MODULAR DATA-FLOW ANALYSIS OF STATICALLY TYPED OBJECT-ORIENTED PROGRAMMING LANGUAGES

BY RAMKRISHNA CHATTERJEE

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Barbara Gershon Ryder

and approved by

New Brunswick, New Jersey

January, 2000

ABSTRACT OF THE DISSERTATION

Modular Data-flow Analysis of Statically Typed Object-oriented Programming Languages

by Ramkrishna Chatterjee

Dissertation Director: Barbara Gershon Ryder

Abstract

The solution of data-flow analysis of object-oriented programming languages such as C++/Java is needed for many important applications: aggressive code optimization, side-effect analysis, program specialization, program slicing and data-flow-based testing. However, data-flow analysis of object-oriented programming languages is difficult due to a large number of heap-allocated objects whose fields point to other heap-allocated objects (recursive structures), dynamic dispatch, frequent method invocations, a large number of methods, many invocation contexts per method and exceptions.

In this thesis we present a new data-flow analysis technique called **Relevant Context Inference (RCI)** for modular, flow- and context-sensitive data-flow analysis of statically typed object-oriented programming languages such as C++ and Java. This technique has been designed to overcome the above difficulties. *RCI* has several long sought-after characteristics:

1. It can analyze programs by keeping only a part of the programs in memory at a time, with a constant bound on the number of times a procedure needs to be in

memory.

2. It can analyze incomplete programs such as libraries.

3. It can analyze programs that have exceptions.

We have built a prototype of *RCI* for points-to analysis of C++ programs. The empirical results obtained using this prototype and presented in this thesis show that *RCI* is effective in practice.

We present several new complexity characterizations of points-to analysis in the presence of object-oriented language constructs: exceptions and dynamic dispatch. Our results clearly identify the difficult features and indicate approximations that any efficient algorithm has to make.

We also present a new approach to data-flow-based testing of object-oriented libraries using *RCI* . We show how the information computed by *RCI* can be used for generating relevant test cases.

Acknowledgements

I would like to thank Prof. Barbara Ryder, my advisor, for her support and guidance over the years. I am grateful to her for giving me freedom to work independently and for being there whenever I needed help. I thank Dr. Bill Landi from whom I learned many things, knowingly and unknowingly. I thank Prof. Tom Marlowe for his insightful comments and suggestions, and for his constant encouragement. I also acknowledge the support and friendship of other members of PROLANGS.

I thank Prof. Eric Allender, Prof. Uli Kremer and Dr. Frank Tip for their valuable comments on this thesis.

I would like to use this opportunity to thank Prof. Priti Shankar, my advisor at IISc, Bangalore. I am fortunate to have her as my teacher.

I thank Prof. Biswajit Chakraborty and Dr. Bhavani Mukherjee for motivating me to pursue higher studies. Although they would have been happier to see my PhD in theoretical Physics.

Finally, I would like to thank Kevin Longfellow for letting me use his cryptocard, without it I could not have worked on my thesis.

Dedication

To my best friend,

Who has accompanied me

sometimes as mother, sometimes as father,

sometimes as sister, sometimes as friend

and sometimes as teacher.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Data-flow analysis

The goal of data-flow analysis [MR90a][1] of statically typed object-oriented programming languages (e.g., Java, C++) is to compute information about programs (such as finding possible values of pointer variables at a program point) by statically analyzing the programs at compile time. The information computed by data-flow analysis has many important applications such as static resolution of dynamically dispatched calls [PR96], side-effect analysis [LRZ93], testing [RW85], program slicing [Tip96, TCFR97], program specialization [TS97] and aggressive compiler optimization [ASU86].

Iterative data-flow analysis as defined in [MR90a] is a fixed point calculation for recursive equations defined on a graph representing a program, that safely approximates the *meet over all paths solution (MOP)* of a data-flow problem. The graph representing the program is usually[2] a statement-level interprocedural control flow graph or **ICFG** [LR92]. The ICFG is a union of the statement-level control flow graphs of all procedures in the input program. Every statement in a procedure is represented by a node in the control flow graph of the procedure, with (intraprocedural) edges to successors and predecessors. Every test expression (e.g., the test expression of an *if* or *while*) is represented by a *condition* node which has two edges going to its successors: one to the node that represents the first statement that is executed if the condition is true and the other to the node that represents the first statement that is executed if the condition

[1] [MR90a] is a survey paper on data-flow analysis and it refers to other standard references [Kil73, Hec77, ASU86] on data-flow analysis.

[2] Other more efficient program representations have been used by some researchers [CKT86, Cal88], which are suitable for specific data-flow analysis problems.

is false. There are (interprocedural) edges between nodes representing call sites and the (possible) targets of these call sites. The call graph of the program consists of the procedures of the program and the edges between the call sites and their targets.

As discussed in [MR90a], different solutions for a data-flow problem form a meet semi-lattice. For example, consider a data-flow analysis problem called points-to analysis. The goal of points-to analysis is to find possible values of pointer variables at a program point. If there is a path from the beginning of the program to the program point in question such that flow along this path causes a pointer variable to have a specific value, then this pointer-value pair is in the solution set. As a result, the meet operator for this problem is the *union (logical or)* operator and the *MOP* is the union of the solutions reaching a program point along different paths. Next, consider another data-flow analysis problem called constant propagation. The goal of constant propagation is to find whether a variable has a constant value on all paths reaching a program point. Thus, the meet operator for this problem is the *intersection (logical and)* operator and the *MOP* is the intersection of the solutions reaching a program point along different paths.

Any point in the solution lattice that is below ($\leq$) the lattice point that represents *MOP* is called a *safe* solution. For example, for points-to analysis the partial order for the solution lattice is the superset relation and the safe solutions are supersets of *MOP*. Similarly, for constant propagation the partial order for the solution lattice is the subset relation and the safe solutions are subsets of *MOP*. An iterative data-flow analysis algorithm as discussed in [MR90a] computes a *maximal fixed point (MFP)* of the data-flow equations that the algorithm uses to represent the data-flow problem. *MFP* is a safe approximation of *MOP* and thus *MFP* is located below *MOP* in the solution lattice. For certain data-flow analysis problems, *MFP* is same as *MOP* [KU76]; while for certain other data-flow analysis problems (e.g., points-to analysis), computing *MOP* is undecidable [Lan92b] and *MFP* is only a safe approximation of *MOP*.

