* or it is a conjunction of potential aliases and potential non-aliases between unknown initial values. Each **potential alias** has the form:

$(uiv_1$ **eq** $uiv_2)$

and each **potential non-alias** has the form:

$(uiv_1$ **neq** $uiv_2)$.

Here uiv_1 and uiv_2 are unknown initial values. These potential aliases and potential non-aliases are inferred by the algorithm during analysis; some data-flow facts are dependent on specific conditions of this sort.

A *type context* is a conjunction of type constraints or *empty*. Each type constraint has the form:

$(uv.A::x)$,

where *uv* is an unknown initial value, A is a class and x is a dynamically dispatched method defined in A (a virtual method in C^{++}). $A::x$ represents the set of classes containing A and all the subtypes of A for which a virtual invocation of method x will be resolved to the definition of method x in class A, $A::x$. The constraint means that the associated *dfelm* is valid only in those contexts in which the concrete, run-time type of *uv* (not the declared type) belongs to $A::x$.

As with reaching aliases in our earlier work on flow-sensitive aliasing[LR92], the relevant context of a *dfelm* provides a good approximation for a calling context. This allows data-flow information to be propagated along good approximations of realizable paths.

3 Modular Points-to Analysis

In this section, we explain *RCI* as applied to points-to analysis.

3.1 Four Phases of *RCI*

RCI is an iterative worklist algorithm that is flow- and context-sensitive. *RCI* takes as input a statement-level interprocedural control flow graph or **ICFG** [LR92]. From this, an initial approximate call graph is formed and decomposed into SCC's. The following phases are performed using the SCC condensation (**SCC-DAG**).

- **Phase 0:** In this phase, *RCI* constructs a safe overestimate of the call graph called the **initial call graph** by resolving dynamically dispatched calls using hierarchy analysis [DMM96].[4] Then *RCI* uses a linear-time algorithm[CLR92] to construct the SCC-DAG of the initial call graph. Note that the initial call graph need not be precise, it only needs to be a safe overestimate; the precision of any safe initial call graph only affects the *efficiency* of *RCI*, and not the *safety* of the computed solution. The initial call graph can be made more precise (e.g., by using [BS96]); however, in practice we have found hierarchy analysis to be adequate.
- **Phase I:** *RCI* traverses the SCC-DAG in a reverse topological order (bottom-up) and analyzes each method assuming parameters and global variables have unknown initial values. For each method *RCI* computes, in terms of unknown initial values, a safe approximation to the method's complete transfer function for pointers. We will call this approximation, the **summary transfer function**. The summary transfer function of a method *M* is the set of *dfelms* that reach the exit node of *M* and that do not represent values of local variables of *M*. This function summarizes the possible effects of method invocation on *dfelms*. The summary transfer functions of methods in the same SCC have cyclic dependences, so they are computed simultaneously by fixed point iteration. In contrast, the summary transfer functions of methods in different SCC's have hierarchical dependences (or no dependence at all), and hence are computed by bottom-up traversal on SCC-DAG, without iteration. After Phase I, the summary transfer function of each method and the points-to solution at each node in the method are expressed in terms of the *dfelms*, defined in Section 2, which may contain unknown initial values.

[4] In the initial call graph, function pointer call targets can be approximated by those functions whose addresses have been stored and whose signatures match with the type of the called function.

- **Phase II:** *RCI* traverses the SCC-DAG in a topological order (top-down) and propagates *concrete values* of unknown initial values to the *entry* nodes of methods. This phase involves only the entry nodes and call nodes, and as our empirical results show, it is extremely fast. *RCI* considers only *reachable* methods during this phase.
- **Phase III:** This phase involves only nodes that are not entry nodes. In this phase, the unknown initial values in the *dfelms* computed during Phase I are instantiated by their concrete values computed in Phase II. This phase is completely *demand-driven*, and needs to be performed only at those nodes where the final solution is needed. After this phase, the points-to solution at each node is expressed entirely in terms of program variables and heap-names.

During the construction of the initial call graph using hierarchy analysis, each method needs to be in memory once. After this, each node of SCC-DAG (and hence each method) needs to be in memory only three more times, once during each of the phases I, II and III. The rest of the time, only a method's summary transfer function or the Phase II solution at the entry node of a method needs to be in memory. Hence, this is a *modular* approach and requires less memory than other whole-program-analysis techniques, in which a method cannot be moved out of memory without the possibility of it being needed again, until the final solution is computed. In these techniques, if the whole program is not kept in memory, there is no *a priori* constant bound on the number of times a method needs to be moved into or out of memory.

Of course, in the worst case, the entire initial call graph may be a single SCC and *RCI* may need to keep the whole program in memory. However, our empirical results show that SCC's are quite small in practice and *RCI* is able to analyze almost a method at a time. In some very specific domains (e.g., recursive descent parsing), SCC's may be occasionally large, however, even in these cases the entire initial call graph is unlikely to be a single SCC. For example, in parsing, the SCC's of methods dealing with statements are likely to be different from the SCC's of methods dealing with types.

3.2 Phase I

In Phase I, *RCI* analyzes each method using the unknown initial values of parameters and global variables. By analyzing a method, it infers the relevant potential aliasing between unknown initial values and their relevant potential concrete (not declared) types. *RCI* computes a *dfelm* conditioned on such potential aliases (i.e., *dfelm*'s alias context) and concrete types (i.e., *dfelm*'s type context).

Propagation in this phase occurs in reverse topological order on the SCC-DAG; the objective is to calculate the points-to solution at each node in terms of unknown initial values and to calculate a summary transfer function for each method in the program, so it can be used to compute the summary transfer function of the callers of the method. Intuitively, a *dfelm* d at the exit node of a method M is valid at a call site that invokes M if and only if all the conjuncts of the relevant context of d are *true* at the call site. In general, at a call site that invokes M, one of the following three things happens for a conjunct of the relevant context of d: the conjunct evaluates to *true*, it evaluates to *false*, or it is translated into a similar conjunct involving the unknown initial values of the caller.

For propagation of *dfelms* between methods in different SCC's, no iteration is necessary. Using actual-parameter bindings, the summary transfer function of the called method can be used to calculate the *dfelms* returned from the call.

However, to obtain the proper propagation of *dfelms* through a non-trivial SCC, it is necessary to propagate *dfelms* on the graph of the SCC itself. Since there are cyclic dependences, iteration must be performed until a fixed point is reached. During this iteration, only a partial summary transfer function may be available at a method's exit node; this is used when processing same-SCC callers of this method. Whenever a new *dfelm* is added to the partial summary transfer function of a method, the same-SCC callers of this method are informed about this *dfelm*, so that corresponding call sites can process this new *dfelm*.

The calculation of the points-to solution in terms of unknown initial values and the calculation of the summary transfer function of a method are accomplished by the propagation of *dfelms* from method entry to exit using a worklist algorithm. Initially the solution at each node is empty; it grows monotonically as new *dfelms* are added. The code of the method is represented in the ICFG. Points-to information which reaches an ICFG node serves as input to its node transfer function which embodies the data-flow effect of the semantics of code corresponding to that node. We will specify the statement transfer functions for points-to analysis at pointer assignments and call statements. In the next section, we present examples showing these transfer functions and how relevant contexts are manipulated; we give pseudocode for Phase I in Appendix A.

3.2.1 Examples of *dfelm* Propagation

Figure 2 shows the final Phase I solution computed by *RCI* at the top of each of the statements 1, 2, 3 and 4 (i.e., program points 1,2,3,4). The *dfelms* at program point 1 say that all the variables have the same values as at the entry node of *method1*. Since statement 1 assigns the value of *a3* to the *next* field of the value of *a1*, **2:1**, the first *dfelm* at program point 2, says that the *next* field of the unknown initial value of *a1* points to the unknown initial value of *a3*. The relevant context $\langle$*empty, empty*$\rangle$ means that both alias context and type context are *empty*, and therefore this *dfelm* is valid in all contexts.

