Relevant Context Inference

DCS-TR-360

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 2901 {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. We show that RCI can handle exceptions. We also discuss application of RCI to unit testing of libraries and explain how the information computed by RCI can be used for generating relevant test cases. RCI is presented in the context of points-to analysis. 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, 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, Ru95, 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++/Java is difficult, due to dynamic dispatch, objects containing pointers to subobjects (i.e., recursive types), exceptions, 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 simple object-oriented language that captures the essential features of C++ and Java (except threads). 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 information, by inferring them from the code of the method and those other methods it may invoke directly or indirectly during its lifetime. 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 twofold: (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++/Java. Moreover, although in this paper we explain RCI in the context of points-to analysis, the technique can be extended 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 simple object-oriented language that captures the essential features of C++ and Java (except threads).
- An approach to handle exceptions in the context of modular analysis.
- An approach to analyze a special class of incomplete programs, such as libraries.
- The definition of a new coverage metric and a discussion of how RCI can help in generating relevant test cases for the unit testing of libraries.
- Empirical evidence for the effectiveness of RCI as applied to points-to analysis of C++.

The rest of this paper is organized as follows. Section 2 contains some technical definitions needed to explain RCI. Section 3 presents RCI for points-to analysis. Section 4 explains how RCI handles exceptions. Section 5 explains how RCI analyzes libraries and how the information computed by RCI can be used for generating relevant test cases for libraries. Section 6 contains empirical evidence for the effectiveness of RCI. Section 7 compares RCI with other approaches. Finally, Section 8 summarizes the main points of this work and discusses directions for future work.

2 Definitions

This section presents many technical definitions needed to explain the algorithm and also delimits the subsets of C++ and Java that are handled.

RL. For the ease of presentation, in this paper we describe RCI for a simple object-oriented language RL that has the essential features of C++ and Java (except threads). This allows us to simplify the presentation while demonstrating the interesting parts of our algorithm. RL is defined in Figure 1. It includes single inheritance, dynamic dispatch, recursive types, exceptions and pointer assignment statements with a single level of dereferencing. The semantics of exceptions is same as in Java. The semantics of other constructs is same as in C++. RL excludes multiple inheritance, an explicit address operator (i.e., pointers to the stack), the C++ 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 and arrays (array elements are mapped to a single representative element). Many of these constructs can be easily accommodated by extending the algorithm; we will briefly indicate these extensions where relevant.

If the algorithm is understood fully for RL, then handling of most of C++ and Java (except threads) 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 RCI excludes arbitrary casting, uninstanciated 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 derived class to a base class can be handled. Our implementation of the algorithm is consistent with this bigger subset, only excluding in addition, multiple inheritance and exceptions.

In addition, RCI can essentially handle Java without threads. Under certain circumstances, we have to exclude some other features: Since, in Java, finalizers are invoked non-deterministically during garbage collection, we exclude finalizers that modify pointers. RCI can handle only those finalizers that can be ignored for points-to analysis. We exclude static initializations that depend upon the order in which files are loaded. RCI can handle only those static initializations for which it is safe (with respect to points-to analysis) to consider them to be executed in program source order (or any other order specified by the user) at the start of program execution. Finally, we exclude classes whose code is constructed on the fly and is not known statically.

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

\[(\text{Expr}) \rightarrow \{ \text{pattern} \}
\]

\[(\text{pattern})^+ \text{ means one or more occurrences of the pattern.}\)

\[(\text{pattern})^* \text{ means zero or more occurrences of the pattern.}\)

\[\{ a | b \} \text{ means } a \text{ or } b. \]
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\textsubscript{init} to represent the unknown initial value for the global or parameter var. Note that var\textsubscript{init} denotes the unknown initial object to which var points and not the address of that object. var\textsubscript{init} represents the unknown initial value to which var\textsubscript{next} points. var\textsubscript{init}, next\textsubscript{init} represents the unknown initial value to which var\textsubscript{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 (see Appendix D for details).

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

\[(relevant\ context,\ points-to),\]

where points-to represents a pair of the form:

\[(var,\ object)\]

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:

\[(alias\ context,\ type\ context)\]

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:

\[(uiv\textsubscript{1}\ \textbf{eq} \ uiv\textsubscript{2})\]

and each potential non-alias has the form:

\[(uiv\textsubscript{1} \textbf{neq} uiv\textsubscript{2})\]

Here uiv\textsubscript{1} and uiv\textsubscript{2} are unknown initial values. These potential aliases and potential non-aliases are inferred by the algorithm during analysis; some dfelms 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:
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, first an initial approximate call graph is formed and then decomposed into strongly connected components (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

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

Figure 2: Phase I

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.

- **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

A high-level pseudocode for Phase I is given in Figure 2.

mark-reachable() initialize-worklist() process-worklist()

for each SCC in reverse topological order

mark-reachable() initialize-worklist() process-worklist()
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 dependencies, 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 3 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 $a_3$ to the next field of the value of $a_1$, 2:1, the first dfelm at program point 2, says that the next field of the unknown initial value of $a_1$ points to the unknown initial value of $a_3$. The relevant context (empty, empty) 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 (empty, empty). If the unknown initial values of $a_1$ and $a_2$ are the same, statement 1 also modifies the next field of the unknown initial value of $a_2$. The dfelms 2:2 and 2:3 keep track of this potential modification. 2:2 is applicable to only those contexts in which $a_{1\text{init}}$ and $a_{2\text{init}}$ are equal, while 2:3 is applicable to only those contexts in which $a_{1\text{init}}$ and $a_{2\text{init}}$ are not equal. The dfelms 3:1 and 3:2 are implied respectively by 2:2 and 2:3.

RCI does not consider $a_{3\text{init}.next}$ 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 $a_{3\text{init}}$ with $a_{1\text{init}}$ and $a_{2\text{init}}$ are not relevant to method1 because first, $a_{3\text{init}.next}$ is not used in method1 and second, $a_{2\text{init}.next}$ 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 $a_{1\text{init}}$. As a result, RCI computes the values of global2 at program point 4 conditioned on the potential concrete types of $a_{1\text{init}}$. The dfelm 4:1 says that global2 points to $a_{1\text{init}}$ in those contexts in which the concrete type of $a_{1\text{init}}$ 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 $a_{2\text{init}}$ and $a_{3\text{init}}$ are not relevant to method1. 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.

```cpp
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); } 
```

At call site 5, RCI translates the dfelms at the exit node of method1 by replacing the unknown initial values of $a_1$, $a_2$ and $a_3$ with their values at call site 5 (i.e., the unknown initial values of $a_4$, $a_5$ and $a_6$ respectively). Note: this is propagating data-flow information from callee back to caller.

For example:

- $\langle\langle(a_{1\text{init}} eq a_{2\text{init}}),\emptyset), (\text{global1}, a_{3\text{init}})\rangle$ is translated to $\langle\langle(a_{1\text{init}} eq a_{5\text{init}}),\emptyset), (\text{global1}, a_{6\text{init}})\rangle$.
- $\langle\emptyset,(\text{type}(a_{1\text{init}}) C:choose)\rangle$, $\langle\text{global2}, a_{2\text{init}}\rangle$ is translated to $\langle\emptyset,(\text{type}(a_{4\text{init}}) C:choose)\rangle$, $\langle\text{global2}, a_{5\text{init}}\rangle$.

\*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 Section 5).
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);
    }
    ...
};

At call site 8, RCI evaluates the relevant contexts of the deflms at the exit node of method1 using objects as a1init and objectx as a2init and a3init. For example:

- $((a_{1\text{init}} \text{ eq } a_{2\text{init}}), \text{empty})$, $(\text{global1}, a_{3\text{init}})$ is not applicable to this call site as $(\text{objectx} \text{ eq } \text{objecty})$ is false.
- $((a_{1\text{init}} \text{ neq } a_{2\text{init}}), \text{empty})$, $(\text{global1}, a_{2\text{init}}, \text{next}_{\text{init}})$ is translated to $((\text{empty}, \text{empty})$, $(\text{global1}, \text{null})$ because $(\text{objectx} \text{ neq } \text{objecty})$ is true and the value of $\text{objecty}_{\text{next}}$ is null at program point 8.
- $((\text{empty}, \text{type}(a_{1\text{init}}) \in A:\text{choose})$, $(\text{global2}, a_{1\text{init}})$ is translated to $((\text{empty}, \text{empty})$, $(\text{global2}, \text{objectx})$ because B, the concrete type of $a_{1\text{init}}$, 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 deflms that imply these bindings. At call sites 5 and 8, these relevant contexts happen to be $(\text{empty}, \text{empty})$. When the actual-to-unknown-initial-value bindings are used for replacing unknown initial values with actuals in a defelm $d1$, each resulting defelm $d2$ 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 3 has recursive types, only a finite number of unknown initial values are accessed in it, so representative names are not needed.