1.2 Flow and context sensitivity

```
class B {
};
class C: public B {
};
class A {
  public:
    void method( ) {
      B *p;

      l:  p = new B();
      // Note:  objects created at statement l are represented by object_l
      // solution computed by a flow-insensitive
      // algorithm:  {⟨p,object_l ⟩, ⟨p,object_m ⟩}
      // solution computed by a flow-sensitive
      // algorithm:  {⟨p,object_l ⟩}

      ...

      m:  p = new C();
      // solution computed by a flow-insensitive
      // algorithm:  {⟨p,object_l ⟩, ⟨p,object_m ⟩}
      // solution computed by a flow-sensitive
      // algorithm:  {⟨p,object_m ⟩}
    }
};
```

Figure 1.1: flow-sensitivity

```
class B {
};
class A {
  public:
    static B* method1( B *param ) {
      return param;
    }

    static void method2( ) {
      B *p,*q;

      l1:  p = new B();
      q = method1( p );

      // solution computed by a context-sensitive algorithm:
      // $\{\langle$q, $object_{l1}$ $\rangle\}$.  $object_{l1}$ represents objects created at
      // program point l1.

      // solution computed by a context-insensitive algorithm:
      // $\{\langle$q, $object_{l1}$ $\rangle$, $\langle$q, $object_{l2}$ $\rangle\}$

      // Note:  $\langle$q, $object_{l2}$ $\rangle$results from the unrealizable path:
      // method3 $\xrightarrow{calls}$ method1 $\xrightarrow{returns-to}$ method2
    }

    static void method3( ) {
      B *p,*q;

      l2:  p = new B();
      q = method1( p );

      // solution computed by a context-sensitive algorithm:
      // $\{\langle$q, $object_{l2}$ $\rangle\}$

      // solution computed by a context-insensitive algorithm:
      // $\{\langle$q, $object_{l1}$ $\rangle$, $\langle$q, $object_{l2}$ $\rangle\}$
    }
}
```

Figure 1.2: Context-sensitivity

Flow- and context-sensitivity affect both the precision and cost of data-flow analysis. A *flow-insensitive* algorithm ignores the ordering of statements within a method[3]; 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 *flow-sensitive* algorithm performs strong updates or kills, i.e., if a statement updates a variable (e.g., $p = q;$), then a *flow-sensitive* algorithm removes from the solution set the previous values of the variable reaching the statement and the solution set for the variable (e.g., p) after the statement contains only the values assigned by the statement. The example in Figure 1.1 illustrates this difference. 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. The example in Figure 1.2 illustrates this difference.

Existing algorithms for data-flow analysis vary in their flow and context sensitivity; they compute approximate solutions which are safe approximations of the precise 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, RHS95, Ruf95, PR96] are the most precise but also the most expensive (in time and memory). Approaches like [PS91, PC94, GDDC97] are in between the above two extremes.

1.3 Motivation for modular data-flow analysis

The precision of the solution computed by data-flow analysis directly affects its utility in applications. However, flow- and context-sensitive data-flow analysis of C++/Java is difficult, due to dynamic dispatch, exceptions, many heap-allocated objects that point to each other (recursive structures), frequent method invocations and 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

[3] We will use the terms method and procedure interchangeably.

programs. In this thesis, we present a new data-flow analysis technique called *Relevant Context Inference* or **RCI** for modular, flow- and context-sensitive data-flow analysis of statically typed object-oriented languages such as C++ and Java. The above mentioned difficulties were addressed in the design of this new technique. By the term *modular*, we mean a technique for data-flow analysis which never requires the entire program source to be in memory at any one time and there is a constant bound on the number of times a procedure needs to be moved into or out of memory[4]. In traditional, non-modular data-flow analysis techniques, a method cannot be moved out of memory without the possibility of it being needed again, until the final, fixed point 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. This is shown in Figure 1.3. Figure 1.3 shows two procedures: a caller and a callee. Each new piece of contextual information propagated from the caller to entry node of the callee is pushed through the callee and the result is returned to the caller. This means that if the callee is not kept in memory throughout the analysis, for each new piece of contextual information propagated to the entry node of the callee, the callee needs to be brought into memory. Moreover, such non-modular techniques need to analyze the whole program simultaneously: the caller cannot be analyzed separately from the callee because the caller needs the *result* back from the callee and the callee cannot be analyzed separately from the caller because we need the data-flow information at the entry node of the callee before it can be pushed through the callee. Hence, such non-modular techniques cannot analyze incomplete programs such as libraries. However, much object-oriented code is written as libraries; thus another goal of our new technique is to analyze incomplete programs such as libraries.

[4]Ideally we would like to analyze one procedure at a time with a constant bound on the number of times a procedure needs to be in memory. In this thesis, we will come close to this goal by analyzing only a strongly connected component of the call graph at a time. This is close to the ideal situation because the strongly connected components of the call graph are small in practice (see Chapter 5) and contain only a few procedures.

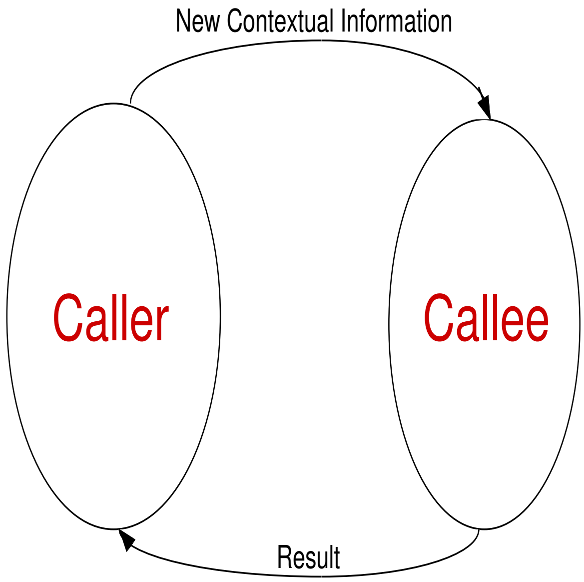

Figure 1.3: Traditional non-modular analysis

1.4 Main contributions of this thesis

The main contributions of this thesis are as follows:

1. Chapters 1, 2 and 4 present a new data-flow analysis technique called *Relevant Context Inference* for modular, flow- and context-sensitive data-flow analysis of statically typed object-oriented programming languages such as C++ and Java. *RCI* has has several long sought-after characteristics [CRL99]:

   (a) *RCI* can analyze programs by keeping only a part of the program in memory at a time (with a constant bound of 4 on the number of times a procedure needs to be moved into and out of memory).