At an assignment statement, for each set s of *dfelms* that gives the values of the left and right hand sides, the relevant context of the corresponding *dfelm* (implied by s) resulting from the assignment is the conjunction of the relevant contexts of the *dfelms* of s. For statement 1, there is only one such s which consists of **1:1** and **1:3**, and the conjunction of the relevant contexts of the *dfelms* of s is $\langle$*empty, empty*$\rangle$. If the unknown initial values of *a1* and *a2* are the same, statement 1 also modifies the *next* field of the unknown initial value of *a2*. The *dfelms* **2:2** and **2:3** keep track of this potential modification. **2:2** is applicable to only those contexts in which $a1_{init}$ and $a2_{init}$ are equal, while **2:3** is applicable to only those contexts in which $a1_{init}$ and $a2_{init}$ are not equal. The *dfelms* **3:1** and **3:2** are implied respectively by **2:2** and **2:3**.

RCI does not consider $a3_{init}.next$ for potential modification at statement 1 because it is not used in *method1*. *RCI* generates fields of unknown initial values *lazily* (as explained in Section 3.2.3), only inferring *relevant* potential aliases and potential non-aliases. For example, the relationships of $a3_{init}$ with $a1_{init}$ and $a2_{init}$ are not relevant

```
class A {
  public:  A* next; public:  A() { next = 0; };
  public:  virtual A* choose( A* a1, A* a2 ) {
    return a1;
  };
};
class B: public A {
};
class C: public A {
  public:  A* choose( A* a1, A* a2 ) {
    return a2;
  };
};
class test {
  public:  static A* global1;
  public:  static A* global2;
  public:  static void method1( A* a1, A* a2, A* a3 ) {
    1:  a1 → next = a3;
    2:  global1 = a2 → next;
    3:  global2 = a1 → choose( a1, a2 );
    4:
  };
  ...
};
```

1:1 $\langle\langle \mathit{empty},\ \mathit{empty}\rangle,\ \langle \mathit{a1},\ a1_{init}\rangle\rangle$
1:2 $\langle\langle \mathit{empty},\ \mathit{empty}\rangle,\ \langle \mathit{a2},\ a2_{init}\rangle\rangle$
1:3 $\langle\langle \mathit{empty},\ \mathit{empty}\rangle,\ \langle \mathit{a3},\ a3_{init}\rangle\rangle$
1:4 $\langle\langle \mathit{empty},\ \mathit{empty}\rangle,\ \langle \mathit{global1},\ \mathit{global}1_{init}\rangle\rangle$
1:5 $\langle\langle \mathit{empty},\ \mathit{empty}\rangle,\ \langle \mathit{global2},\ \mathit{global}2_{init}\rangle\rangle$
1:6 $\langle\langle \mathit{empty},\ \mathit{empty}\rangle,\ \langle a1_{init}.\mathit{next},\ a1_{init}.\mathit{next}_{init}\rangle\rangle$
1:7 $\langle\langle \mathit{empty},\ \mathit{empty}\rangle,\ \langle a2_{init}.\mathit{next},\ a2_{init}.\mathit{next}_{init}\rangle\rangle$

2:1 $\langle\langle \mathit{empty},\ \mathit{empty}\rangle,\ \langle a1_{init}.\mathit{next},\ a3_{init}\rangle\rangle$
2:2 $\langle\langle (a1_{init}\ \mathbf{eq}\ a2_{init}),\mathit{empty}\rangle,\ \langle a2_{init}.\mathit{next},\ a3_{init}\rangle\rangle$
2:3 $\langle\langle (a1_{init}\ \mathbf{neq}\ a2_{init}),\mathit{empty}\rangle,\ \langle a2_{init}.\mathit{next},\ a2_{init}.\mathit{next}_{init}\rangle\rangle$
2: + {**1:1 to 1:5**}

3:1 $\langle\langle (a1_{init}\ \mathbf{eq}\ a2_{init}),\mathit{empty}\rangle,\ \langle \mathit{global1},\ a3_{init}\rangle\rangle$
3:2 $\langle\langle (a1_{init}\ \mathbf{neq}\ a2_{init}),\mathit{empty}\rangle,\ \langle \mathit{global1},\ a2_{init}.\mathit{next}_{init}\rangle\rangle$
3: + {**1:1 to 1:3,1:5, 2:1 to 2:3** }

4:1 $\langle\langle \mathit{empty},(a1_{init}.\mathit{A::choose})\rangle,\ \langle \mathit{global2},\ a1_{init}\rangle\rangle$
4:2 $\langle\langle \mathit{empty},(a1_{init}.\mathit{C::choose})\rangle,\ \langle \mathit{global2},\ a2_{init}\rangle\rangle$
4: + {**1:1 to 1:3,2:1 to 2:3, 3:1, 3:2** }

Figure 2: Phase I Solution

to *method1* because first, $a3_{init}.next$ is not used in *method1* and second, $a2_{init}.next$ is not directly modified in *method1*.

Statement 3 is a dynamically dispatched call site. It can invoke either *A::choose* or *C::choose* depending upon the concrete type of $a1_{init}$. As a result, *RCI* computes the values of *global2* at program point 4 conditioned on the potential concrete types of $a1_{init}$. The *dfelm* **4:1** says that *global2* points to $a1_{init}$ in those contexts in which the concrete type of $a1_{init}$ belongs to the set of types represented by *A::choose* (i.e., $\{A,B\}$). The meaning of **4:2** is similar. As before with aliases, *RCI* only infers relevant potential concrete types; for example, the concrete types of $a2_{init}$ and $a3_{init}$ are not relevant to *method1*.[5] *RCI* does not kill any *dfelm* at a call site unless the *dfelm* represents a value of the variable in which the result of the call is stored, so the *dfelms* {**1:1** to **1:3**, **2:1** to **2:3**, **3:1**, **3:2**} go across the call in statement 3.

The summary transfer function of *method1* consists of the *dfelms* at program point 4 except **1:1**, **1:2** and **1:3**, as these three *dfelms* represent values of variables local to *method1*.

Now consider the call sites that invoke *method1* from another *static* method *method2* of *test*.

```
static void method2( A* a4, A* a5, A* a6 ) {
  A *p, *q;
  5:  method1(a4,a5,a6);
  6:  p = new B(); 7:  q = new C();
  8:  method1(p,q,q);
  9:  }
```

At call site 5, *RCI* translates the *dfelms* at the exit node of *method1* by replacing the unknown initial values of *a1*, *a2* and *a3* with their values at call site 5 (i.e., the unknown initial values of *a4*, *a5* and *a6* respectively). Note: this is propagating data-flow information from callee back to caller. For example:

- $\langle\langle (a1_{init}\ \mathbf{eq}\ a2_{init}),\mathit{empty}\rangle,\ \langle \mathit{global1},\ a3_{init}\rangle\rangle$
  is translated to
  $\langle\langle (a4_{init}\ \mathbf{eq}\ a5_{init}),\mathit{empty}\rangle,\ \langle \mathit{global1},\ a6_{init}\rangle\rangle$.

- $\langle\langle \mathit{empty},(a1_{init}.\mathit{C::choose})\rangle,\ \langle \mathit{global2},\ a2_{init}\rangle\rangle$
  is translated to
  $\langle\langle \mathit{empty},(a4_{init}.\mathit{C::choose})\rangle,\ \langle \mathit{global2},\ a5_{init}\rangle\rangle$.

At call site 8, *RCI* evaluates the relevant contexts of the *dfelms* at the exit node of *method1* using $object_6$ as $a1_{init}$ and $object_7$ as $a2_{init}$ and $a3_{init}$. For example:

- $\langle\langle (a1_{init}\ \mathbf{eq}\ a2_{init}),\mathit{empty}\rangle,\ \langle \mathit{global1},\ a3_{init}\rangle\rangle$
  is not applicable to this call site as
  ($object_6$ **eq** $object_7$) is false.

- $\langle\langle (a1_{init}\ \mathbf{neq}\ a2_{init}),\mathit{empty}\rangle,\ \langle \mathit{global1},\ a2_{init}.\mathit{next}_{init}\rangle\rangle$
  is translated to
  $\langle\langle \mathit{empty},\ \mathit{empty}\rangle,\ \langle \mathit{global1},\ \mathit{null}\rangle\rangle$
  because ($object_6$ **neq** $object_7$) is *true* and the value of $object_7.next$ is *null* at program point 8.

- $\langle\langle \mathit{empty},(a1_{init}.\mathit{A::choose})\rangle,\ \langle \mathit{global2},\ a1_{init}\rangle\rangle$
  is translated to
  $\langle\langle \mathit{empty},\ \mathit{empty}\rangle,\ \langle \mathit{global2},\ object_6\rangle\rangle$
  because B, the concrete type of $a1_{init}$, belongs to A::choose, i.e., {A,B}.