### 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].

Appendix C presents an example that illustrates this advantage of unknown initial values over heap-names.

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 deflms. RCI performs kills for the fields of unknown initial values lazily. During Phase I, when a defelm $d$ 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 $n$ 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 deflms are killed according to the current solution. If not, it unmarks the deflms that should not be killed, and restarting iteration to propagate these unmarked deflms. The propagation stops for a SCC when none of the remaining marked deflms require repropagation (i.e., the marked nodes kill the marked deflms 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 6, 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
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. 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-unknown-initial-value bindings, at pointer assignment statements in methods invoked from M through a series of calls. For example, in Figure 3, the next field of a3init 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-unknown-initial-value 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 a3init is not considered for potential modification at statement 1 of method1, although a3init has the same type as a1init.

Write-only Fields. Consider the following method which is part of class test defined in Figure 3.

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

This method uses the next fields of the unknown initial values prm1init, prm2init and prm3init. Thus, the next fields of prm2init and prm3init should be considered for potential modification at statement 1. However, the next field of prm3init 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 prm3init. This modification will be automatically seen at a call site of method3 where the values of prm1init and prm3init are the same. On the other hand, consider a call site C that invokes method3 and where it is possible for the values of prm1init and prm3init 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 prm1init and prm3init 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 3.

```
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 object1 is a value of the unknown initial value prminit and it is also used in method4 during Phase I. The next field of object1 is read at statement 3 and the type of object1 is the same as the type of prminit. However, at statement 2, when the next field of prminit is modified, the next field of object1 need not be considered for potential modification. This is safe because at statement 6, where object1 is the value of prminit, object1 represents run-time objects created at statement 1 when method4 is called from statement 5; and for the call to method4 at statement 6, object1 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 \( dfeims \) computed during Phase I that represent the values of \( p \) at \( C \). \( RCI \) evaluates the \( dfeims \) of \( S \) by instantiating unknown initial values with their concrete values computed at the entry node of \( M \). Those \( dfeims \) whose relevant contexts evaluate to \textit{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 \textit{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-unknown-initial-value 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-unknown-initial-value 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 \textit{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 \textit{reachable methods}, and then incrementally expands this set during Phase II. For complete programs, the initial set consists of only \textit{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 3 and \textit{method2} (see Section 3.2.1). For simplicity, assume that \textit{method2} is reachable. As a result, \textit{method1} is also reachable as it is invoked from \textit{method2}. At call site 8, \( RCI \) stores an actual-to-unknown-initial-value binding between \textit{object6} and \textit{a1\textsubscript{init}}, and the only relevant context of this binding is \{empty, empty\}. Since the relevant context \{empty, empty\} trivially evaluates to \textit{true}, Phase II propagates \textit{object6} as a concrete value of \textit{a1\textsubscript{init}} to the entry node of \textit{method1}. At the call site in statement 3, the value of the receiver is the value of \textit{a1}, and the value of \textit{a1} is given by the \textit{dfeim} \{empty, empty\}. \textit{Phase II substitutes \textit{object6} for \textit{a1\textsubscript{init}} in this \textit{dfeim} to obtain \textit{object6} as a concrete value of the receiver. This implies that \textit{A::choose} is invocable from statement 3 and the final call graph has an edge from statement 3 to \textit{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 \textit{dfeim} 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 \textit{true} yield the solution of points-to analysis at \( n \). After instantiation with concrete values, a \textit{dfeim} yields a \textit{points-to} of the form \{\textit{var}, \textit{object}\}, where \textit{var} is a local pointer variable, a global pointer variable or a heap-name’s field of pointer type, and \textit{object} is a heap-name.

Consider Phase III for the example in Figure 3 and \textit{method2} (see Section 3.2.1). Suppose, Phase III needs to be done at program point 4. For this, each \textit{dfeim} at program point 4 is instantiated with the concrete values computed at the entry node of \textit{method1}. For example, when \textit{a1\textsubscript{init}} is instantiated by \textit{object6} in the first \textit{dfeim} at program point 4, the relevant context of the \textit{dfeim} evaluates to \textit{true} and the \textit{dfeim} yields the \textit{points-to} \{\textit{global2}, \textit{object6}\}.

\textbf{Demand Table}. The solution computed at a program point by \( RCI \) is sufficient to find the values of variables or fields of unknown initial values used by a method, directly or indirectly through a call. Many important applications like static resolution of dynamic dispatch [PR96] and side-effect analysis [LRZ93] need only this information. However, certain other applications (e.g., detection of dangling pointers in C++) may need values of variables or fields of unknown initial values not used by a method. In order to compute the values of such variables and fields, \( RCI \) stores additional information in a table called \textit{DemandTable}. \( RCI \) uses the \textit{DemandTable} to compute on demand the values of variables and fields of unknown initial values not used by a method. Further details are given in Appendix E.

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 \( RL \) defined in Figure 1, except exceptions, but this does not significantly impact the final results.

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

\textbf{Phase 1}. The scheme for dealing with recursive types ensures that the total number of unknown initial values and hence the total number of possible \textit{dfeims} 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 \textit{dfeim} at a program point and at each step \( RCI \) considers a new \textit{dfeim} at a program point, \( RCI \) always terminates.

Let the total number of unknown initial values generated by \( RCI \) be \( N_{\text{init}} \), the number of user-defined pointer variables

\textsuperscript{9}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.

\textsuperscript{9}For incomplete programs like libraries, the initial set consists of all the methods that could be directly invoked from outside the incomplete program.
be $N_{dfelm}$, the maximum number of fields of a class (including inherited fields) be $F_{max}$, the total number of classes be $N_{class}$, the number of TCFG nodes be $N_{nodes}$, the total number of heap-names be $N_h$ and the bound on the number of conjuncts in a relevant context be $k$.

Let $N_{rc}$ be the number of possible relevant contexts. $N_{rc}$ is at most $O((pa + tc)*)$, where $(pa = 2N_{uiv}, tc = N_{uiv}F_{max})$. 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 $N_{pt}$ be the number of possible points-to, $N_{pt}$ is at most

$$O(fm * sm), \text{ where } \left\{ \begin{array}{l} fm = (N_h + N_{uiv})F_{max} + N_{var} \\ sm = N_h + N_{uiv} \end{array} \right..$$

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 $N_{dfelm}$ be the total number of possible $dfelm$, $N_{dfelm}$ is at most $O(N_{rc}N_{pt})$. Hence the total number of possible $dfelm$ is polynomial in $N_{uiv}, N_{var}, N_h, F_{max}$, 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 $dfelm$ directly generated by the pointer assignment statement. Here $nl$ is an upper bound on the number of elements in $rhs_{rc\text{-loc-pairs}}$ (see Appendix A) and similarly, $nr$ is an upper bound on the number of elements in $rhs_{rc\text{-loc-pairs}}$. $O(N_{uiv} * cs)$, where $(cs = N_hN_{uiv}(N_{uiv} + N_h))$, is an upper bound on the work done in generating $dfelm$ due to potential aliases. Here $cs$ is an upper bound on the number of $dfelm$ in $new\_generated\_dfels$ (see Appendix A) that may generate $dfelm$ due to potential aliases. Finally, $O(nl * cs)$ is an upper bound on the work done in generating $dfelm$ due to potential non-aliases. Here $cs$ is an upper bound on the number of $dfelm$ in the current solution of the pointer assignment node that may generate $dfelm$ 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 $N_{uiv}, N_h, N_{var}, F_{max}$, and $F_{max}$ (assuming $k$ is constant and a constant bound on the number of intraprocedural successors of a node). Let $N_{work}$ 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 $O(N_{dfelm} * N_{nodes} * N_{work})$. Hence the total work done by $RCI$ is polynomial in $N_{uiv}, N_h, N_{var}, F_{max}$, and $N_{nodes}$, assuming $k$ is constant.

$N_{h}, N_{var}, F_{max}$, 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 on $t$ 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 $C_{max}$ be the maximum number of call nodes in a procedure/method. Let $N_{map}$ be the maximum number of actual-to-unknown-initial-value 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-unknown-initial-value mapping. Let $N_{prop}$ 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 $(ce = N_hN_{h}k + N_{h}, cp = N_h)$. 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 $O(ce * cp)$, where $(ce = N_hN_{h}k, cp = N_{h}^2)$. Here $ce$ is the worst case cost of evaluating the relevant context of the $dfelm$ and $cp$ is the worst case cost of evaluating the points-to of the $dfelm$. Hence the worst case cost of this phase is $O(N_hN_{h}k * N_{nodes} * N_{dfelm})$.