   (b) *RCI* can analyze incomplete programs such as libraries.

   (c) *RCI* can analyze programs that have exceptions.

2. Chapter 3 presents several new complexity characterizations of points-to analysis in the presence of object-oriented language constructs such as dynamic dispatch and exceptions [CRL98a].

3. Chapter 4 presents a new approach to data-flow-based testing of object-oriented libraries [CR99].

4. Chapter 5 discusses a proof-of-concept prototype of *RCI* for points-to analysis of C++ programs [CRL99]. The empirical results obtained using this prototype show that *RCI* is effective in practice.

1.5 Thesis Organization

The rest of this thesis is organized as follows. In Chapters 2 and 4 respectively, we present two instantiations of *RCI* for two data-flow analysis problems: **points-to analysis** and **def-use analysis**. We have chosen these two data-flow analysis problems because these are important data-flow analysis problems which have been widely studied and which have many practical applications. Chapter 3 presents our new complexity characterizations. Chapter 5 presents empirical results obtained using our prototype. The rest of this chapter is organized as follows. First, we will give a brief overview of *RCI* . Next, we will introduce a simple object-oriented language used to present our algorithms. Finally, we will present an example that will be used to illustrate our algorithms.

1.6 Overview of Relevant Context Inference

Intuitively, *RCI* is a data-flow analysis *schema* that analyzes 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.

RCI is an iterative worklist algorithm [KU76] that is flow- and context-sensitive. An iterative worklist algorithm maintains a worklist containing (1) program points where new data-flow elements have reached and (2) these new data-flow elements. Whenever a new data-flow element reaches a program point, the program point and the data-flow element are put on the worklist for future processing. The algorithm iterates until the worklist becomes empty, indicating that a fixed point has been reached.

RCI takes as input an ICFG. From the ICFG, 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].[5] 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. The choice of unknown initial values depends upon the specific data-flow analysis problem being solved. For each method *RCI* computes, in terms of unknown initial values, a safe approximation to the method's complete transfer function for the data-flow analysis problem being solved. We will call this approximation, the **summary transfer function**. The summary transfer function of a method M is the set of data-flow elements 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 data-flow elements. 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

[5]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.

(or no dependence at all), and hence are computed by bottom-up traversal on SCC-DAG, without iteration.

The results of this phase on a SCC are two-fold: (i) a data-flow solution (for the data-flow problem being solved) at each node of every method in the SCC and (ii) a summary transfer function for every method in the SCC, both of which are parametrized by unknown initial values and conditions on 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. The propagation within an SCC is done iteratively until a fixed point is reached, while propagation across SCC's is done in a top-down manner without iteration. This phase involves only the entry nodes and call nodes, and as our empirical results show, it is extremely fast.

- **Phase III:** This phase involves only nodes that are not entry nodes. In this phase, the unknown initial values in the data-flow elements 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 data-flow solution at each node is expressed entirely in terms of concrete values.

Modularity 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.

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.

1.7 RL

For the ease of presentation, in this thesis we describe our algorithms for a simple object-oriented language **RL** (Figure 1.4) 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 algorithms. *RL* is defined in Figure 1.4. *Expr* is any side-effect-free expression that does not have any function call. We also assume that no pointer field is accessed in *Expr*. Note that there is no loss of generality because any expression can be put in this form using temporaries. *[pattern]* means at most one occurrence of the *pattern*. $\{pattern\}^+$ means one or more occurrences of the *pattern*. $\{pattern\}^*$ means zero or more occurrences of the *pattern*. $\{a \mid b\}$ means a or b. The terminal symbols are underlined. *RL* includes single inheritance, dynamic dispatch, recursive types, exceptions and pointer assignment statements with a single level of dereferencing[6]. Considering only pointer accesses with a single level of dereferencing does not result in loss of generality because using temporaries any pointer access with any level of dereferencing can be translated into an equivalent sequence of statements each having single level of dereferencing. The semantics of the constructs is same as in C++. All exception types must be derived from the class Exception, as in Java. The semantics of *finally* statements and exceptions is same as in Java [GJS96]. *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).

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

If the algorithms are 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 algorithms. The subset of C++ that can easily be handled by the algorithms excludes arbitrary casting, unions, uninstantiated templates, pointers to data members and pointers to member methods (which are different from ordinary function pointers). Up-cast of a derived class to a base class and down-cast of a base class to a derived class can be handled.

In addition, the algorithms can essentially handle Java without threads; however, 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 locations accessible outside the *finalizers*. We exclude static initializations that depend upon the order in which files are loaded. The algorithms can handle only those static initializations for which it is safe (with respect to def-use 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.

1.8 Example

The example in Figure 1.5 will serve as the running example throughout this thesis. We will describe our results for libraries with multiple **public interface methods**, library methods that can be directly invoked from outside the library. Since a complete program can be considered to be a library with a single *public interface method*, i.e., *main*, our results also trivially apply to complete programs. We expect the reader to skip the details of this example during the first reading of this chapter and come back to it when we refer to this example from other places in the thesis. The class *Lib* has been declared as *friend* of classes *A* and *C* because we want the to access *private* methods of classes *A* and *C* from methods of class *Lib*. This ensures that the only entry points of the code in Figure 1.5 are the *public interface methods* of class *Lib*.

```
Program := {Class | Proc}+
Class := class ClassName [: public (ClassName | Exception)]
  { {DataMember| Method}+ }
DataMember := Protection [static] Type FieldName ;
Type := ClassName * | PremitiveType
PremitiveType := int | char | float | bool
Method := Protection [static | virtual] [void | Type]
  MethodName ( [Param RestParam*] ) { Body }
Param := Type VarName
RestParam := , Param
Body := Decls Stmnt+
Decls := Decl*
Decl := Type VarName ;
Proc := {void | Type} ProcName ( [Param RestParam*] )
  { Body }

Stmnt := AssignmentStmnt | NewStmnt | Call | If |
  While| ReturnStmnt | Break | Continue | Try | Throw | ;
ReturnStmnt := return VarName ;
Break := break [Label] ;
Continue := continue [Label] ;

AssignmentStmnt := Lhs = Rhs ;
Lhs := VarName | VarName→FieldName
Rhs := VarName | VarName→FieldName | 0 | Expr

Call := [VarName = ]
  {VarName→MethodName|MethodName
  |ClassName::MethodName|ProcName}
  ( [VarName RestVar*] ) ;

NewStmnt := VarName = new ClassName ( [VarName RestVar*] ) ;
RestVar := , VarName

If := if ( Expr ) {Stmnt+ } [else { Stmnt+ } ]

While := [Label :] while ( Expr ) { Stmnt+ }

Throw := throw VarName ;
Try := try { Stmnt+ } Catch* [Finally]
Catch := catch ( ClassName * VarName ) { Stmnt+ }
Finally := finally { Stmnt+ }

VarName := Name ; FieldName := Name ; ProcName := Name ; MethodName := Name;
ClassName := Name Label := Name; Protection := public : | protected : | private :
```