To summarize, at each call site *RCI* stores the bindings between the actuals and the unknown initial values of the methods invocable from the call site and the relevant contexts of the *dfelms* that imply these bindings. At call sites 5 and 8, these relevant contexts happen to be $\langle \mathit{empty},\ \mathit{empty}\rangle$. When the actual-to-uiv bindings are used for replacing unknown initial values with actuals in a *dfelm d1*, each resulting *dfelm d2* is associated with a new relevant context. This context is the conjunction of the contexts of the bindings used in generating *d2* and the context of *d1*, instantiated with the actuals.

Although the example in Figure 2 has recursive types, only a finite number of unknown initial values are accessed in it, so representative names are not needed.

[5] In C^{++}, a call through an unknown initial value of a function pointer can be handled similarly by conditioning on the relationships between the unknown initial value (of the function pointer) and the functions that can be potentially invoked, that is, functions whose addresses have been stored in a function pointer and whose signatures match with the type of the call. For libraries, we assume that no function external to the library is invoked through a function pointer in the library (see [CRL98b] for details).

3.2.2 Lazy Strong Update

An assignment to a local or global variable can be *safely* killed at a subsequent assignment to the same variable. In contrast, an assignment to a field of a heap-name cannot be killed (without computing additional information), because a heap-name may represent more than one run-time object. *RCI* sometimes is able to kill assignments to the fields of unknown initial values because for a particular call to a method, an unknown initial value represents the same run-time object throughout that method execution[WL95, CRL98b].

When a points-to set of a pointer is of cardinality one, then *may* points-to information is effectively *must* points-to information (remember we explicitly track *null*). This forms the basis on which our algorithm performs *kills* of *dfelms*. *RCI* performs kills for the fields of unknown initial values lazily. During Phase I, when a *dfelm* d representing the value of a field of an unknown initial value reaches a pointer assignment node n that can kill d according to the current points-to solution (i.e., the current solution implies that n *must* update that field of the unknown initial value), the decision for killing d cannot be made immediately. Until a fixed point is reached, only a partial solution is available at n and the *must* update could change to a *may* update according to a potentially larger fixed-point solution. In this situation, *RCI* does not propagate d immediately to the successors of n; instead, it marks n and d. After the worklist for a SCC becomes empty, *RCI* revisits the marked nodes to check if the marked *dfelms* are killed according to the current solution. If not, it unmarks the *dfelms* that should not be killed, and restarts iteration to propagate these unmarked *dfelms*. The propagation stops for a SCC when none of the remaining marked *dfelms* require repropagation (i.e., the marked nodes kill the marked *dfelms* according to the fixed-point solution). In practice, we have found this scheme to be quite effective.

This lazy propagation is efficient because the nodes that need to be revisited belong to the same SCC and their number is bounded by the size of the SCC. As shown in Section 5, the SCC's are usually small.

3.2.3 Optimizations of Phase I

There are several optimizations used in Phase I to minimize the amount of data-flow information which needs to be propagated by the algorithm, as this directly affects its cost.

Limiting relevant context. Let d be a *dfelm* at the exit node of a method M, C be a call site that invokes M and rc be the relevant context of d. If rc evaluates to *true* at C, any relevant context t that is contained in rc (i.e., the set of conjuncts of t is a subset of the set of conjuncts of rc) also evaluates to *true* at C. As a result, we have the following theorem:[6]

Theorem 1 *For any dfelm with relevant context r, it is safe to replace r with a relevant context s that is contained in r.*

Due to Theorem 1, instead of the complete relevant context, *RCI* can use any subset of these conjuncts without compromising *safety*, although this may cause propagation of spurious *dfelms* to call sites where only a part of the original relevant context is valid (i.e., we are using approximate context sensitivity). Many heuristics can be used for choosing the part of the complete relevant context that is stored. At present we use a simple heuristic: if the user specifies a bound of k on the number of conjuncts of a specific kind, we store the first k conjuncts of this kind associated with a *dfelm*; the rest of the conjuncts are dropped. This bound is imposed uniformly for all *dfelms*; however, *RCI* allows different bounds for different *dfelms*.

Reduction of alias context. Let *t1* and *t2* be two classes. If *t1* and *t2* satisfy any of the following: (i) *t1* is same as *t2*, (ii) *t1* is a subtype of *t2* or (iii) *t1* is a supertype of *t2*, then *t1* and *t2* are called *compatible*. Two unknown initial values are called *compatible* if and only if their declared classes are *compatible*. For the concrete values of two unknown initial values to be the same, the unknown initial values must be compatible.[7] Thus, only compatible unknown initial values can participate in a potential alias or a potential non-alias. Whenever a field of an unknown initial value is modified, the same field of another compatible unknown initial value can also be potentially modified, and for *safety*, *dfelms* with appropriate alias contexts need to be generated to record this potential modification. However, the following three optimizations enable *RCI* to avoid the generation of some of these *dfelms* without compromising *safety*.

Lazy Generation of Fields. *RCI* considers only those fields of an unknown initial value that are used in a method *M*, either directly at pointer assignment statements in *M* or indirectly, through actual-to-uiv bindings, at pointer assignment statements in methods invoked from *M* through a series of calls. For example, in Figure 2, the *next* field of $a3_{init}$ is not considered because it is not used in *method1*. When a field f of an unknown initial value is found to be used for the first time in *M*, this fact is propagated to the callers of *M* in the same SCC as *M*. If through actual-to-uiv bindings, access to f causes access to other fields for the first time in same-SCC callers, facts representing access to these fields are also propagated iteratively using a worklist. At an assignment statement, the fields that are considered for potential modification and generation of alias contexts are the fields that are used in the method. For example, the *next* field of $a3_{init}$ is not considered for potential modification at statement 1 of *method1*, although $a3_{init}$ has the same type as $a1_{init}$.

Write-only Fields. Consider the following method which is part of class *test* defined in Figure 2.

```
void method3( A* prm1, A* prm2, A* prm3 ) {
  1:  prm1 → next = prm3;
  2:  global1 = prm2 → next;
  3:  prm3 → next = global2; }
```

This method uses the *next* fields of the unknown initial values $prm1_{init}$, $prm2_{init}$ and $prm3_{init}$. Thus, the *next* fields of $prm2_{init}$ and $prm3_{init}$ should be considered for potential modification at statement 1. However, the *next* field of $prm3_{init}$ is only used for writing and it is never read in *method3*. As a result, at statement 1, it is unnecessary to generate a *dfelm* to represent the potential modification of the *next* field of $prm3_{init}$. This modification will be automatically seen at a call site of *method3* where the values of $prm1_{init}$ and $prm3_{init}$ are the same. On the other hand, consider a call site C that invokes *method3* and where it

[6] This is similar to the use of one reaching alias instead of a set of reaching aliases in [LR92].

[7] Multiple inheritance can be easily accommodated by extending the definition of *compatible* classes so that two classes are compatible if either one of the above three conditions hold or these two classes have a common derived class.

is possible for the values of $prm1_{init}$ and $prm3_{init}$ to be distinct unknown initial values. Suppose C is contained in method M. Further, let *uv1* and *uv2* be two distinct unknown initial values that are respectively values of $prm1_{init}$ and $prm3_{init}$ at C. If the *next* field of *uv2* is read in M, RCI will generate a *dfelm* with appropriate alias context in M to record the potential modification to the *next* field of *uv2* due to the modification of the *next* field of *uv1* at the statement 1 of *method3*.

Initially, RCI considers all the fields of unknown initial values to be *write-only*. When a field is found to be read for the first time, it is made *read/write* and a candidate for potential modification; thereafter, it will be considered for the generation of alias context.