A single-level type [CRL98] is either a class all of whose fields are of primitive types or it is an array of a primitive type. For programs with only single-level types and without dynamic dispatch, [CR97] shows that an algorithm similar to $RCI$ computes the precise solution of points-to analysis in $O(n^4)$ worst-case time, where $n$ is roughly the program size. This is under the usual assumption of data-flow analysis: all realizable paths are executable. This is an improvement over the $O(n^7)$ worst-case bound achievable by applying previous techniques [LR91, RHS95] to this case. If dynamic dispatch or fields of pointer type are allowed, points-to analysis is P-space hard [CRL98, Lan92].

4 Exceptions

In this section, we extend $RCI$ for points-to analysis in the presence of exceptions because, unlike C++, exceptions are frequently used in Java programs. In [CRL98, CRL97], we showed how to do points-to analysis of whole programs in the presence of exceptions. The algorithm in [CRL98, CRL97] needs to keep the whole program in memory and cannot analyze incomplete programs. Here, we extend the ideas presented in [CRL98, CRL97] for modular analysis. The semantics of exceptions in Java is discussed in [GJS96, CRL97].

Data Representations. In the presence of exceptions, each $dfelm$ computed by $RCI$ during Phase I has one of the following two forms:

1. $(ECFI, \text{relevant context, points-to})$
2. $(\text{relevant context, exception object})$

A $dfelm$ of the first kind represents a value of a pointer. points-to is as defined in Section 3.2. ECFI or exception control-flow information is one of the following:

- **excp-type**, the run-time type of an exception object,
- **label**, when a finally statement is entered without any uncaught exception, the number of the statement where control should go after exit from the finally statement,
• iv, when the unknown initial value iv is thrown as an exception object or
• empty, for propagation along a path from the entry node of a method to a statement in the method such that the statement is not contained in a finally statement and there is no uncaught exception along the path.

Intuitively, a non-empty ECFI summarizes the control-flow due to uncaught exceptions and finally statements along paths through which a dfelm reaches a program point, and determines the future propagation of the dfelm from the program point. labels are needed for paths that enter a finally statement without an exception: (1) due to falling through the associated try statement or one of the catch statements of the try statement, (2) due to a labelled break or continue [GJS96] in the try statement or one of its catch statements or (3) due to a return in the try statement or one of its catch statements.

A relevant context has the form:

(alias context, type context, EOTC)

alias context and type context are as defined in Section 3.2. EOTC or exception object type context is empty or it is a conjunction of type constraints of one of the following two forms:

1. \((\text{type}(iv) \leq T)\) or
2. \((\text{type}(iv) \not\leq T)\).

The first type constraint says that the associated dfelm holds only in those contexts where the concrete type of the unknown initial value iv is class T or a subtype of T. While the second type constraint says that the the associated dfelm holds only in those contexts where the concrete type of the unknown initial value iv is neither T nor a subtype of T.

A dfelm of the second kind represents an exception object. exception object is either a unknown initial value or a heap-name. The definition of relevant context is the same as in the first case. Intuitively, dfelms of the second kind are needed because they determine the values of the parameters of the catch statements. An exception object is assigned to the parameter of a catch statement that catches the exception object.

As far as exceptions are concerned, compared to [CRL98, CRL97], the use of a unknown initial value as ECFI and EOTC are the new ideas, which enable modular analysis.

Propagation of dfelms. As before, during Phase I, RCI traverses SCC-DAG in a reverse topological order and analyzes each method in terms of unknown initial values. We will use the example in Figure 4 to briefly explain how RCI propagates the above dfelms during this phase. Details are similar to those given in Section 3.2. Figure 4 shows a part of the solution computed by RCI before each of the statements 2, 3, 4, 5 and 6. For simplicity, we ignore the dfelms representing the value of p.

The two dfelms at program point 2 represent the values of e. Statement 2 throws the object to which e points as exception object. As a result, 3:1 to 3:4 are propagated to statement 3, the exit node of the try statement. The dfelm 3:1 represents a value of e. The ECFI ET2 means that 3:1 reaches program point 3 from the entry node of method8 along paths that have an uncaught exception of run-time type ET2. The relevant context (empty,empty,empty) means that 3:1 holds in all contexts. 3:1 is created from 2:1 when object1 is thrown at statement 2. Similarly, 3:2 also represents a value of e. It is created when pinit is thrown at statement 2. pinit is the ECFI because the run-time type of the exception object depends upon the type of concrete value of pinit and could be any subtype of the declared type of pinit. When a dfelm that has a unknown initial value as ECFI is propagated from the exit node of a method to a call site that invokes the method, the values of the unknown initial value at the call site are used to determine the types of the exception. 3:5 also represents a value of e and the ECFI empty means that 3:5 reaches program point 3 along paths without any uncaught exceptions. 3:3 and 3:4 represent the exception objects thrown at statement 2. Again, the relevant context (empty,empty,empty) means that 3:3 and 3:4 hold in all contexts.

Now consider the four dfelms at program point 4. RCI uses the ECFIs of the dfelms at the exit node of a try statement to decide how to propagate the dfelms. The ECFI of 3:1 is ET2 and exceptions of type ET2 or any subtype of ET2 are caught by the catch statement. So 3:1 is propagated to the entry node of the catch statement and 4:1 is generated from it because the uncaught exception of type ET2 has been caught by the catch statement. Similarly, since the exception object pinit is caught by the catch statement if and only if the type of the concrete value of pinit is ET2 or a subtype of ET2, 4:2 generates an uncaught exception of run-time type ET2. So it is propagated to the entry node of the catch statement, where it is used to instantiate the parameter of the catch statement. As a result, 4:3 is generated. Similarly, 4:4 is generated from 3:4.

Next, consider the four dfelms at program point 5. 3:5 reaches program point 3 along a path without any uncaught exception. So 3:5 should propagate to program point 6. However, the finally statement must be executed no matter how the try statement terminates, with an exception or without an exception. RCI records the fact that 3:5 has to be propagated to program point 6 after the finally statement is executed as ECFI 6 and the result is 5:1. Similarly, 4:1 also generates 5:1 and 4:2 generates 5:2. Since, the exception object pinit is not caught by the catch statement if and only if the type of the concrete value of pinit is neither ET2 nor a subtype of ET2, 3:2 generates 5:3. Similarly, 3:4 generates 5:4.

Finally, consider the four dfelms at program point 6. These are generated respectively from the four dfelms at program point 5 in the obvious way.

A call or try statement inside a finally statement can cause ECFIs to stack up. At a call node inside a finally statement, with each relevant context r of an actual-to-unknown initial value binding, RCI also remembers the corresponding ECFIs of the dfelms that imply the binding and have r as their relevant context. When the actual is substituted for the unknown initial value in a dfelm d propagated from the exit node of the callee to the call site, the ECFIs of the actual-to-unknown initial value binding are used as ECFIs of the dfelms resulting from the substitution. However, if d already has an ECFI that represents an uncaught exception, then, according to the semantics of Java [GJS96], the ECFI of d overrides the ECFIs of the actual-to-unknown initial value binding. This is the reason why one ECFI is sufficient. A try statement nested inside a finally statement is treated like a call to an anonymous procedure.

Phases 0, II and III of RCI are same as in the absence of
exceptions. Also, Theorem 1 is true for the new definition of relevant context. Moreover, under the assumptions explained in Section 3.5, RCI is polynomial-time even in the presence of exceptions. The hardness of points-to analysis in the presence of exceptions is discussed in [CRL98, CRL97].

We have considered only exceptions generated by throw statements; since run-time exceptions can be generated by almost any statement, we have ignored them. Our algorithm can handle run-time exceptions if the set of statements that can generate these exceptions is given as an input. If all statements that can potentially generate run-time exceptions are considered, we will get a safe solution; however, this may generate far more information than what is useful.

5 Analysis and Testing of Libraries

In this section, we explain how RCI analyzes incomplete programs like libraries and how alias context, type context and EOTC computed by RCI can be used for unit testing of libraries.

Analysis of Libraries. Let $M$ be a method in a library $L$. $M$ is called a public interface method of $L$ if and only if $M$ is in the public interface of $L$ or it overrides a method in the public interface of $L$. A public interface method can be directly invoked from a call site outside the library. A unknown initial value at the entry node of a public interface method is called an interface initial value. During Phase II, RCI treats an interface initial value like a concrete value and propagates it to the entry nodes of other methods if it is the value of an unknown initial value at a call site. During Phase II or III, when RCI instantiates an unknown initial value in a conjunct with an interface initial value, it makes conservative, worst-case assumptions about the interface initial value for evaluating the conjunct.