Figure 1.4: Grammar for *RL*

```
class A {
  private:  A *next;
  private:  void virtual update(){
    1:  Lib::global1 = new A();
    1a:  Lib::global3 = Lib::global1;}
  friend Lib; }
class B : public A {}
```

```
class C: public A {
  private:  void update() {
    2:  Lib::global1 = new C();
    2a:  Lib::global3 = Lib::global1; }
    friend Lib; }
```

```
class ET1 :  public Exception { };
class ET2 :  public ET1 { };
class ET3 :  public ET2 { };
class ET4 :  public ET1 { };
class ET5:  public Exception { };
```

```
class Lib {
public:  static A *global1, *global2, *global3;
private: static void method1(A* a1, A* a2, A* a3) {
  A *r, *u;

  3:  r = a1;
  4:  a1→next = a3;
  5:  r→update();
  6:  global2 = a2→next;
  7:  u = global1;
  8:  global1 = new A(); 8a:  }

public: static void method2( A* a4, A* a5, A* a6 ) {
  /*** public interface method ***/
  A *p, *q;
  9:  method1(a4,a5,a6);
  10:  p = new B(); 11:  q = new C();
  12:  method1(p,q,q);
  13:  }
```

```
private: static void method3 ( ET1 *e1 ) {
  A *u1;
  try {
      14:  global3 = new A();
      15:  throw e1;
    16:
  }
  17:catch ( ET2 *excp ) {
    18:  u1 = global3;
  18a:  }
  19:  }

public: static void method4( ET1 *e2 ) {
  /*** public interface method ***/
  A *u2,*u3; ET1 *e3, *e4;
  if ( _ ) {
    20:  e3 = new ET4();
    try {
      21:  method3( e3 );
    21a:  }
    22:catch( ET4 *excp1 ) {
      23:  u2 = global3;
    23a:  }
  }
  else {
    23b:  e4 = e2;
    try {
      24:  method3( e4 );
    24a:  }
    25:  catch( ET4 *excp2 ) {
      26:  u2 = global3;
    }
    27:  catch( ET1 *excp3 ) {
      28:  u3 = global3;
```

```
    }
  }
  29: }
private: static void method5( A *param ) {
  ET1 *unexp;
  30: global1 = param;
  31: unexp = new ET1();
  32: throw unexp;
  33: }

public: static void method6( ) {
/*** public interface method ***/
  A *local;
  ET1 *unexp1;
  34a: unexp1 = 0;
  34: local = new A();
  try {
    35: method5( local );
  35a: }
  36: catch( ET5 *param1 ) {
  }
  37: finally {
    38: try {
      39: unexp1 = new ET1();
      40: throw unexp1;
    40a: }
    41: catch( ET1 *param2 ) {
    41a: }
    42: finally {
      43:
    }
  44: } // End of outer finally

  45:
```

```
}
} /*** end of class Lib ***/
```

Figure 1.5: Library $L_{example}$

The example needs to be here.

Figure 1.5: Library $L_{example}$

Chapter 2

Modular Points-to Analysis

This chapter explains the instantiation of *RCI* schema for the data-flow analysis problem called points-to analysis [EGH94]. The goal of points-to analysis is to compute at each program point, the set of objects to which a pointer may point during some execution. We will call the instantiation of *RCI* for points-to analysis **Points-to-Algo** [CRL99, CRL98b]. The rest of this chapter is organized as follows. Section 2.1 presents some definitions needed to explain Points-to-Algo. Section 2.2 briefly describes the semantics of exception-handling constructs in Java. Recall that the semantics of exception-handling constructs in *RL* is same as in Java. Section 2.3 describes the data-flow elements computed by Phase I of Points-to-Algo. Section 2.4 gives an overview of the various steps of Points-to-Algo. Section 2.5 explains some advantages of bottom-up analysis using unknown initial values as compared to top-down analysis using concrete values. Section 2.6 explains how Points-to-Algo (and *RCI* in general) handles recursive types. Section 2.7 explains how Points-to-Algo finds reachable nodes in the ICFG. Section 2.8 presents some details of Phase I of Points-to-Algo. Section 2.9 describes Phase II of Points-to-Algo. Section 2.10 describes Phase III of Points-to-Algo.

For simplicity, in this thesis we will assume that a library does not invoke a method external to the library. How *RCI* can be extended to handle those libraries that can invoke methods external to them is discussed in Section 7.1.2.

2.1 Definitions

This section presents some definitions needed to explain Points-to-Algo.

Public interface method: A library method that can be directly invoked from outside the library is called a *public interface method* of the library.

Heap-name: Any static analysis technique needs to represent a potentially infinite number of heap-allocated, run-time objects with a finite number of names. Like many previous researchers [RM88, LR92, CBC93, MLR$^+$93, EGH94, WL95, Ste96], we represent all the run-time objects created at a program point n with the single name $object_n$, a **heap-name**.