Restricting alias context to unknown initial values. At a pointer-assignment statement, RCI only considers the fields of unknown initial values (not heap-names) for potential modification and the generation of alias context. This is safe because for any particular call to a method, the run-time objects represented by a heap-name in the method during Phase I are different from the run-time objects represented by any of the unknown initial values. Although a heap-name appearing in a method during Phase I could be associated with an unknown initial value of the method in Phase II, for any particular call to the method, the run-time objects represented by the heap-name in the two cases are different. Consider the following methods which are part of class *test* defined in Figure 2:

```
A* method4( A *prm ) {          void method5() {
  A *p, *q;                        A *r, *s;
  1:  p = new A();                 4:  r = new A();
  2:  prm → next = 0;              5:  s = method4(r);
  3:  q = p → next;                6:  r = method4(s); }
  return p; }
```

The heap-name $object_1$ is a value of the unknown initial value prm_{init} and it is also used in *method4* during Phase I. The *next* field of $object_1$ is read at statement 3 and the type of $object_1$ is the same as the type of prm_{init}. However, at statement 2, when the *next* field of prm_{init} is modified, the *next* field of $object_1$ need not be considered for potential modification. This is safe because at statement 6, where $object_1$ is the value of prm_{init}, $object_1$ represents run-time objects created at statement 1 when *method4* is called from statement 5; and for the call to *method4* at statement 6, $object_1$ appearing in the Phase I solution of *method4* represents run-time objects created at statement 1 when *method4* is called from statement 6.[8]

3.3 Phase II

For each concrete value of an unknown initial value, computed at the entry node of a method M during this phase, RCI visits each of the call sites in M and does the following:

- For each dynamically dispatched call site C in M, RCI incrementally computes the set of methods invocable from C. Suppose the receiver at C is the value of a pointer variable p. Let S be the set of *dfelms* computed during Phase I that represent the values of p at C. RCI evaluates the *dfelms* of S by instantiating unknown initial values with their concrete values computed at the entry node of M. Those *dfelms* whose relevant contexts evaluate to *true* yield the concrete values of p at C, which are used to determine the set of methods invocable from C. Thus, at the end of Phase II, RCI produces the **final call graph** which is a significant refinement of the initial call graph. At a dynamically dispatched call site in the final call graph, the only targets considered invocable are those that are invocable using the concrete values of the receiver computed at the call site during Phase II.
- At each call site C, RCI uses the actual-to-uiv bindings computed during Phase I (see Section 3.2.1) to propagate concrete values to the methods invocable from C. RCI evaluates the relevant contexts associated with an actual-to-uiv binding by substituting the unknown initial values in the relevant contexts with their concrete values computed at the entry node of M. A binding is used for propagation if and only if at least one of the relevant contexts associated with the binding evaluates to *true*.

For methods in the same SCC, the concrete values are propagated iteratively until a fixed point is reached, while for methods in different SCC's, the propagation is done in a top-down manner without iteration.

In order to avoid propagation of concrete values from unreachable methods, RCI computes an initial set of *reachable* methods, and then incrementally expands this set during Phase II. For complete programs, the initial set consists of only *main*.[9] When RCI visits each node of SCC-DAG in a topological order during Phase II, it considers only those methods in the current SCC that have been marked reachable. Whenever RCI finds an unmarked method to be invocable from a call site in a reachable method, it marks the new method also as reachable.

For example, consider Phase II for the example given in Figure 2 and *method2* (see Section 3.2.1). For simplicity, assume that *method2* is reachable. As a result, *method1* is also reachable as it is invoked from *method2*. At call site 8, RCI stores an actual-to-uiv binding between $object_6$ and $a1_{init}$, and the only relevant context of this binding is $\langle empty,\ empty\rangle$. Since the relevant context $\langle empty,\ empty\rangle$ trivially evaluates to *true*, Phase II propagates $object_6$ as a concrete value of $a1_{init}$ to the entry node of *method1*. At the call site in statement 3, the value of the receiver is the value of *a1*, and the value of *a1* is given by the *dfelm* $\langle\langle empty,\ empty\rangle,\ \langle a1,\ a1_{init}\rangle\rangle$. Phase II substitutes $object_6$ for $a1_{init}$ in this *dfelm* to obtain $object_6$ as a concrete value of the receiver. This implies that *A::choose* is invocable from statement 3 and the final call graph has an edge from statement 3 to *A::choose*.

3.4 Phase III

Let n be a non-entry node in a reachable method M. If the solution of points-to analysis is needed at n, each *dfelm* computed at n during Phase I is instantiated with the concrete values computed at the entry node of M during Phase II. Those instantiations for which relevant contexts evaluate to *true* yield the solution of points-to analysis at n. After instantiation with concrete values, a *dfelm* yields a *points-to* of the form $\langle var,\ object\rangle$, where *var* is a local pointer variable, a global pointer variable or a heap-name's field of pointer type, and *object* is a heap-name.

[8] In C^{++} a pointer to a global can be created and hence a global can be a value of an unknown initial value. Therefore, a global can be used in a method either directly or through an unknown initial value, and thus a global can also be part of alias context.

[9] For incomplete programs like libraries, the initial set consists of all the methods that could be directly invoked from outside the incomplete program.

Consider Phase III for the example in Figure 2 and *method2* (see Section 3.2.1). Suppose, Phase III needs to be done at program point 4. For this, each *dfelm* at program point 4 is instantiated with the concrete values computed at the entry node of *method1*. For example, when $a1_{init}$ is instantiated by $object_6$ in the first *dfelm* at program point 4, the relevant context of the *dfelm* evaluates to *true* and the *dfelm* yields the *points-to* $\langle global2, object_6\rangle$.

3.5 Complexity

In this section, we briefly discuss the complexity of the various steps of *RCI*. We will focus on the important, dominating terms and, for simplicity, ignore less important terms. The analysis is for the subset of C++ in Figure 1, but this does not significantly impact the final results.

Phase 0. Let $\underline{T_{max}}$ be the maximum number of targets of a call site in the initial call graph, let $\underline{N_c}$ be the total number of call nodes and let $\underline{N_{proc}}$ be the total number of procedures/methods. The complexity of Phase 0 is $O(T_{max}N_c + N_{proc})$.

Phase I. The scheme for dealing with recursive types ensures that the total number of unknown initial values and hence the total number of possible *dfelms* is finite, even if no bound is imposed on the number of conjuncts in a relevant context. Since *RCI* does only a finite amount of work for each *dfelm* at a program point and at each step *RCI* considers a new *dfelm* at a program point, *RCI* always terminates. Let the total number of unknown initial values generated by *RCI* be $\underline{N_{uiv}}$, the number of user-defined pointer variables be $\underline{N_{var}}$, the maximum number of fields of a class (including inherited fields) be $\underline{F_{max}}$, the total number of classes be $\underline{N_{class}}$, the number of *ICFG* nodes be $\underline{N_{nodes}}$, the total number of heap-names be $\underline{N_h}$ and the bound on the number of conjuncts in a relevant context be $\underline{k}$.

Let $\underline{N_{rc}}$ be the number of possible relevant contexts. N_{rc} is at most $O((pa+tc)^k)$, where $\{pa = 2N_{uiv}^2,\ tc = N_{uiv}T_{max}\}$. Here *pa* is an upper bound on the number of possible potential aliases and potential non-aliases, and *tc* is an upper bound on the number of possible type constraints. Let $\underline{N_{pt}}$ be the number of possible *points-tos*. N_{pt} is at most
$O(fm * sm)$, where $\left\{\begin{array}{l} fm = (N_h + N_{uiv})F_{max} + N_{var} \\ sm = N_h + N_{uiv} \end{array}\right\}$.
Here fm is an upper bound on the number of pointers that can be the first member of a *points-to* pair and *sm* is an upper bound on the number of values for the second member. Now, let $\underline{N_{dfe}}$ be the total number of possible *dfelms*. N_{dfe} is at most $\underline{O(N_{rc}N_{pt})}$. Hence the total number of possible *dfelms* is polynomial in N_{uiv},N_{var},N_h, T_{max} and F_{max}, assuming k is constant.

Now consider the work done by *RCI* at a pointer assignment node. For each *dfelm* reaching such a node, $O(nl * nr)$, where $\{nl = (N_{uiv} + N_h) * N_{rc},\ nr = nl\}$, is an upper bound on the work done for *dfelms* directly generated by the pointer assignment statement. Here *nl* is an upper bound on the number of elements in *lhs_rc_loc_pairs* (see Appendix A) and similarly, *nr* is an upper bound on the number of elements in *rhs_rc_loc_pairs*. $O(N_{uiv} * cs)$, where $\{cs = N_{rc}N_{uiv}(N_{uiv} + N_h)\}$, is an upper bound on the work done in generating *dfelms* due to potential aliases. Here *cs* is an upper bound on the number of *dfelms* in *new_generated_dfes* (see Appendix A) that may generate *dfelms* due to potential aliases. Finally, $O(nl * cs)$ is an upper bound on the work done in generating *dfelms* due to potential non-aliases. Here *cs* is an upper bound on the number of *dfelms* in the current solution of the pointer assignment node that may generate *dfelms* due to potential non-aliases.