The above scheme is for those situations in which the points-to solution for a library is needed without the availability of any driver (e.g., for optimizing the library before shipping it). Another possibility is not to perform Phase II and Phase III on the library until drivers are available. In this case, different drivers can share the Phase I solution of the library and the cost of doing Phase I on the library will be amortized over the drivers. This is a big advantage because Phase I is the costliest step and Phase II and Phase III are quite fast (see Section 6).

The above schemes compute a safe solution (for points-to analysis) with respect to a driver program if at each virtual function invocation site in the library, one of the following two conditions holds:

- none of the receivers computed by RCI is an interface initial value, or
- for each interface initial value that is a possible receiver, the subtypes of the declared type of the interface initial value defined in the driver program do not override the method invocable according to the declared type of the interface initial value;

and, at each call site in the library that makes a call through a function pointer, one of the following two conditions holds:

- none of the values of the function pointer computed by RCI is an interface initial value, or
- an interface initial value is a possible value of the function pointer, but for each function whose address is taken in the driver program, either (i) the signature of the function does not match the signature of the functions that can be invoked through the function pointer or (ii) the address of the function is also taken in the library.

There are many such situations in practice where a library is used like a component and the above scheme is adequate. Also, in Java, final methods and methods of classes not in the public interface cannot be overridden by a driver. Moreover, the above two restrictions can be easily and automatically checked to see if the solutions computed by the above schemes are safe for a driver.

When an interface initial value is a possible receiver at a virtual function invocation site in a library and one of the subtypes of the declared type of the interface initial value defined in a driver overrides the method invocable from the call site according to the declared type of the interface initial value\(^{10}\), then a method that is not known while analyzing the library can be invoked from a call site in the library. In this case, after the call site, one of the possible values of each variable or field that can be modified by the unknown overriding method or UOM is a TUobj (yped unknown object) of the declared type of the variable or field. There is one TUobj for each type. RCI makes conservative, worst-case assumptions about TUobj’s while analyzing the library.

At a call site $C$ that invokes a public interface method from a driver, a TUobj of type $T$ maps to all the heap-names whose allocation sites are reachable from $C$ and whose types are $T$ or a subtype of $T$. Note that at a call site that can invoke a UOM, only global variables and fields of objects accessible through the values of the globals and parameters at the call site can be modified or read by the UOM. Moreover, the library writer can use his/her knowledge of the semantics of the called methods to provide stylized annotations about globals and type signatures of fields that can be modified or read by UOM’s. The type signature of a field $f$ is a pair of the name of the class that defines $f$ and $f$’s own name. Among the globals and fields of unknown initial values and heap-names that can be modified by an UOM, only the globals mentioned in the annotations and the fields whose type signatures are mentioned in the annotations need to point to TUobj’s after a call site that can invoke the UOM. The implementation of TUobj’s, the investigation of their impact on precision and the use of annotations to increase precision are part of future work.

Application to Testing of Libraries. Alias context, type context and EOTC 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; such information cannot be obtained using a whole-program-analysis technique on a particular driver and the library. These contexts suggest a new coverage measure for the unit testing of libraries:

- $k_1 k_2 k_3$-level relevant context cover or $k_1 k_2 k_3$-RCC,

which can be used in addition to other standard coverage measures [Bei90, RW85]. Let $p$ be an execution path from the entry node of a public interface method $M$ of a library $L$ to a statement $s$ of $L$, not necessarily in $M$, such that $p$ does not return to the call site in a driver program from which $p$ starts. Let $z$ be a variable, a field of an unknown initial value of $M$ or a field

\(^{10}\) or an analogous situation exists for a call site that calls through a function pointer.
class ET1 : public Exception { };
class ET2 : public ET1 { };
class ET3 : public ET2 { };

void method8 ( ET1 *p ) {
    ET1 *e = new ET2();
    try {
        if ( _ ) {
            if ( _ ) e = p;
            2: throw e;
        }
    3: }
    catch ( ET2 *excp ) {
        4: }
    finally {
        5: }
    6: }
};

Figure 4: Exceptions

of a heap-name such that x is of reference type and x is read at s when p is executed. An relevant context z is called the contextual support of a points-to (x,y) with respect to p if and only if the following two conditions hold:

1. If z holds at the entry node of M, then when p is executed, x points to y at s.

2. Let t be any relevant context properly contained in z (i.e., the set of conjuncts of t is a proper subset of the set of conjuncts of z). Then there exists a relevant context u such that if t and u simultaneously hold at the entry node of M, when p is executed, x does not point to y at s.

Intuitively, z is a minimal relevant context such that if z holds at the entry node of M, then when p is executed, s points to y at s.

Let S be a set whose each element is either empty or it is a conjunction of potential aliases, potential non-aliases and type constraints. S is called a k1k2k3-RCC of L if and only if for each contextual support cs of L, there exists an element e of S such that the following conditions hold:

1. Each potential alias of e is contained in cs.
2. Each potential non-alias of e is contained in cs.
3. Each type constraint of e is contained in cs.
4. Either each potential alias of cs is contained in e or e has at least k1 potential aliases.
5. Either each potential non-alias of cs is contained in e or e has at least k2 potential non-aliases.
6. Either each type constraint of cs is contained in e or e has at least k3 type constraints.

Intuitively, a k1k2k3-RCC says which conjunctions of potential aliases, potential non-aliases and type constraints with at most k1 potential aliases, k2 potential non-aliases and k3 type constraints are relevant for points-to associations in a library and hence should be exercised during unit testing.

RCI can be easily adapted for computing a k1k2k3-RCC. This is explained in Appendix F.

The following example illustrates relevant context cover. method9 is part of class test defined in Figure 3.

void method9( A *a1, A *a2, A *a3 ) {
    A *t;
    1: a1--next = a3;
    2: l = a2--next;
}

Let p be the path from the entry node of method9 to statement 2. x is a2next, next and y is a3next, then z is (a1next eq a2next). For this method, 
{(a1next eq a2next), empty} is a k1k2k3-RCC for any k1, k2 and k3. Apart from a1next and a2next, potential aliasing between any other unknown initial values is not relevant for unit testing of this method, and as a result such irrelevant aliases do not appear in any CS. However, aliases that are irrelevant for unit testing of method9 may be relevant for a driver program. For example, if the values of a1init and a3init are same for a driver, a3init, next will be modified by method9. However, whether this modification has any effect on the rest of the driver program depends on the semantics of the driver and cannot be tested without the driver. For unit testing of method9, test cases must be generated with a1init and a2init aliased. However, generating a test case with any other unknown initial values aliased (e.g, a1init and a3init) is not needed. Thus, RCI can help in the generation of relevant test cases.

The alias contexts, type contexts and EOTCs computed by RCI can also be used by program understanding tools. RCI can help by uncovering unexpected but relevant potential aliases, potential non-aliases and type constraints. Moreover, the complexity of alias contexts, type contexts and EOTCs define a measure of code complexity. Complicated alias contexts, type contexts or EOTCs point out portions of code that are hard to maintain and understand.
6 Implementation

Our implementation has been built using the PROLANGS Analysis Framework (PAF) which incorporates the Edison Design Group front end for ANSI C++.\(^\text{11}\) 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].\(^\text{12}\) 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, \(t_{1+2}\) 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;

\(^{11}\) See [http://www.prolangs.rutgers.edu/public.html](http://www.prolangs.rutgers.edu/public.html).

\(^{12}\) 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, smatrix and vector 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.
Table 1: Benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>Ph 0</th>
<th>Ph 1</th>
<th>Ph 2</th>
<th>Ph 3</th>
<th>Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>trees</td>
<td>0.01</td>
<td>0.04</td>
<td>0.05</td>
<td>0.03</td>
<td>(1,1)</td>
</tr>
<tr>
<td>deriv1</td>
<td>0.02</td>
<td>0.06</td>
<td>0.05</td>
<td>0.06</td>
<td>(1,1)</td>
</tr>
<tr>
<td>deriv1</td>
<td>0.01</td>
<td>0.07</td>
<td>0.05</td>
<td>0.03</td>
<td>(1,1)</td>
</tr>
<tr>
<td>employ</td>
<td>0.02</td>
<td>0.14</td>
<td>0.16</td>
<td>0.19</td>
<td>(1,1)</td>
</tr>
<tr>
<td>richards</td>
<td>0.09</td>
<td>0.14</td>
<td>0.16</td>
<td>0.19</td>
<td>(1,1)</td>
</tr>
<tr>
<td>deltable</td>
<td>0.03</td>
<td>0.17</td>
<td>0.18</td>
<td>0.20</td>
<td>(1,1)</td>
</tr>
<tr>
<td>sampleAdv</td>
<td>0.03</td>
<td>0.17</td>
<td>0.18</td>
<td>0.20</td>
<td>(1,1)</td>
</tr>
<tr>
<td>vmatrix</td>
<td>0.04</td>
<td>0.17</td>
<td>0.18</td>
<td>0.20</td>
<td>(1,1)</td>
</tr>
<tr>
<td>vvector</td>
<td>0.05</td>
<td>0.17</td>
<td>0.18</td>
<td>0.20</td>
<td>(1,1)</td>
</tr>
<tr>
<td>opProd</td>
<td>0.08</td>
<td>0.17</td>
<td>0.18</td>
<td>0.20</td>
<td>(1,1)</td>
</tr>
<tr>
<td>penguin</td>
<td>0.08</td>