Unknown initial value (UIV): We use var_{init} to represent the unknown initial object to which the global or parameter *var* of pointer type points at the entry node of a method. Similarly, $var_{init}.next_{init}$ represents the unknown initial value to which *var*→*next* points and $var_{init}.next_{init}.next_{init}$ represents the unknown initial value to which *var*→*next*→*next* points. For example, consider the unknown initial value of the parameter *a1* of *method1* defined in Figure 1.5. Figure 2.1 shows the relationships between the different unknown initial values. Obviously, in the presence of recursive types, the number of unknown initial values accessed in a method can 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. Patterns in the *access paths* [Deu94] of unknown initial values are used to form these sets (see Section 2.6).

Interface initial value: An unknown initial value at the entry node of a *public interface method* is called an interface initial value.

Compatible classes: Two classes are *compatible* if they are the same class or they are related by a subtype-supertype relationship.

Compatible unknown initial values: Two unknown initial values are *compatible* if their declared classes are *compatible*.

Exception Block: An *exception block* is the body of a try statement, a catch statement, a finally statement or a method.

Innermost Enclosing Exception Block: Given a statement *S*, *innermost-enclosing-exception-block(S)* is the innermost *exception block* containing *S*.

Precision and Safety of points-to analysis: Let *ptr* be a pointer and m be a program point. A pair $\langle$*ptr,object*$\rangle$belongs to the *precise* solution of points-to analysis at program point m if and only if there exists an execution path p from the entry node of a *public interface method* to m and an environment[1] e at the entry node of the *public interface method*, such that when p is executed starting with environment e at the entry node of the *public interface method*, at m the pointer *ptr* is read and *ptr* points to an object represented by *object*. Here *object* is an object allocated during the execution of p or *object* is an initial value in e. A **safe** solution of points-to analysis is one that is a superset of the *precise* solution. Points-to-Algo calculates a safe solution.

Note that in this thesis we are concerned only with the values of pointer variables or fields that are read at a program point. Most important applications (e.g., static resolution of dynamic dispatch [PR96] and side-effect analysis [LRZ93]) that use the solution of points-to analysis need only this information. This is why we have defined the precise solution of points-to analysis at a program point to include only the solution for the pointer locations that are read at the program point. In this thesis we will show that the solution computed at a program point by Points-to-Algo is sufficient to find the values of pointer variables or fields that are read at the program point.

There are other approaches [EGH94] that define the precise solution of points-to analysis at a program point to include the values of all pointer variables visible at the program point. This alternative definition of the precise solution of points-to analysis can be stated as follows: a pair $\langle$*ptr,object*$\rangle$ belongs to the *precise* solution of points-to analysis at program point m if and only if *ptr* is visible at m and there exists an execution path p from the entry node of a *public interface method* to m and an environment e at the entry node of the *public interface method*, such that when p is executed starting with environment e at the entry node of the *public interface method*, at m the pointer *ptr* points to an object represented by *object*.

As shown in Appendix D, it is easy to extend Points-to-Algo to compute the value of any pointer visible at a program point. However, for reasons of simplicity, we will not

[1] An environment is a function that maps locations to their values.

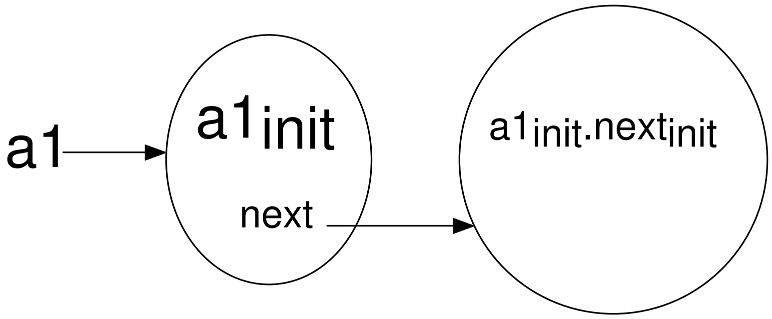

Figure 2.1: Unknown initial values

discuss these extensions any more in this thesis.

2.2 Semantics of Exceptions

Before describing Points-to-Algo in detail, we will briefly describe the semantics of exception-handling constructs in Java [GJS96][2] as it is important for understanding the rest of this chapter. The hierarchy of exception classes in Java is shown in Figure 2.2. User-defined exception types are usually subtypes of the predefined class *Exception*[3]. The language constructs used for dealing with exceptions are: *throw*, *try*, *catch* and *finally* statements. A user-generated exception is thrown using a throw statement. For example, *throw p;* throws the exception object to which the variable p points. If a throw statement appears inside a try statement (which is the usual situation), then the exception object thrown by the throw statement is propagated to the exit of the try statement. In general, the exception thrown by a throw statement s is propagated to the exit of *innermost-enclosing-exception-block(s)*. A try statement can have a sequence of catch statements associated with it. A catch statement catches all exception objects whose run-time types are subtypes of the declared type of the parameter of the catch statement. For example, a catch statement with prototype *catch(ET* param)* catches any exception object whose run-time type is ET or a

[2]Recall that the semantics of exception-handling constructs in *RL* is same as in Java.

[3]In *RL* we require that all user-defined exception types are subtypes of *Exception*.