Similarly, it can be shown that at any node, for each *dfelm* reaching that node, the work done by *RCI* is polynomial in $N_{uiv}, N_h, N_{var}, T_{max}$ and F_{max} (assuming k is constant and a constant bound on the number of intraprocedural successors of a node). Let $\underline{N_{work}}$ be the maximum amount of work done by *RCI* for a *dfelm* at a program point.

Since at each step *RCI* considers a new *dfelm* at a program point, the total amount of work done by *RCI* is $O(N_{dfe} * N_{nodes} * N_{work})$. Hence the total work done by *RCI* is polynomial in $N_{uiv}, N_h, N_{var}, T_{max}, F_{max}$ and N_{nodes}, assuming k is constant.

$N_h, N_{var}, T_{max}, F_{max}$ and N_{nodes} are obviously bounded by the size of the program. However, in theoretically contrived cases, *RCI* can generate an exponential number of unknown initial values (see Appendix B). Although we never encountered this in practice, it can be easily avoided by enforcing a bound t on the lengths of *access paths* of unknown initial values (analogous to *k-limiting* [JM82]). Among the unknown initial values accessible from a root unknown initial value, all the unknown initial values of the same type and having *access paths* longer than t will be represented by the same representative name. This will ensure that N_{uiv} is polynomial in $N_{proc}, N_{var}, F_{max}$ and N_{class}.

Phase II. Let $\underline{C_{max}}$ be the maximum number of call nodes in a procedure/method. Let $\underline{N_{map}}$ be the maximum number of actual-to-uiv mappings stored at a call node. N_{map} is at most $O(np * nr)$, where $\{np = (N_h + N_{uiv})N_{uiv},\ nr = N_{rc}\}$. Here *np* is the maximum number of pairs of actuals and unknown initial values and the term *nr* counts the relevant contexts associated with each actual-to-uiv mapping. Let $\underline{N_{prop}}$ be the maximum amount of work done by *RCI* at a call node for a concrete value of an unknown initial value. N_{prop} is at most $O(N_{map} * (ce + cp))$, where $\{ce = N_h^{2k}k * N_h,\ cp = N_h\}$. Here *ce* is the worst case cost of evaluating a relevant context and the associated actual. This is because each unknown initial value can have at most N_h concrete values. *cp* is the worst case cost of propagating concrete values to the entry node of a target procedure. So the total amount of work done during this phase in the worst case is $O((N_{uiv}N_h)C_{max}N_{prop})$. Here $O(N_{uiv}N_h)$ is an upper bound on the number of pairs of unknown initial values and their concrete values.

Phase III. The worst case of evaluating a *dfelm* during this phase is $O(er * ep)$, where $\{er = N_h^{2k}k,\ ep = N_h^2\}$. Here *er* is the worst case cost of evaluating the relevant context of the *dfelm* and *ep* is the worst case cost of evaluating the *points-to* of the *dfelm*. Hence the worst case cost of this phase is $O(N_h^{2k+2}k * N_{nodes} * N_{dfe})$.

4 Extensions

In this section, we briefly describe some extensions of *RCI* whose details are given in [CRL98b].

RCI has been designed to work on both C++ and Java, as our subset of C++ contains the essential features of both languages except threads. Modular points-to analysis has been extended to handle Java exceptions. The essential idea is to store additional information in the relevant context that summarizes control-flow due to exceptions. The modular nature of *RCI* is unaffected and its overall structure remains the same. Only the definition of the *dfelms* computed by Phase I must be extended to account for the possible control-flow due to exceptions. Under the same assumptions as in Section 3.5, *RCI* is polynomial-time even in the presence of exceptions. A complexity classification of doing analysis in the presence of exceptions is given in [CRL98a].

Another important property of *RCI* is that incomplete programs such as libraries can be analyzed. A unknown initial value at the entry node of a method of a library, that can be directly invoked from a call site outside the library, is called an **interface initial value**. When *RCI* is used for analyzing a library L, during Phase II *RCI* treats an interface initial value like a concrete value and similarly propagates an interface initial value to the entry nodes of other methods. During Phase II or III, when *RCI* instantiates an unknown initial value in a conjunct with an interface initial value, conservative worst-case assumptions are made about the interface initial value for evaluating the conjunct.

Relevant contexts computed by *RCI* can be used for generating relevant test cases for libraries. These contexts provide valuable information about how a library may be used in general; this information cannot be obtained from a whole-program-analysis of a particular driver with the library. These contexts suggest a new coverage measure for the unit testing of libraries that can be used with other standard coverage measures.

5 Implementation

Our implementation has been built using the PROLANGS Analysis Framework (PAF) which incorporates the Edison Design Group front end for ANSI C^{++}.[10] Our initial empirical results with modular points-to analysis are encouraging; however, this is a proof-of-concept implementation and there is scope for optimization.

Table 1 contains some characteristics of the thirteen C^{++} programs we have analyzed. These are some of the benchmarks used in [PR96, BS96, CGZ95].[11] The columns *lines*, *ICFG nodes*, *methods*, *virtual calls*, *SCC's* and *Max SCC* respectively show the number of lines of code, ICFG nodes, methods, dynamically dispatched call sites, nodes in SCC-DAG and methods in the maximum-sized SCC for each program.

Table 2 contains the timings using a Sparc-20 with 352 megabytes of memory. These timings do not include time for scanning, parsing or I/O. The column *Bounds* contains pairs (i, j) which mean that the bounds on the number of potential aliases and type constraints of each relevant context were i and j respectively.

We analyzed *richards*, *deriv1*, *FeynLib*, *opProd*, *penguin*, *Bdecay* and *electron* with more than one set of bounds. The second row of (1,1) for *richards* corresponds to an analysis that only allowed either one potential alias or one type constraint per relevant context. Different bounds yielded measurable variations in those characteristics shown in Tables 2 and 3, but there was no difference in the information reported for the two applications in Table 4. We analyzed with smaller bounds those programs with large running times for higher bounds, and studied the cost-precision tradeoff with respect to the applications reported in Table 4.

Although Phase III is demand-driven, for these experiments Phase III was performed at all non-entry nodes. The Phase I solution of a library can be shared by different driver programs. This is illustrated by the Phase I timings of *Bdecay*, *electron*, *opProd* and *penguin*. For these programs, $t_1 + t_2$ means the time for the driver code is t_1 and the time for the library code is t_2, a shared cost which is only incurred once.

Although *trees* and *employ* are small benchmarks, these are important to show the two orders of magnitude timing improvement over a previous flow- and context-sensitive, whole-program-analysis technique [PR96], which took 690 and 450 seconds for these benchmarks.

In Table 3, the columns *PA's* and *TC's* show the total number of potential aliases and type constraints generated. The column *Max Size Relevant Context* shows the maximum number of conjuncts in any non-empty relevant context. In order to get an estimate of the memory saving obtained by using summary transfer functions, we normalized the size of the summary transfer function of each method with the size of the complete Phase I solution of the method. To compute the latter, we considered only those nodes which are relevant for points-to analysis: we excluded those nodes which preserve points-to information. Then we averaged these normalized sizes over all methods in a program to compute the average size of a summary transfer function for a program; results are presented as a percentage in column *Ave Size Summ Fcn*. Another possibility would have been to compare the sizes of the summary transfer functions with the number of nodes in the corresponding methods. But this is not reasonable because the cost of reanalyzing a method depends upon the size of the points-to solution of the method and the number of nodes reachable from the method during an invocation (not just the number of nodes in the method). Our comparison gives a better indication of the reduction in memory requirement achieved, as *RCI* needs to keep in memory only the summary transfer functions of the methods called from the current SCC, rather than the complete solution of those methods.

No counts are reported for potential non-aliases because at present, our implementation does not generate any potential non-aliases. By Theorem 1, this does not affect safety of the computed solution. The impact of potential non-aliases on the precision of points-to analysis is likely to be small, because a potential non-alias must evaluate to *false* to increase precision, and this will only occur if both uiv_1 and uiv_2 map to the same unknown initial value during Phase I. If they both map to the same heap-name, the potential non-alias evaluates to *true* because the heap-name can represent more than one run-time object. We have chosen to describe *RCI* as including calculation of potential non-aliases, because we believe they can be useful for other applications (e.g., analyzing libraries, testing etc).