<td>0.17</td>
<td>0.18</td>
<td>0.20</td>
<td>(1,1)</td>
</tr>
<tr>
<td>Bdecay</td>
<td>0.08</td>
<td>0.17</td>
<td>0.18</td>
<td>0.20</td>
<td>(1,1)</td>
</tr>
<tr>
<td>electron</td>
<td>0.08</td>
<td>0.17</td>
<td>0.18</td>
<td>0.20</td>
<td>(1,1)</td>
</tr>
</tbody>
</table>

Table 2: Timings in Seconds

<table>
<thead>
<tr>
<th>Program</th>
<th>PA’s</th>
<th>TC’s</th>
<th>Max Size</th>
<th>Ave Size</th>
<th>Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>trees</td>
<td>4</td>
<td>12</td>
<td>1</td>
<td>8.7%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>deriv1</td>
<td>0</td>
<td>116</td>
<td>3</td>
<td>6.4%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>deriv1</td>
<td>0</td>
<td>116</td>
<td>1</td>
<td>6.3%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>employ</td>
<td>0</td>
<td>12</td>
<td>1</td>
<td>5.8%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>richards</td>
<td>85</td>
<td>5</td>
<td>3</td>
<td>9.0%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>richards</td>
<td>84</td>
<td>5</td>
<td>2</td>
<td>9.0%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>richards</td>
<td>77</td>
<td>5</td>
<td>1</td>
<td>9.0%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>deltable</td>
<td>1</td>
<td>12</td>
<td>1</td>
<td>3.9%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>sampleAdv</td>
<td>11</td>
<td>5</td>
<td>2</td>
<td>7.4%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>vmatrix</td>
<td>10</td>
<td>6</td>
<td>2</td>
<td>7.2%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>vvector</td>
<td>10</td>
<td>6</td>
<td>2</td>
<td>7.2%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>FeynLib</td>
<td>115</td>
<td>535</td>
<td>1</td>
<td>3.5%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>FeynLib</td>
<td>245</td>
<td>579</td>
<td>2</td>
<td>3.7%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>opProd</td>
<td>115</td>
<td>535</td>
<td>1</td>
<td>3.5%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>opProd</td>
<td>245</td>
<td>579</td>
<td>2</td>
<td>3.7%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>penguin</td>
<td>115</td>
<td>535</td>
<td>1</td>
<td>3.5%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>penguin</td>
<td>245</td>
<td>579</td>
<td>2</td>
<td>3.7%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>Bdecay</td>
<td>115</td>
<td>535</td>
<td>1</td>
<td>3.5%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>Bdecay</td>
<td>245</td>
<td>579</td>
<td>2</td>
<td>3.7%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>electron</td>
<td>115</td>
<td>535</td>
<td>1</td>
<td>3.5%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>electron</td>
<td>245</td>
<td>579</td>
<td>2</td>
<td>3.7%</td>
<td>(1,1)</td>
</tr>
</tbody>
</table>

Table 3: Performance Data

<table>
<thead>
<tr>
<th>Program</th>
<th>PA’s</th>
<th>TC’s</th>
<th>Max Size</th>
<th>Ave Size</th>
<th>Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>trees</td>
<td>4</td>
<td>12</td>
<td>1</td>
<td>8.7%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>deriv1</td>
<td>0</td>
<td>116</td>
<td>3</td>
<td>6.4%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>deriv1</td>
<td>0</td>
<td>116</td>
<td>1</td>
<td>6.3%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>employ</td>
<td>0</td>
<td>12</td>
<td>1</td>
<td>5.8%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>richards</td>
<td>85</td>
<td>5</td>
<td>3</td>
<td>9.0%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>richards</td>
<td>84</td>
<td>5</td>
<td>2</td>
<td>9.0%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>richards</td>
<td>77</td>
<td>5</td>
<td>1</td>
<td>9.0%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>deltable</td>
<td>1</td>
<td>12</td>
<td>1</td>
<td>3.9%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>sampleAdv</td>
<td>11</td>
<td>5</td>
<td>2</td>
<td>7.4%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>vmatrix</td>
<td>10</td>
<td>6</td>
<td>2</td>
<td>7.2%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>vvector</td>
<td>10</td>
<td>6</td>
<td>2</td>
<td>7.2%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>FeynLib</td>
<td>115</td>
<td>535</td>
<td>1</td>
<td>3.5%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>FeynLib</td>
<td>245</td>
<td>579</td>
<td>2</td>
<td>3.7%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>opProd</td>
<td>115</td>
<td>535</td>
<td>1</td>
<td>3.5%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>opProd</td>
<td>245</td>
<td>579</td>
<td>2</td>
<td>3.7%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>penguin</td>
<td>115</td>
<td>535</td>
<td>1</td>
<td>3.5%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>penguin</td>
<td>245</td>
<td>579</td>
<td>2</td>
<td>3.7%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>Bdecay</td>
<td>115</td>
<td>535</td>
<td>1</td>
<td>3.5%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>Bdecay</td>
<td>245</td>
<td>579</td>
<td>2</td>
<td>3.7%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>electron</td>
<td>115</td>
<td>535</td>
<td>1</td>
<td>3.5%</td>
<td>(1,1)</td>
</tr>
<tr>
<td>electron</td>
<td>245</td>
<td>579</td>
<td>2</td>
<td>3.7%</td>
<td>(1,1)</td>
</tr>
</tbody>
</table>
Table 4: Applications

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

Figure 5: Differences in Precision between RCI and Hierarchy Analysis
[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 interprocedural type propagation and context-insensitive interprocedural type propagation. [GGDC97] 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.

8 Conclusion and Future Work

We have presented a new technique called relevant context inference or RCI for modular data-flow analysis of substantial subsets of C++ and Java. We have explained RCI in the context of modular points-to analysis. We have shown that RCI can do points-to analysis even in the presence of exceptions. We have also shown that RCI can analyze incomplete programs like libraries. Further, we have presented a new coverage measure for unit testing of libraries and shown that the information computed by RCI can be used for generating relevant test cases for unit testing of libraries. Finally, we have presented empirical evidence for the effectiveness of RCI.

We plan to run RCI on larger (20000 lines) programs in future. We also plan to use RCI for solving other data-flow analysis problems, such as finding def-use associations. We want to explore flow-insensitive but context-sensitive variants of RCI as well.

RCI can also help in parallelizing data-flow analysis. Phases I and II can be done in parallel on SCC’s that do not have any dependence on each other. Phase III can be done in parallel for each node at which the solution for data-flow analysis is needed. Exploration of these issues is part of future work.

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

References


A Pointer Assignment

The data-flow transfer function, apply, for a pointer assignment node \( n \), is defined in Figures 6 and 7. \( n.lhs \), \( n.rhs \) and \( n.sol \) are respectively the left hand side expression, the right hand side expression and the current solution at \( n \). \( \text{rdfe} \) is a new \( \text{dfelm} \) \((rc, (x, y))\) reaching the node \( n \) (i.e., \( \text{rdfe} \not\in n.sol \)). \( \text{lhs}_{\text{rc_loc_pairs}} \) is the set of pairs of relevant contexts and objects that are modified by node \( n \). Note that if \( n.lhs \) has a dereference, \( \text{lhs}_{\text{rc_loc_pairs}} \) is initially \( \emptyset \). Otherwise, it is initially \( \{ (\text{empty}, \text{empty}), n.lhs \} \). Consider the following example. Let \( n.lhs \) be \( p := f 1 \). A \( \text{dfelm} \) \((rc, (x, y))\) can imply a new \( \text{lhs}_{\text{rc_loc_pairs}} \) if and only if \( x \) is \( p \). Thus if the on top solution for this assignment was

\[
\{(\text{empty}, \text{empty}), (p, l)\} \\
\{(\text{init}, \text{eq} \text{init}), \text{empty}, (p, p \text{init})\} \\
\{(\text{empty}, \text{empty}), (q, m)\}
\]

then \( \text{lhs}_{\text{rc_loc_pairs}} \) would be

\[
\{(\text{empty}, \text{empty}), (p, l), \\
\{(\text{init}, \text{eq} \text{init}), \text{empty}, (p, p \text{init})\}\}
\]

If \( n.lhs \) does not have dereference (i.e., \( n.lhs \) is a variable name \( p \)), then \( \text{lhs}_{\text{rc_loc_pairs}} \) contains a single element \( \{ (\text{empty}, \text{empty}), p \} \) and new \( \text{lhs}_{\text{rc_loc_pairs}} \) is always \( \emptyset \). \( \text{rh}_{\text{rc_loc_pairs}} \) is very similar to \( \text{lhs}_{\text{rc_loc_pairs}} \) except it is for \( n.rhs \) (i.e., the set of pairs of relevant contexts and objects that are values of \( n.rhs \)).

\( \text{potAliases} \) computes \( \text{dfelm} \)s that need to be generated due to potential aliases. \( \text{potNonAliases} \) computes \( \text{dfelm} \)s that need to be generated due to potential non-aliases.

The if statement at program point 3 checks if \( rdfe \) is killed by node \( n \). \text{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 \( p := f 1 \). \( rdfe \) represents the value of \( f 1 \) 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 \( A^* \) and the type of \( s \) are compatible). In this case, \( \text{potNonAliases} \) is called to generate \( \text{dfelm} \)s that condition the propagation of \( rdfe \) across \( n \) on potential non-aliases. These \( \text{dfelm} \)s 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.
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_generated_dfes ∪ {new_dfe};
      }
   }
}

2: for (each rc2, v) in new_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_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;
}

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
if (u is an unknown initial value)
   {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 = new_dfe ∪ {new_dfe};
      } } return generated_dfes;
}

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
if (v is an unknown initial value whose type is compatible with A)
   {for (each (rc1, (u.f1, y)) in dfes such that u is a
unknown initial value whose type is compatible with A)
   if (v is an unknown initial value compatible with u) {
      rc3 = rc1 and rc2 and (u neq v);
      rc4 = impose user defined bounds on rc3;
      new_dfe = (rc4, (u.f1, y));
      generated_dfes = new_dfe ∪ {new_dfe};
   } else {
      generated_dfes ∪ {(rc1, (u.f1, y))};
   } return generated_dfes;
}

Figure 7: apply2

Figure 6: apply1
B Complexity of RCI

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