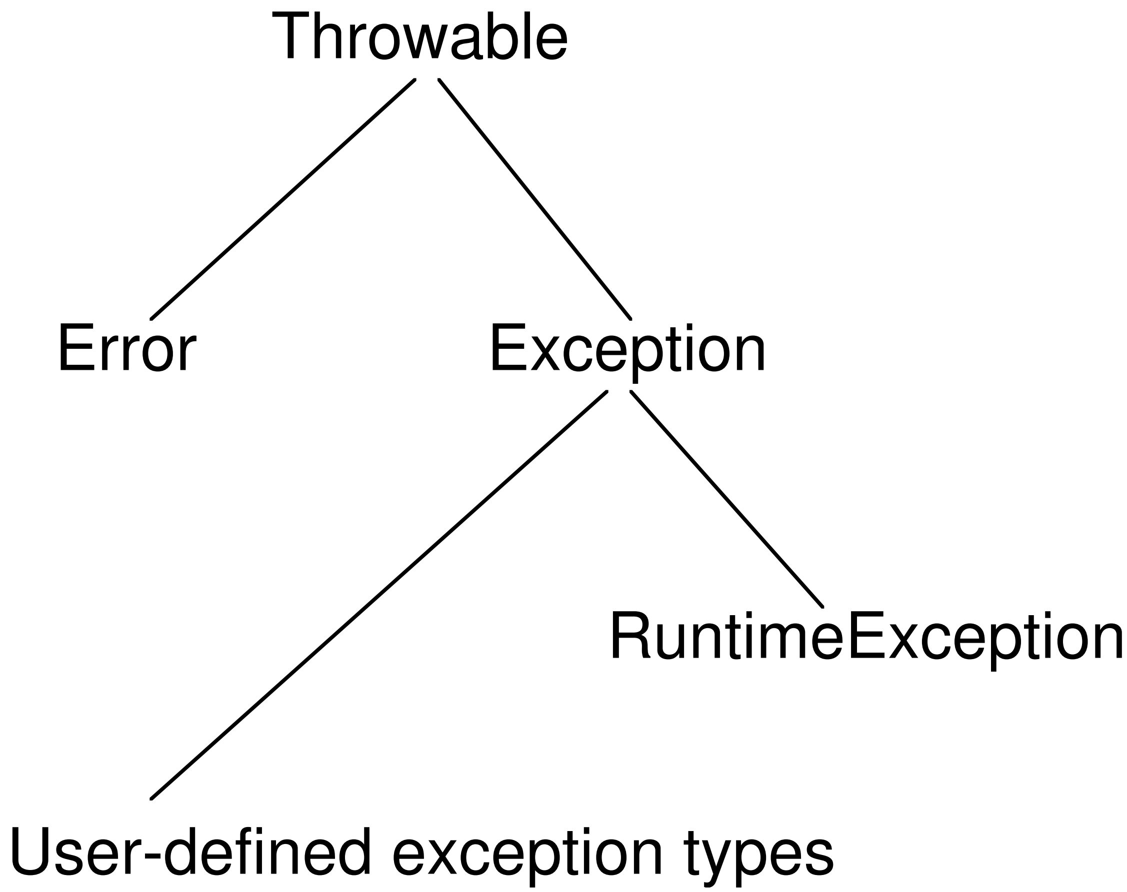

Figure 2.2: Class hierarchy for exception types in Java

subtype of *ET*. When an exception object reaches the exit of a try statement, the sequence of catch statements (if any) associated with the try statement is searched sequentially to find the first catch statement that can catch the exception object. If an appropriate catch statement is found, then the exception object is propagated to the entry of the catch statement and the exception object is assigned to the parameter of the catch statement. If none of the catch statements associated with the try statement can catch the exception object and there is no finally statement associated with the try statement, then the exception object is propagated to the exit of the innermost exception block containing the try statement and the control flow continues from this exit point in a similar manner. If an exception thrown in a method is not caught inside the method then the search for an appropriate catch statement (i.e., exception handler) continues up the dynamic call chain, in the currently active callers.

A try statement can optionally have a finally statement associated with it. Intuitively, the code in a finally statement is used for clean-up operations. However, a finally statement in Java has involved semantics. It is executed no matter how the try statement terminates: normally or due to an exception. A finally statement is always entered with a reason, which could be (1) an exception thrown in the corresponding try statement or one of the corresponding catch statements, or (2) leaving the try statement or one of its catch statements due to a *return*, (labelled) *break* or (labelled) *continue*, or by falling through. This reason is remembered on entering a finally statement, and unless the finally statement creates its own reason to exit itself, at the exit of the finally statement this reason is used to determine subsequent control flow. If the finally statement creates its own reason to exit itself (e.g., due to an exception), then this new reason **overrides** any previous reason for entering the finally statement. Moreover, a call or a try statement nested inside a finally statement can cause exceptions (and other reasons for entering finally statements) to stack up. The following examples show some interesting situations that can arise in the presence of finally statements. A formal description of all the possible situations is given in [GJS96], and we strongly recommend that the reader goes through the relevant sections in [GJS96] before reading the rest of this thesis.

```
proc() {
int i,j;
  i = 0;
  loop: while ( _ ) {
    1: j = i;
    while ( _ ) {
      try {
        2: continue loop;
      }
      finally {
        3: i++;
      }
    }
  }
}
```

Figure 2.3: Reason for entering a finally statement

```
proc() {
ET1 *u;
  try {
    u = new ET1();
    1: throw u;
  }
  finally {
    2: return;
  }
}
```

Figure 2.4: Overriding of the reason for entering a finally statement

First, consider the example in Figure 2.3. It shows one of the possible reasons for entering a finally statement. The reason for entering the finally statement is the labelled continue statement at program point **2**. After the continue statement, the finally statement is executed before the control goes back to program point *loop*. As a result, at program point **1**, the value of i reflects the increment at program point **3**.

Next, consider the example in Figure 2.4. It shows how the reason for entering a finally statement can be overriden inside the finally statement. The reason for entering the finally statement is the exception thrown at program point **1**. However, the return

```
proc() {
ET1 *u;
  try {
    u = new ET1();
    1: throw u;
  }
  finally {
    try {
      u = new ET2();
      2: throw u;
    }
    3: catch( ET2* param ) {
    }
    4:
  }
}
```

Figure 2.5: Stacking of exceptions inside a finally

at program point **2** overrides the exception and causes return from *proc* without any exception.

Finally, consider the example in Figure 2.5. It shows the stacking of exceptions in the presence of a try statement nested inside a finally statement. At program point **3** there are two active exceptions: the one thrown at program point **1** is at the bottom of the stack and the one thrown at program point **2** is at the top of the stack. The exception thrown at program point **2** is caught by the catch at program point **3** and as a result, at program point **4**, only the exception thrown at program point **1** is active.

The types of all run-time exceptions are subtypes of the class *RuntimeException*. We have considered only exceptions generated by *throw* statements in this thesis. Since run-time exceptions can be generated by almost any statement, we have ignored them. Our algorithms 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.

There are two kinds of **PTA-dfelms**:
PTA-dfelm-a. [*ECFInfo*, *relevant context*, *points-to*]
PTA-dfelm-b. [*relevant context*, *exception object*]

Figure 2.6: Phase I-pta data-flow elements

2.3 Phase I-pta data-flow elements

The data-flow elements computed by Phase I-pta (**PTA-dfelms**) are shown in Figure 2.6. *PTA-dfelms* represent (a) values of pointer variables and (b) exception objects.