In order to test the quality of the solution computed by *RCI*, we used the solution for two different applications: *side-effect analysis* or **MOD** [LRZ93, SRLZ98] and *virtual function resolution* [PR96]. The results for *MOD* are shown in Table 4. The column *AvConcrete MOD* shows the average number of heap-names whose fields are modified by a pointer-assignment statement according to the Phase III solution. In object-oriented programs, since a method is usually called from many different contexts, the number of heap-names modified at a statement could be large even in

[10] See **http://www.prolangs.rutgers.edu/public.html**.

[11] *trees* implements trees, *deriv1* implements arithmetic expression trees, *employ* implements a class hierarchy for different kinds of employees in a company, *richards* is an operating system scheduler, *deltablue* is a symbolic constraint solver, *sampleAdv, vmatrix* and *vvector* perform matrix computations, and *FeynLib* is a library for drawing Feynman Diagrams for which *Bdecay, electron, opProd and penguin* are drivers for different kinds of elementary particles.

the precise solution. We verified by inspection that multiple calling contexts is the reason why some of the *AvConcrete MOD* numbers are high. Therefore, we used the Phase I solution to compute the average number of unknown initial values or heap-names whose fields are modified at a pointer-assignment statement. This factors out the effect of expansion of an unknown initial value into multiple concrete values because of the invocation of a method from multiple contexts. Column *AvAbstract MOD* shows this second average over all pointer-assignment statements for each benchmark.

Table 4 and Figure 3 show our results for virtual function resolution. *RCI* is probably more precise than necessary for solving virtual function resolution for C^{++}, because it is really aimed at problems where there is more gain from flow and context sensitivity; nevertheless, our results show that there are calls for which *RCI* enables much better resolution, with concomitant opportunities for aggressive optimizations such as instruction scheduling and specialization. Column *Reachable Virtual Calls* shows how many of the virtual calls in each program, are actually reachable (with the given drivers for libraries). Column *Unique Hierarchy* shows the number of reachable calls which are uniquely resolved by hierarchy analysis. Column *Differences RCI-Hierarchy* shows the number of those reachable calls for which the number of targets found by *RCI* is less than the number of targets found by hierarchy analysis. In Figure 3 we show the number of these differing calls in each program and the sizes of the differences. As noted above, changing bounds made no difference in these results. The possible loss of precision due to imposing bounds on the number of conjuncts in a relevant context, was not observed in the applications we considered. For them, using lower bounds improved running time significantly. However, for other applications and benchmarks, the situation might be different.

6 Related Work

Wilson and Lam [WL95] have also used unknown initial values in their algorithm for points-to analysis of *C* programs; however, there are significant differences between *RCI* and their approach. Their algorithm needs to know the exact alias relationships between the unknown initial values before a procedure can be analyzed. Their algorithm keeps the whole program in memory and cannot analyze incomplete programs. They construct *partial transfer functions*, while *RCI* constructs summary transfer functions that approximate complete transfer functions. As their algorithm is for *C*, they handle neither dynamic dispatch nor exceptions, however, they do handle function pointers. Sometimes, if there are very few calls to a method in a complete program and very few of the possible relevant contexts occur, using summary transfer functions instead of partial transfer functions could be more costly.

Unknown initial values are similar to *non-visible variables* used in [LR92] and *invisible variables* used in [EGH94] for summarizing the values of pointers that point to out-of-scope variables. Moreover, there are aspects of *hybrid data-flow analysis* [MR90] in which unknown initial values are used to model representative external values in a local analysis.

In [FF97], the authors present a method for doing set-based analysis in a component-wise manner for Scheme programs. Since this is not a data-flow analysis and is not aimed at an object-oriented language, it is difficult to directly compare this work to *RCI*. Essentially, [FF97] describes an optimization technique for set-based analyses, when it is known

programs	lines	ICFG nodes	methods	virtual calls	SCC's	Max SCC
trees	217	280	25	3	21	3
deriv1	192	320	27	28	23	3
employ	947	463	60	4	60	1
richards	987	918	87	82	87	1
deltablue	1509	1471	112	185	112	1
sampleAdv	3130	2203	130	9	130	1
vmatrix	3607	2783	128	9	128	1
vvector	3681	3401	138	9	138	1
FeynLib	6222	7247	229	47	219	4
opProd	6289	7340	230	47	219	4
penguin	6313	7450	230	47	219	4
Bdecay	6293	7465	230	47	219	4
electron	6321	7502	230	47	219	4

Table 1: Benchmarks

program	Ph 0	Ph I	Ph II	Ph III	Bounds
trees	0.01	0.64	0.05	0.30	(∞,∞)
deriv1	0.02	5.95	0.05	0.66	(∞,∞)
deriv1		2.42	0.06	0.52	$(\infty,1)$
employ	0.01	0.76	0.05	0.31	(∞,∞)
richards	0.02	33.50	0.16	9.99	(∞,∞)
richards		17.56	0.15	8.38	(1,1)
richards		9.87	0.17	6.22	(1,1)
deltablue	0.03	4.33	0.24	1.84	(∞,∞)
sampleAdv	0.03	4.42	0.03	0.79	(∞,∞)
vmatrix	0.04	6.98	0.12	4.07	(∞,∞)
vvector	0.05	10.27	0.16	5.87	(∞,∞)
opProd	0.08	3.21+99.52	2.74	34.46	(1,1)
opProd		5.38+437.72	1.64	68.38	(2,2)
penguin	0.09	3.98+99.52	2.14	30.29	(1,1)
penguin		5.25+437.72	1.52	51.70	(2,2)
Bdecay	0.08	4.15+99.52	2.10	29.92	(1,1)
Bdecay		7.24+437.72	1.41	51.54	(2,2)
electron	0.09	5.06+99.52	2.18	33.41	(1,1)
electron		4.25+437.72	1.50	61.59	(2,2)

Table 2: Timings in Seconds

program	PA's	TC's	Max Size Relevant Context	Ave Size Summ Fcn	Bounds
trees	4	12	1	8.7%	(∞,∞)
deriv1	0	116	3	6.4%	(∞,∞)
deriv1	0	116	1	6.3%	$(\infty,1)$
employ	0	36	1	5.8%	(∞,∞)
richards	85	5	3	9.0%	(∞,∞)
richards	84	5	2	9.0%	(1,1)
richards	77	5	1	9.0%	(1,1)
deltablue	1	12	1	3.9%	(∞,∞)
sampleAdv	11	5	2	7.4%	(∞,∞)
vmatrix	10	2	2	7.4%	(∞,∞)
vvector	10	6	2	7.2%	(∞,∞)
FeynLib	115	535	1	3.5%	(1,1)
FeynLib	245	579	2	3.7%	(2,2)
opProd	115	535	1	3.5%	(1,1)
opProd	245	579	2	3.7%	(2,2)
penguin	115	535	1	3.5%	(1,1)
penguin	245	579	2	3.7%	(2,2)
Bdecay	115	535	1	3.5%	(1,1)
Bdecay	245	579	2	3.7%	(2,2)
electron	115	535	1	3.5%	(1,1)
electron	245	579	2	3.7%	(2,2)

Table 3: Performance Data

program	AvAbstract MOD	AvConcrete MOD	Reachable Virtual Calls	Unique Hierarchy	Differences RCI-Hierarchy	Bounds
trees	1	1.54	1	0	1	(∞,∞)
deriv1	1	3.67	20	10	10	(∞,∞)
employ	1	2.60	4	0	4	(∞,∞)
richards	1.100	2.48	82	81	1	(∞,∞)
deltablue	1.062	2.50	133	132	1	(∞,∞)
sampleAdv	1	5.14	2	0	2	(∞,∞)
vmatrix	1	15.14	6	5	1	(∞,∞)
vvector	1	9.40	2	0	2	(∞,∞)
opProd	1.007	2.54	5	0	5	(1,1)
penguin	1.007	1.88	6	0	6	(1,1)
Bdecay	1.007	1.98	4	0	4	(1,1)
electron	1.007	2.17	7	0	7	(1,1)