```c
class A_i { 
    public: A_{i+1} *f_i; 
    public: A_i *p_i; 
    ...
};
```

```c
void method(A_i *a) { 
    A_i *p_i;
    ...
    A_{i+1} *f_i; 
    A_i *p_i; 
    ...
    ...
    ...
    ...
    A_{i-1} {
        public: A_{i-1} *f_i;
        public: A_i *p_i;
        ...
    }
    ...
    ...
    class A_n { };
    ...
    ...
    ...
    }
}```

C Strong Update

First, Consider \texttt{method6} which is part of class \textit{test} defined in Figure 3.

```c
void method6() { 
    A *p,*q,*r;
    1: p = new A();
    do { 
        2: r = new A();
        if ( ) p = r;
        q = r;
    } while(.);
    3: p→next = new A();
    4: q→next = new A();
}
```

At program point 3, both \( p \) and \( q \) may point to the same heap-name \textit{object2}; however, they may point to different run-time objects. Thus, the assignment to the \textit{next} field of \textit{object2} at statement 3 cannot be killed by the assignment to the same field at statement 4.

Next, let us consider a slightly modified version of \texttt{method6}.

```c
void method7( A *prm ) { 
    A *p,*q,*r;
    1: p = new A();
    do { 
        2: r = prm;
        if ( ) p = r;
        q = r;
    } while(.);
    3: p→next = new A();
    4: q→next = new A();
}
```

In \texttt{method7}, at program point 3, both \( p \) and \( q \) may point to the same unknown initial value \textit{prmInit}. Although for different calls to \texttt{method7}, \textit{prmInit} may represent different run-time objects, for a given call to \texttt{method7}, \textit{prmInit} represents the same run-time object throughout the method. Thus, the assignment to the \textit{next} field of \textit{prmInit} at statement 3 can be safely killed by the assignment to the same field at statement 4.

\footnote{For clarity this example is not written in RL.}
D Recursive Types

space(uv). The unknown initial values defined with respect to the unknown initial value uv of a parameter or a global variable comprise the space of initial values associated with uv. space(uv) denotes this space of initial values associated with uv. For example:

class E {}
class D {
  public: F *f1;
  public: E *f;
};
class F {
  public: F *f2;
};

proc(D *a) {
  space(a1) consists of a1init, a1-f1init, a1-f1init-f1init, and a1-f1init-f2init.
  In the presence of recursive types, space(uv) can be infinite as shown in the following example,

class H {
  class G {
    proc(G *p) {
      public: G *p;
      G *parent;
      H *child;
    }
  }
  public: H *parent;
  public: F *field2;
};

class E {
  public: F *field1;
  public: F *field2;
};

where space(pinit) consists of pinit, pinit-childinit, pinit-childinit-parentinit, pinit-childinit-parentinit-childinit and so on.

Representative names. Since RCI potentially needs to represent any arbitrary element of a space(unknown initial value), in the presence of recursive types, RCI divides an infinite space(unknown initial value) into a finite number of possibly intersecting subsets. All the elements in a subset are represented by a single, representative name or rep. Any dfelm involving a rep represents a set of dfelm's containing one dfelm for each instantiation of the rep by a member of the corresponding subset.

To intuitively describe the algorithm used by RCI to generate a rep, consider a unknown initial value pinit, f1init, f2init, ..kinit in space(pinit). To find the corresponding rep, RCI considers this unknown initial value as a path through the infinite tree of unknown initial values which could be possibly constructed from the type of the root unknown initial value, i.e., pinit. Each internal node has as its children, the unknown initial values of all the fields of the type of the internal node. A path through this tree to a node is an unknown initial value of the above form. RCI essentially collapses specific paths in this tree which end in the same type to one path, thus handling any recursion in the type definition. There are many ways of performing this collapse; RCI's choice is just one of them.

All the unknown initial values represented by a rep have the same type. Subsets which contain elements of the same type are distinguished by the field selectors used in the construction of their elements. As a rule of thumb, given a unknown initial value pinit, f1init, f2init, ..kinit, RCI considers the tree in which this unknown initial value is a path and starting at the root, traverses the path corresponding to the unknown initial value by selecting each field fi in turn, for i = 1..k. RCI builds the corresponding rep during this tree traversal by discarding repetitive subpaths it explores. Given the selected field fi, RCI looks forward on the path until it finds the last field (closest to the end of the path) with the same type as the type of fi, say it is fj. Then RCI skips fi1init, ..fiinit and restarts traversal from fjinit. This process continues until RCI reaches the end of the path. Now RCI has the rep for the original unknown initial value.

A rep of a unknown initial value pinit, f1init, f2init, ..finit has the form v0, v1, .., vl, where

- vl is pinit or [pinit] and
- for each vi, i = 1..l, there exist a l ≤ k such that vi is finit or [finit].

Here [x] means a subpath starting at x was collapsed. Note that by construction each vi, i = 0..t, has a distinct type. This ensures that the number of rep's is finite.

Mapping between actuals and unknown initial values. Let approx-space(uv) be the approximation of space(uv) constructed by RCI using rep's and unknown initial values. Here uv is the unknown initial value of a global or a parameter at the entry node of a method M. Since approx-space(uv) contains rep's, the elements of approx-space(uv) can form cycles. At a call site that invokes M, RCI uses the edges (i.e., associations between fields and what they point to) between the elements of approx-space(uv) to determine the actuals that bind to a node of approx-space(uv), which could be a unknown initial value or a rep. Note that as described in Section 3.2.3, RCI only uses the edges (or fields) that are used in M. Starting at a binding between an actual and the root node of approx-space(uv), for each binding b between an actual v and a node s of approx-space(uv), RCI recursively computes the bindings between the the nodes of approx-space(uv) to which the fields of s point and the values of the corresponding fields of v. Since the elements of approx-space(uv) form a general directed graph rather than a tree, a node of approx-space(uv) can map to multiple actuals.

Example. The example in Figure 8 shows how rep's are used to represent the summary transfer function of a method.

The dfelm ((empty, empty), (paraminit, f2, xinit)) is part of the summary transfer function of proc1 and RCI propagates it first to program point 10 and from there to program point 9. paraminit is a rep and the subset represented by it binds to {object4, object5} at the call site 8. Figures 9 and 10 show the bindings between the elements of approx-space(paraminit) and the actuals at call site 8. As a result, this dfelm expands to {((empty, empty), (object4,f2, xinit)), ((empty, empty), (object5,f2, xinit))} at program point 9.