Intuitively, a *PTA-dfelm*-**a** represents a value of a pointer variable at a specific program point, which incorporates relevant context information and exception control flow information. The relevant context information summarizes the contexts in which the *PTA-dfelm*-**a** holds at the program point and the exception control flow information summarizes the information about uncaught exceptions along paths through which the *PTA-dfelm*-**a** reaches the program point. The exception control flow information is used for deciding control flow at the exit node of a try statement, a catch statement, a finally statement and a method.

Intuitively, a *PTA-dfelm*-**b** represents an uncaught exception object reaching a program point. When such an exception object is caught by a catch statement, the *PTA-dfelm*-**b** representing the exception object is used for determining the value of the parameter of the catch statement, as the exception object is assigned to the parameter.

Now in the following paragraphs, we will describe the *PTA-dfelms* formally and present examples to illustrate them.

2.3.1 PTA-dfelm-a

As stated above, a **PTA-dfelm-a** represents a value of a pointer. An **ECFInfo** or *exception control-flow information* is one of the following:

- *excp-type*, meaning the associated *PTA-dfelm*-**a** reaches a program point along paths from the entry node of the method containing the program point such that when these paths are followed, at the program point there is an uncaught exception whose run-time type is *excp-type*,

- *uiv*, meaning the associated *PTA-dfelm-***a** reaches a program point along paths from the entry node of the method containing the program point such that when these paths are followed, at the program point there is an uncaught exception object which is the unknown initial value *uiv*,

- *label*, meaning the associated *PTA-dfelm-***a** reaches a program point along paths from the entry node of the method containing the program point such that when these paths are followed, at the program point there is a pending transition (due to a break statement, a continue statement, a return statement or falling through a try statement or catch statement associated with a finally statement) to statement number *label*, or

- *empty*, meaning the associated *PTA-dfelm-***a** reaches a program point along paths from the entry node of the method containing the program point such that when these paths are followed, at the program point there is no uncaught exception or pending transition.

Intuitively, a *non-empty ECFInfo* stores the signature of an uncaught exception (or label of the statement to which there is a pending transition) along a path through which a *PTA-dfelm-***a** reaches a program point, and determines the future propagation of the *PTA-dfelm-***a** from that program point. For example, statement **15** in Figure 1.5 throws $e1_{init}$ and hence when the *PTA-dfelm-***a** representing the value of *global3* (i.e., [*empty*,*rc1*,⟨*global3*,$object_{14}$⟩], where *rc1* is a relevant context defined later) is propagated across statement **15**, $e1_{init}$ becomes the *ECFInfo* of the new *PTA-dfelm-***a** (i.e., [$e1_{init}$,*rc1*,⟨*global3*,$object_{14}$⟩]) propagated to statement **16**. The reason for storing an unknown initial value *uiv* as the signature of exception when *uiv* is thrown as the exception object is that the concrete type of the exception can be *typeof(uiv)* or any of its subtypes. The concrete type of the exception will be determined using concrete values of *uiv*. For example, in Figure 1.5 when *method3* is invoked from statement

21, the concrete type of the exception thrown at statement **15** is *ET4*, and thus on returning from the call, the *ECFInfo* $e1_{init}$ is replaced by *ET4* and the new *PTA-dfelm*-**a** [*ET4*,*rc2*,⟨*global3*,$object_{14}$⟩] (again *rc2* is a relevant context defined later) is propagated to statement **22**. The reason why a singleton *ECFInfo* is sufficient is that in the absence of finally statements, exceptions (and other reasons for entering a finally statement) do not stack up. A call or a try statement nested inside a finally statement can cause exceptions to stack up; however, a singleton *ECFInfo* is still enough, because the stack is implicitly maintained at call sites by storing *ECFInfo* in each actual-to-uiv binding, and a try statement nested inside a finally statement is treated like a call to an anonymous procedure. An anonymous procedure representing a try statement directly contained in a finally statement is considered to be part of the SCC containing the method that contains the try statement.

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

⟨*var*, *val*⟩.

Here *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 *val* is an unknown initial value or heap-name. *val* can also be *null*, which is treated as a special heap-name.

A *bf relevant context* has the form:

⟨*alias context*, *type context*, *exception 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_1 **eq** uiv_2)

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

(uiv_1 **neq** uiv_2).

Here uiv_1 and uiv_2 are unknown initial values. For example, at program point **5** in Figure 1.5, Points-to-Algo computes [*empty*,⟨($a1_{init}$ **eq** $a2_{init}$),*empty*,*empty*⟩,⟨$a2_{init}.next$,$a3_{init}$⟩]

to record the potential modification to $a2_{init}.next$ by statement **4** in those contexts in which *a1* and *a2* are aliased at the entry node *method1*. Similarly, at program point **5** in Figure 1.5, Points-to-Algo also computes [*empty*,⟨(*a*1$_{init}$ **neq** *a*2$_{init}$),*empty*,*empty*⟩,⟨$a2_{init}.next$,$a2_{init}.next_{init}$⟩], which shows that $a2_{init}.next$ retains its value across statement **4** in those contexts in which *a1* and *a2* are not aliased at the entry node *method1*.

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

(type(uiv) $\in$ *T::x)*,

where *uiv* is an unknown initial value, T is a class and x is a dynamically dispatched method defined in class T (a virtual method in C^{++}). *T::x* represents the set of classes containing T and all the subtypes of T such that at a dynamically dispatched call site that invokes x, if the type of the receiver is any of these subtypes, then the invocation of method x will be resolved to the definition of method x in class T (i.e., *T::x*). The constraint means that the associated *PTA-dfelm* is valid only in those contexts in which the concrete, run-time type of *uiv* (not the declared type) belongs to *T::x*. For example, in Figure 1.5 *A::update* represents **{***A*,*B***}**. A type constraint in a *type context* is inferred when an unknown initial value is the receiver of a dynamic dispatch. For example, the *PTA-dfelm*-**a** [*empty*,⟨*empty*,(*type*($a1_{init}$)∈*A::update*),*empty*⟩,⟨*global1*,$object_1$⟩] is computed at program point **6** in Figure 1.5 because the unknown initial value $a1_{init}$ is the receiver at program point **5**.

An **exception type context** is *empty* or it is a conjunction of type constraints of one of the following two forms:

1. $(type(uiv) \leq T)$ or
2. $(type(uiv) \not\leq T)$.