Table 4: Applications

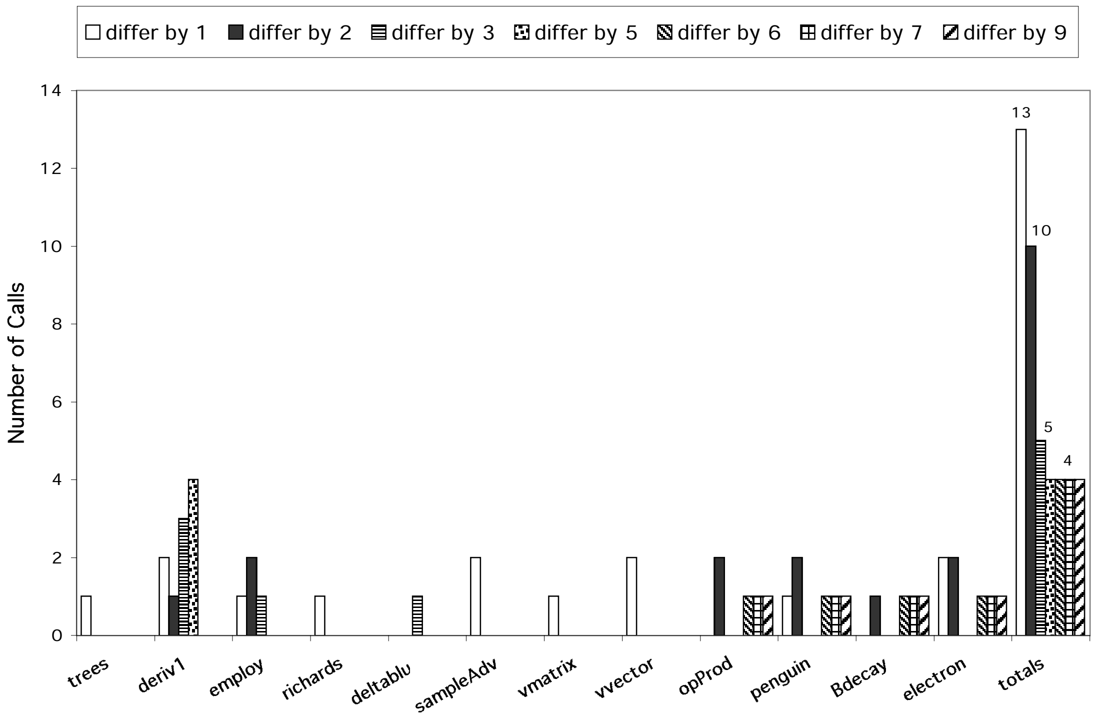

Figure 3: Differences in Precision between RCI and Hierarchy Analysis

that different sets of constraints correspond to separate modules of the program. It is modular in the sense that the simplified constraint sets can be obtained separately. Using the terminology of Section 1, the algorithm of [FF97] is flow-insensitive and therefore, does not perform strong updates.

Concrete type inference and call graph construction for object-oriented languages are subsumed by our analysis. There are many non-modular whole-program-analysis approaches for these two problems. Most use constraint-based analysis [Suz81, PS91, PC94, Age95, GDDC97, DGC98], but a few [CHS95, PR96, DMM96] are data-flow-based. Both [CHS95, PR96] are flow- and context-sensitive algorithms for C^{++}; we have compared our empirical results with those in [PR96] which were non-scalable. No implementation is reported in [CHS95]. [DMM96] presents different type analysis techniques for Modula-3, including hierarchy analysis, flow-sensitive intraprocedural type propagation and context-insensitive interprocedural type propagation. [GDDC97] presents a more complex notion of context which subsumes calling context and type context for some program variables. It presents a general framework for call graph construction of object-oriented programs. Among constraint-based approaches, [Suz81, DGC98] are flow- and context-insensitive, while [PS91, PC94, Age95] are flow-insensitive, but context-sensitive. None of the above approaches handle exceptions. [BS96] is an extension of hierarchy analysis for C^{++} and thus it is flow- and context-insensitive.

7 Conclusions

A new technique called *RCI* for *modular* flow- and context-sensitive data-flow analysis has been presented, in the context of its application to points-to analysis for a substantial subset of C^{++}. This scalable algorithm does whole program analysis without needing the entire program source in memory; each method must be in memory only four times during algorithm execution. Initial empirical results attest to the effectiveness of this technique on moderate sized programs. Extensions for Java with exceptions (but without threads) and analysis of libraries also were discussed.

Acknowledgements. We thank the anonymous reviewers for their helpful comments, Tom Marlowe for reviewing an earlier draft of this paper, and Atanas Rountev and Matt Arnold for helping with the implementation.

References

[Age95] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of European Conference on Object-oriented Programming (ECOOP '95)*, 1995.

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

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 1996.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[CGZ95] B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioural differences between C and C^{++} programs. *Journal of Programming Languages*, 2:313–351, 1995.

[CHS95] P. Carini, M. Hind, and H. Srinivasan. Flow-sensitive interprocedural type analysis for c++. Technical Report RC 20267, IBM T.J. Watson Research Center, 1995.

[CLR92] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. The MIT Press and McGraw-Hill Book Company, 1992.

[CRL98a] Ramkrishna Chatterjee, Barbara Ryder, and William Landi. Complexity of concrete type-inference in the presence of exceptions. In *LNCS 1381, Proceedings of European Symposium on Programming*, April 1998.

[CRL98b] Ramkrishna Chatterjee, Barbara Ryder, and William Landi. Relevant context inference. Technical Report DCS-TR-360, Dept of CS, Rutgers University, August 1998.

[Deu94] A. Deutsch. Interprocedural may alias for pointers: Beyond k-limiting. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 230–241, June 1994.

[DGC98] Greg DeFouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 222–336, 1998.

[DMM96] Amer Diwan, J.Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 1996.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation*, pages 242–256, 1994.

[Ema93] Maryam Emami. A practical interprocedural alias analysis for an optimizing/parallelizing c compiler, master's thesis. Technical report, School of Computer Science, McGill University, August 1993.

[FF97] Cormac Flanagan and Matthias Felleisan. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation*, pages 235–248, 1997.

[GDDC97] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '97)*, pages 108–124, October 1997.

[Ghi96] Rakesh Ghiya. Connection analysis: A practical interprocedural heap analysis for c. *International Journal of Parallel Programming*, 24(6):547–578, 1996.

[JM82] N. D. Jones and S. Muchnick. Flow analysis and optimization of lisp-like structures. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. 1982.

[LR91] W.A. Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 1991.

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

[LRZ93] W.A. Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1993.

[MLR+93] T. J. Marlowe, W. A. Landi, B. G. Ryder, J. Choi, M. Burke, and P. Carini. Pointer-induced aliasing: A clarification. *ACM SIGPLAN Notices*, 28(9):67–70, September 1993.

[MR90] Thomas. J. Marlowe and Barbara G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 184–196, January 1990.

[PC94] J. Plevyak and A. Chien. Precise concrete type inference for object oriented languages. In *Proceeding of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '94)*, pages 324–340, October 1994.

[PR96] Hemant Pande and Barbara G. Ryder. Data-flow-based virtual function resolution. In *LNCS 1145, Proceedings of the Third International Symposium on Static Analysis*, 1996.

[PS91] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91)*, pages 146–161, October 1991.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[Ruf95] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation*, pages 13–22, June 1995.

[SH97] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[SRLZ98] P.A. Stocks, B.G. Ryder, W.A. Landi, and S. Zhang. Comparing flow- and context-sensitivity on the modification side-effects problem. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 21–31, March 1998. Also available as DCS-TR-335.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[Suz81] N. Suzuki. Inferring types in smalltalk. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 187–199, 1981.

[Wei80] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, January 1980.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation*, pages 1–12, 1995.

[ZRL96] S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer aliasing: A step towards practical analyses. In *Proceedings of the 4th Symposium on the Foundations of Software Engineering*, October 1996.