E Demand Driven Computation

As stated in Section 3.2.3, RCI generates fields of unknown initial values lazily. Similarly, it also considers globals and parameters lazily, only when they are found to be used. During Phase III, at a call node C, for each points-to pt that represents a value of a field of a heap-name at C, the tuple (pt, C) is stored in a global table called DemandTable. Similarly, for each points-to that represents a value of a global variable that is not used in at least one of the methods invoked from C, the tuple (pt, C) is stored in DemandTable. The following example shows how RCI uses DemandTable to compute on demand the values of variables and fields of unknown initial values not used by a method.

class F {
};
class E {
  public: F *field1;
  public: F *field2;
};

14 For clarity this example is not written in RL.
15 For clarity this example is not written in RL.
class A1 { public: B1 *f1; public: C1 *f1; public: B1 *f2; public: A1 *f2; public: A1( ) { public: B1( ) { f1 = f2 = 0; f1 = f2 = 0; } } public: static void foo0( E *e1 ) public: static void foo1( E *e2 ) public: static void main( void ) { public: A1( ) public: B1 *f2; public: A1 *f2; public: B1 *f1; public: C1 *f1; } public: static void proc0( ) { public: static void proc1( A1 *param ) { A1 *tmp; tmp = param; while( tmp != 0 ) { tmp = tmp->f2 = x; if ( tmp->f1 != 0 ) tmp = tmp->f1->f2; else break; } } return; } }

class test1 {
public: static B1 *x;
public: static void proc0( ) {
A1 *p;
B1 *q;
1: p = new A1();
2: q = new B1();
3: p->f1 = q;
4: p->f1->f2 = new A1();
5: q = new B1();
6: p->f1->f2->f1 = q;
7: p->f1->f2->f1->f2 = new A1();
8: proc1( p );
9:
10:
}

class test1 {
public: static void foo0( E *e1 ) {
F *p;
5: p = e1->field1;
6: }
public: static void foo1( E *e2 ) {
7: foo0( e2 );
}
public: static void main( void ) {
E *local;
1: local = new E();
2: local->field1 = new F();
3: local->field2 = new F();
4: foo1( local );
};
}

Phase I computes \((\text{empty}, \text{empty}), (p, e1_{init}.field1_{init})\) and \((\text{empty}, \text{empty}), (e2_{init}.field1, e2_{init}.field1_{init})\) at program points 6 and 7 respectively; and \(e1_{init}.field2_{init}\) and \(e2_{init}.field2_{init}\) do not appear in the Phase I solutions of \(\text{foo0}\) and \(\text{foo1}\) respectively. As a result, during Phase II, the value of \(e2_{init}.field2_{init}\) at the call site 4, which is \(\text{objects}_3\), is propagated to the entry node of \(\text{foo1}\). Instead, the tuple \((\text{objects}_3.field2, \text{objects}_3)\), 4 is stored in \(\text{DemandTable}\). If the value of \(e1_{init}.field2_{init}\) is needed at program point 5, then, since \(e1_{init}\) is \(\text{objects}_1\), \(\text{RCI}\) looks up \(\text{DemandTable}\) for the values of \(\text{objects}_1.field2\). The result of this lookup says that \(\text{objects}_1.field2\) points to \(\text{objects}_3\) at the call site 4. Now \(\text{RCI}\) checks if the entry node of \(\text{foo0}\) is reachable from call site 4. Since it is reachable, \(\text{objects}_3\) is a valid value for \(\text{object1}.field2\) at program point 5.

**Context-sensitive Transitive Closure** Checking reachability of an entry node from a call site is easy in the absence of dynamically dispatched calls as it reduces to computing the transitive closure of the call graph. However, in the presence of dynamically dispatched calls, the transitive closure of the call graph gives an approximate, but safe solution to the above question. This is because in the presence of dynamic dispatch, reachability is context-sensitive and hence non-transitive. In order to improve precision, \(\text{RCI}\) uses the following scheme for computing the context-sensitive transitive closure of the final call graph. It traverses the SCC-DAG in a reverse topological order (bottom-up) and for each reachable method (see Section 3.3) \(m\), it computes the set of methods (say \(\text{reach-set} \)) reachable from \(m\) and relevant contexts, in terms of the unknown initial values of \(m\), under which these methods are reachable. \(\text{RCI}\) uses the final call graph for determining the set of methods invocable from a call site. When the set of methods reachable from a call node \(C\) in a reachable method \(m\) is needed, it is computed on demand using the \(\text{reach-sets}\) of the methods invocable from \(C\). The relevant contexts of the elements of these \(\text{reach-sets}\) are translated by instantiating the unknown initial values of the methods invocable from \(C\) with their values at \(C\). The resulting relevant contexts are then evaluated using the concrete value’s computed at the entry node of \(m\). The set of methods reachable from \(C\) is determined by those elements whose relevant contexts evaluate to \(\text{true}\). The pseudocode for context-sensitive transitive closure is given in [CRL97]. Consider the example given in Figure 3. For the sake of illustration, let us assume that the final call graph has edges from call site 3 to \(A::\text{choose}\) and \(C::\text{choose}\). Under this assumption, the \(\text{reach-set}\) of method1, say \(rs\), is \{ \((\text{empty}, (\text{type}(a1_{init}) \in A::\text{choose})), A::\text{choose})\), \((\text{empty}, (\text{type}(a1_{init}) \in C::\text{choose})), C::\text{choose})\} \}

Now, suppose the set of methods reachable from call site 5, in method2 (defined on page 4), is needed. The only
method invocable from call site 5, in the final call graph, is method1. Thus, the unknown initial values of method1 in the relevant contexts of the elements of rs are instantiated by their values at call site 5, i.e., a1_init is replaced by a4_init. This yields the set rs1:

\[
\{ (\langle\text{empty}, (\text{type}(a4_init) \in A::\text{choose})\rangle, A::\text{choose}),
\langle\text{empty}, (\text{type}(a4_init) \in C::\text{choose})\rangle, C::\text{choose} \}
\]

Now a4_init is instantiated by its concrete value’s at the entry node of method2, and the methods reachable from call site 8 are needed. Now, a1_init in the relevant contexts of the elements of rs is instantiated by object6. This means that the relevant context of the first element of rs evaluates to true and the relevant context of the second element evaluates to false. Hence, the set of methods reachable from call site 8 is

\[
\{ \text{method1, A::choose} \}
\]

F Testing of Libraries

If RCI is used for computing only points-to’s, when a dfelm \(d_1\) reaches a program point \(n\), if the points-to contained in \(d_1\) is contained in another dfelm \(d_2\) that is already present in the current solution of \(n\) and the relevant context of \(d_2\) is contained in the relevant context of \(d_1\), \(d_1\) is ignored. However, \(d_2\) could have been due to a non-executable path. This may cause RCI not to compute relevant context covers. This can be avoided by propagating newly reaching dfelm’s like \(d_1\) as long as their relevant contexts obey the bounds imposed on the numbers of potential aliases, potential non-aliases and type constraints. With this modification, the following assertion about RCI is true. We need some notations to express this assertion. Let \(c\) be any conjunction of potential aliases, potential non-aliases and type constraints and \(\text{conc}(c)\) be the simplified conjunction of potential aliases, potential non-aliases and type constraints obtained from \(c\) by instantiating unknown initial values by their concrete values computed during Phase II and dropping conjuncts that are true after the instantiation. If any conjunct of \(c\) is false after the instantiation, \(\text{conc}(c)\) is false. \(\text{conc}(c)\) is called relevant if either it involves only interface initial value’s of a particular public interface method or it is empty. Let \(S\) be a set whose each element is a conjunction of potential aliases, potential non-aliases and type constraints. Let \(\text{rconc}(S)\) be the set whose each element is a relevant \(\text{conc}(c)\) of some element \(c\) of \(S\). Let \(R\) be the set of relevant contexts of dfelm’s computed by RCI. Then,

- if RCI is run on a library \(L\) with the bounds on the numbers of potential aliases, potential non-aliases and type constraints in any relevant context being at least \(k_1, k_2, k_3\) simultaneously, then \(\text{rconc}(R)\) is a \(k_1k_2k_3\)-RCC of \(L\).