A Pointer Assignment

The data-flow transfer function, *apply*, for a pointer assignment node n, is defined in Figures 4 and 5. *n.lhs, n.rhs* and *n.sol* are respectively the left hand side expression, the right hand side expression and the current solution at n. *rdfe* is a **new** *dfelm* $\langle rc, \langle x, y\rangle\rangle$ reaching the node n (i.e., *rdfe* $\notin$ *n.sol*). *lhs_rc_loc_pairs* is the set of pairs of relevant contexts and objects that are modified by node n. Note that if *n.lhs* has a dereference, *lhs_rc_loc_pairs* is initially ϕ. Otherwise, it is initially $\langle\langle empty,\ empty\rangle,\ n.lhs\rangle$. Consider the following example. Let *n.lhs* be *p→f1*. A *rdfe* $(\langle rc,\ \langle x,\ y\rangle\rangle)$ can imply a new *lhs_rc_loc_pair* if and only if x is p. Thus if the on top solution for this assignment was

$$\langle\langle empty,\ empty\rangle,\ \langle p,\ l\rangle\rangle$$
$$\langle\langle (p_{init}\ \mathbf{eq}\ q_{init}),empty\rangle,\ \langle p,\ p_{init}\rangle\rangle$$
$$\langle\langle empty,\ empty\rangle,\ \langle q,\ m\rangle\rangle$$

then *lhs_rc_loc_pairs* would be

$$\left\{ \begin{array}{l} \langle\langle empty,\ empty\rangle,\ l\rangle, \\ \langle\langle (p_{init}\ \mathbf{eq}\ q_{init}),empty\rangle,\ p_{init}\rangle \end{array} \right\}$$

If *n.lhs* does not have dereference (i.e., *n.lhs* is a variable name p), then *lhs_rc_loc_pairs* contains a single element $\langle\langle empty,\ empty\rangle,\ p\rangle$ and *new_lhs_rc_loc_pairs* is always ϕ. *rhs_rc_loc_pairs* is very similar to *lhs_rc_loc_pairs* except it is for *n.rhs* (i.e., the set of pairs of relevant contexts and objects that are values of *n.rhs*).

pot_aliases computes *dfelms* that need to be generated due to potential aliases. *pot_non_aliases* computes *dfelms* that need to be generated due to potential non-aliases.

The *if* statement at program point 3 checks if *rdfe* is killed by node n. *kills* returns true if *rdfe* represents the value of a pointer variable and n directly updates this pointer variable. If n does not directly kill *rdfe*, there are two cases:

1. The assignment kills *rdfe* if certain unknown initial values are not distinct. Consider when *n.lhs* is of the form *p→f1*, *rdfe* represents the value of $f1$ field of an unknown initial value s and p can point to a location that can be the concrete value of s (i.e., if p is of type $A*$, A and the type of s are *compatible*). In this case, *pot_non_aliases* is called to generate *dfelms* that condition the propagation of *rdfe* across n on potential non-aliases. These *dfelms* require that s is not equal to any of the unknown initial values that are *compatible* with s and to which p currently points. If p points to an object that cannot be same as s, *rdfe* is propagated unconditionally across n.

2. *n.lhs* cannot update the location whose value *rdfe* represents. In this case, *rdfe* is propagated across n unconditionally. Here *RCI* makes conservative assumptions about heap-names. Since, a heap-name can represent more than one run-time object, *RCI* propagates *rdfe* unconditionally across n if *rdfe* represents the value of a field of a heap-name.

B Complexity of *RCI*

In the following example, at program point s, p_n points to an exponential number of unknown initial values.

```
class A1 {                  void method( A1 *a) {
  public:  A2 *f1;            A1 *p1;
  public:  A2 *f2;            ...
};                            Ai *pi;
...                           ...
class Ai {                    An *pn;
  public:  Ai+1 *f1;          p1 = a;
  public:  Ai+1 *f2;          if (_) p2 = p1→f1;
};                              else p2 = p1→f2;
...                           ...
class An-1 {                  if (_) pi = pi-1→f1;
  public:  An *f1;              else pi = pi-1→f2;
  public:  An *f2;            ...
};                            if (_) pn = pn-1→f1;
class An { };                   else pn = pn-1→f2;
                              s:
                            }
```

```
apply( rdfe, n ) {
// rdfe = ⟨rc, ⟨x, y⟩⟩ is a new dfelm
// reaching the pointer assignment node n.

old_lhs_rc_loc_pairs = lhs_rc_loc_pairs;
old_rhs_rc_loc_pairs = rhs_rc_loc_pairs;

new_lhs_rc_loc_pairs =
new rc_loc_pairs for lhs implied by rdfe;

new_rhs_rc_loc_pairs =
new rc_loc_pairs for rhs implied by rdfe;

lhs_rc_loc_pairs =
old_lhs_rc_loc_pairs ∪ new_lhs_rc_loc_pairs;

rhs_rc_loc_pairs =
old_rhs_rc_loc_pairs ∪ new_rhs_rc_loc_pairs;

new_generated_dfes = φ;
// following loop is executed only if n.lhs
// has dereference.  Otherwise new_lhs_rc_loc_pairs
// is φ.
1:  for (each ⟨rc1, u⟩ in new_lhs_rc_loc_pairs) {
  if (u != null) {
    for (each ⟨rc2, v⟩ in rhs_rc_loc_pairs) {
      rc3 = rc1 and rc2;
      rc4 = impose user defined bounds on rc3;
      // assuming n.lhs is of the form p→f1
      new_dfe = ⟨rc4, ⟨u.f1, v⟩⟩;
      new_generated_dfes ∪ {new_dfe};
    }
  }
}

2:  for (each ⟨rc2, v⟩ in new_rhs_rc_loc_pairs) {
  for (each ⟨rc1, u⟩ in old_lhs_rc_loc_pairs) {
    if (u != null) {
      rc3 = rc1 and rc2;
      rc4 = impose user defined bounds on rc3;
      if (n.lhs has dereference)
        // assuming n.lhs is of the form p→f1
        new_dfe = ⟨rc4, ⟨u.f1, v⟩⟩;
      else
        new_dfe = ⟨rc4, ⟨u, v⟩⟩;
      new_generated_dfes ∪ {new_dfe};
    }
  }
}

new_generated_dfes = new_generated_dfes ∪
pot_aliases(new_generated_dfes, n);

new_generated_dfes = new_generated_dfes ∪
pot_non_aliases(new_lhs_rc_loc_pairs, n.sol, n);

3:  if ( !kills(n, rdfe) ) {
  if (can_update(x,n)) {
    new_generated_dfes = new_generated_dfes ∪
    pot_non_aliases(lhs_rc_loc_pairs, {rdfe}, n);
  }
  else {
    new_generated_dfes ∪ {rdfe};
  }
}

return new_generated_dfes;
}
```

Figure 4: apply1

```
kills( n, rdfe ) {
// rdfe = ⟨rc, ⟨x, y⟩⟩
  if (n.lhs does not have dereference) {
    if (x and n.lhs are the same variable)
      return true;
  }
return false;
}

can_update( x, n ) {
if (n.lhs does not have dereference)
  return false;

// else say n.lhs is p→f1 and p is of type A *
if (x is s.f1 and s is an unknown initial value whose
    type is compatible with A)
  return true;
else
  return false;
}

pot_aliases( dfes, n ) {
if (n.lhs does not have dereference)
  return φ;

// else say n.lhs is p→f1
generated_dfes_pa = φ;
for (each dfe ⟨rc1, ⟨u.f1, y⟩⟩ in dfes) {
  if (u is an unknown initial value ) {
    for (each z.f1 such that z is an unknown initial value
          compatible with u and z.f1 has been found
          to be used) {
      rc2 = rc1 and (z eq u);
      rc3 = impose user defined bounds on rc2;
      new_dfe = ⟨rc3, ⟨z.f1, y⟩⟩;
      generated_dfes_pa ∪ {new_dfe};
    }
  }
}
return generated_dfes_pa;
}

pot_non_aliases( rc_loc_pairs, dfes, n ) {
if (n.lhs does not have dereference)
  return φ;

// else say n.lhs is p→f1 and p is of type A *
generated_dfes_pna = φ;
for (each dfe ⟨rc1, ⟨u.f1, y⟩⟩ in dfes such that u is a
      unknown initial value whose type is compatible with A) {
  for (each ⟨rc2, v⟩ in rc_loc_pairs) {
    if (v is an unknown initial value compatible with u) {
      if (v != u) {
        rc3 = rc1 and rc2 and (u neq v);
        rc4 = impose user defined bounds on rc3;
        new_dfe = ⟨rc4, ⟨u.f1, y⟩⟩;
        generated_dfes_pna ∪ {new_dfe};
      }
    }
    else {
      generated_dfes_pna ∪ {⟨rc1, ⟨u.f1, y⟩⟩};
    }
  }
}
return generated_dfes_pna;
}
```

Figure 5: apply2