G Mark-reachable

Figures 11, 12, 13, 14, 15, 16, 17, 18 and 19 define mark-reachable. The data-flow elements computed by mark-reachable have one of the following three forms:

1. \(\langle\text{empty,reachable}\rangle\)
2. \(\langle\text{excp-type,reachable}\rangle\)
3. \(\langle\text{label,reachable}\rangle\)

An exception block is the body of a try, catch, finally or method. Given a statement \(S\), innermost-enclosing-exception-block\((S)\) is the innermost exception block containing \(S\).
Figure 9: Actuals
Figure 10: $\text{approx-space}(\text{param_init})$

mark-reachable() {
    // initialize worklist
    worklist = empty
    for each method $m$ in the current SCC
      wl-node = new worklist node($m$.entry, (empty, reachable))
      add wl-node to worklist
    process-worklist-mark-reachable()
    for each return-site $n$
      $n$.reaching-reachability-dfes = $n$.reaching-reachability-dfes $\cup$ $n$.successor.reaching-reachability-dfes
}

Figure 11: mark-reachable
process-worklist-mark-reachable() {
    while worklist is not empty {
        wl-node = delete node from worklist
        node = wl-node.node
        dfe = wl-node.dfe

        // condition node represents the test expression of a
        // if or while and new node is an object creation site
        if (node is an assignment node or condition node or new node)
            for each successor succ of node
                add-to-soln-and-worklist-if-needed-mr(succ, {dfe})

        if (node is a throw node)
            let excp-type be the declared type of the exception thrown by node
            x = exit of innermost-enclosing-exception-block(node)
            add-to-soln-and-worklist-if-needed-mr(x, {⟨excp-type, reachable⟩})

        if (node is the entry node of a try)
            process-try-entry-mr( node, dfe )
        if (node is the exit node of a try block)
            process-try-exit-mr( node, dfe )
        if (node is the entry node of a catch)
            add-to-soln-and-worklist-if-needed-mr(node.successor, {⟨empty, reachable⟩})
        if (node is the exit node of a catch)
            process-catch-exit-mr( node, dfe )
        if (node is the exit node of a finally)
            process-finally-exit-mr( node, dfe )
        if (node is the exit node of a method)
            process-method-exit-mr( node, dfe )
        if (node represents a break statement)
            process-break-mr( node, dfe )
        if (node is a call node)
            process-call-mr( node, dfe )
        if (node represents a return statement)
            process-return-statement-mr( node, dfe )
    }
}

process-try-entry-mr( node, dfe ) {
    /***
     * node.try is the try statement whose entry node is node ***/
    if ( innermost-enclosing-exception-block(node.try) is a finally )
        // try nested inside a finally
        // treat like a call to an anonymous procedure
        process-call-mr( node.call-node, dfe )
    else
        add-to-soln-and-worklist-if-needed-mr(node.successor, {dfe})
}

Figure 12: mark-reachable 1

Figure 13: mark-reachable 2
process-try-exit-mr(node, dfe) {
/***
 dfe.ecfi is the first element of dfe. ***/
if (dfe.ecfi represents an exception) {
    for each catch ct associated with node
        // ct catches any exception whose type is ct.catch-type or a subtype of ct.catch-type
        if (ct.catch-type is compatible with dfe.ecfi)
            add-to-soln-and-worklist-if-needed-mr(ct.entry, {dfe})
    if (there does not exist a catch ct associated
        with node such that ct.catch-type is same as dfe.ecfi
        or ct.catch-type is a super-type of dfe.ecfi)
        // it is possible for the exception to escape
        // all catches associated with node
        propagate-to-finally-if-needed-mr(node, dfe)
        // else the exception is caught by at least one catch clause
    }
    else {
        propagate-to-finally-if-needed-mr(node, dfe)
    }
}
}

Figure 14: mark-reachable 3

propagate-to-finally-if-needed-mr(node, dfe) {
    if (dfe.ecfi is empty)
        succ = successor of the try-catch-wally construct associated with node
    dfe = ⟨succ, reachable⟩
    if (there is a finally associated with node)
        add-to-soln-and-worklist-if-needed-mr(node.finally.entry, {dfe})
    else
        if (innermost-enclosing-exception-block(node.try) is a finally)
            process-method-exit-mr(node.method-exit, dfe)
        else
            if (dfe.ecfi is a label contained in innermost-enclosing-exception-block(node.try))
                add-to-soln-and-worklist-if-needed-mr(dfe.ecfi, {⟨empty, reachable⟩})
            else
                x = exit of innermost-enclosing-exception-block(node.try)
                add-to-soln-and-worklist-if-needed-mr(x, {dfe})
}

Figure 15: mark-reachable 3a

process-catch-exit-mr(node, dfe) {
    propagate-to-finally-if-needed-mr(node, dfe)
}

Figure 16: mark-reachable 4

process-finally-exit-mr(node, dfe) {
    /***
     node.try is the try statement associated with node. ***/
    if (innermost-enclosing-exception-block(node.try) is a finally)
        process-method-exit-mr(node.method-exit, dfe)
    else
        if (dfe.ecfi is a label contained in innermost-enclosing-exception-block(node.try))
            add-to-soln-and-worklist-if-needed-mr(dfe.ecfi, {⟨empty, reachable⟩})
        else
            x = exit of innermost-enclosing-exception-block(node.try)
            add-to-soln-and-worklist-if-needed-mr(x, {dfe})
}

Figure 17: mark-reachable 5
process-call-mr( node, dfe ) {
  /***
   * node.ecfis contains the ECFI's of the reachability data-flow elements reaching node ***/
  if (dfe.ecfi \notin node.ecfis)
    node.ecfis = node.ecfis \cup \{dfe.ecfi\}

  for each method m invocable from node using hierarchy analysis
    //m.exit.reaching-reachability-dfes contains data-flow-elements that have reached the exit node of m
    if (⟨empty,reachable⟩ \in m.exit.reaching-reachability-dfes)
      add-to-soln-and-worklist-if-needed-mr(node.successor.successor, \{dfe\})
    for each ⟨label,reachable⟩ \in m.exit.reaching-reachability-dfes such that
      label is contained in innermost-enclosing-exception-block(node)
        add-to-soln-and-worklist-if-needed-mr(label, \{dfe\})
    if (dfe is the first data-flow-element reaching node)
      // i.e., node is found reachable for the first time
      for each dfe1 in m.exit.reaching-reachability-dfes
        if (dfe1.ecfi is not empty and dfe1.ecfi is not a label contained
            in innermost-enclosing-exception-block(node))
          add-to-soln-and-worklist-if-needed-mr(node.successor, \{dfe1\})
}

process-return-site-mr( node, dfe ) {
  let dfe be ⟨ecfi,reachable⟩
  if (ecfi is empty)
    successor = ordinary successor of node
  else
    successor = exit of innermost-enclosing-exception-block(node)

  add-to-soln-and-worklist-if-needed-mr(successor, \{dfe\})
}

process-method-exit-mr( node, dfe ){
  for each call site c in the current SCC that can invoke node.method {
    if (c has been found to be reachable)
      if (dfe.ecfi is empty)
        for each ecfi1 \in c.ecfis
          add-to-soln-and-worklist-if-needed-mr(c.successor.successor, \{ecfi1,reachable\})
      if (dfe.ecfi is a label contained in innermost-enclosing-exception-block(c))
        for each ecfi1 \in c.ecfis
          add-to-soln-and-worklist-if-needed-mr(dfe.ecfi, \{ecfi1,reachable\})
      if (dfe.ecfi is an excep-type or dfe.ecfi is a label not contained in innermost-enclosing-exception-block(c))
        add-to-soln-and-worklist-if-needed-mr(c.successor, \{dfe\})
  }
}

Figure 18: mark-reachable 6
process-break-mr( node, dfe ) {
  let y = target of break
  if (y is contained in innermost-enclosing-exception-block(node))
    add-to-soln-and-worklist-if-needed-mr(y, {dfe})
  else
    x = exit of innermost-enclosing-exception-block(node)
    add-to-soln-and-worklist-if-needed-mr(x, {y, reachable})
}

process-continue-mr( node, dfe ) {
  let y = target of continue
  if (y is contained in innermost-enclosing-exception-block(node))
    add-to-soln-and-worklist-if-needed-mr(y, {dfe})
  else
    x = exit of innermost-enclosing-exception-block(node)
    add-to-soln-and-worklist-if-needed-mr(x, {y, reachable})
}

process-return-statement-mr( node, dfe ) {
  let method-exit be the exit node of the method containing node
  if (innermost-enclosing-exception-block(node) is a method body)
    add-to-soln-and-worklist-if-needed-mr(method-exit, {dfe})
  else
    x = exit of innermost-enclosing-exception-block(node)
    add-to-soln-and-worklist-if-needed-mr(x, {method-exit, reachable})
}

add-to-soln-and-worklist-if-needed-mr( node, dfes ) {
  for each dfe ∈ dfes
    if dfe ∉ node.reaching-reachability-dfes
      node.reaching-reachability-dfes = node.reaching-reachability-dfes ∪ {dfe}
      wl-node = new worklist node (node, dfe)
      add wl-node to worklist
}

Figure 19: mark-reachable