The first type constraint says that the associated *PTA-dfelm* holds only in those contexts where the concrete type of the unknown initial value *uiv* is class T or a subtype of T.

While the second type constraint says that the the associated *PTA-dfelm* holds only in those contexts where the concrete type of the unknown initial value *uiv* is neither T nor a subtype of T. The type constraints in an *exception type context* are inferred when an unknown initial value is propagated as an exception object. For example, the *PTA-dfelm*-**a** [*empty*,⟨*empty,empty,(type*($e1_{init}$)$\leq$ *ET2)* ⟩,⟨*global3,object*$_{14}$⟩] is computed at program point **18** in Figure 1.5 because the unknown initial value $e1_{init}$ is thrown as an exception object at program point **15** and this exception is caught by the catch statement at program point **17** only in those contexts in which the concrete type of $e1_{init}$ is *ET2* or a subtype of *ET2*.

These relevant contexts are inferred by the algorithm during analysis and they summarize the contexts under which the corresponding *PTA-dfelms* hold. When a *PTA-dfelm* *dfe* is propagated from the exit node of a method to a call site that invokes the method, one of the following three things happens for a conjunct of the relevant context of *dfe*: the conjunct evaluates to *true*, it evaluates to *false* (*dfe* is not propagated to the call site in this case), or it is translated into a similar conjunct involving the unknown initial values of the caller.

We will use **Empty-context** to represent the relevant context ⟨*empty, empty, empty*⟩.

2.3.2 PTA-dfelm-b

A **PTA-dfelm-b** is used for propagating an uncaught exception object. Recall that a *PTA-dfelm*-**b** has the form [*relevant context* ,*exception object*]. *exception object* is either an unknown initial value or a heap-name. Intuitively, *PTA-dfelms*-**b** 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*. For example, at program point **15** in Figure 1.5, the *PTA-dfelm*-**b** [*Empty-context*,$e1_{init}$] is generated to represent the exception thrown by statement **15**. We will see later in Section 2.4.2 how this *PTA-dfelm*-**b** determines the value of the parameter of the catch statement at program point **17**.

2.3.3 Limiting relevant context

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

Theorem 1 *For any PTA-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, Points-to-Algo can use any subset of conjuncts contained in the relevant context without compromising *safety*, although this may cause propagation of spurious *PTA-dfelms* to those 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 *PTA-dfelm*; the rest of the conjuncts are dropped. This bound is imposed uniformly for all *PTA-dfelms*; however, Points-to-Algo allows different bounds for different *PTA-dfelms*.

2.4 Overview of Points-to-Algo

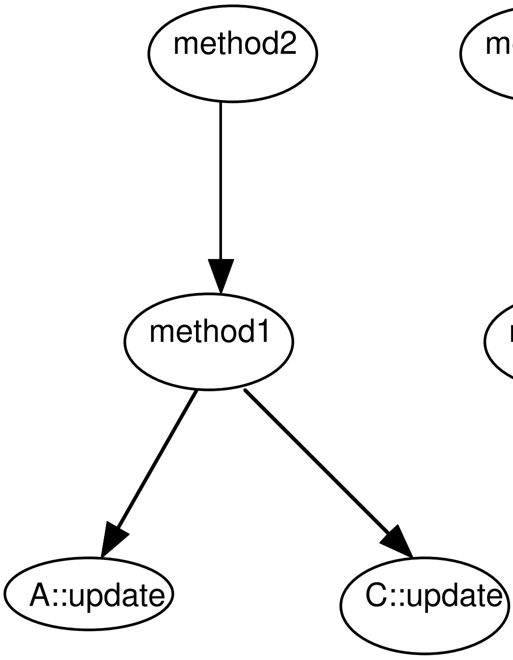

Figure 2.7: SCC-DAG for the example in Figure 1.5

The Points-to-Algo has the following four non-interacting phases:

2.4.1 Phase 0

This is same as described in Section 1.6. For example, the SCC-DAG for the example in Figure 1.5 is shown in Figure 2.7.

2.4.2 Phase I-pta

Points-to-Algo traverses the SCC-DAG (constructed during Phase 0) in a reverse topological order (bottom-up) and analyzes each method assuming parameters and global variables have unknown initial values. For each method Points-to-Algo computes, in terms of unknown initial values, a safe approximation to the method's complete transfer function for points-to analysis. We will call this approximation, the **pointer summary transfer function**. The pointer summary transfer function of a method M is the set of data-flow elements (representing values of pointers) that reach the exit node of M and that do not represent values of the local variables of M. This function summarizes the possible effects of method invocation on pointers. The pointer summary transfer functions of methods in the same SCC have cyclic dependences, so they are computed simultaneously by fixed point iteration. In contrast, the pointer 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 of SCC-DAG, without iteration.

The results of this phase on a SCC are two-fold: (i) a points-to solution at each node of every method in the SCC and (ii) a pointer summary transfer function for every method in the SCC, both of which are parametrized by unknown initial values and the conditions on unknown initial values.

2.4.2.1 Conditional contexts

The effects of a method on pointers is calculated dependent on conditions on (1) the aliasing between unknown initial values of parameters and globals, and (2) the concrete types (run-time types) of these unknown initial values. For example, statement **4** in

Figure 1.5 can modify the *next* field of $a2_{init}$ if *a1* and *a2* are aliased at the entry of *method1*, and hence Phase I-pta infers that at the top of statement **5**, $a2_{init}.next$ points to $a3_{init}$ under the condition that $a1_{init}$ and $a2_{init}$ are the same object at the entry of *method1*. Similarly, statement **5** can invoke *A::update* or *C::update* depending upon the concrete type of $a1_{init}$, and hence Phase I-pta infers that at the top of statement **6**, *global1* points to $object_1$ under the condition that the concrete type of $a1_{init} \in \{A,B\}$, and *global1* points to $object_2$ under the condition that the concrete type of $a1_{init} \in \{C\}$. These conditions are evaluated at a call site of a method, using the actual values of the unknown initial values of the method at the call site, to propagate data-flow elements from the exit of the method to the call site in a context-sensitive manner (i.e., only those instantiations of the data-flow elements are propagated for which the conditions either evaluate to true or the conditions are instantiated into satisfiable conditions on the unknown initial values associated with the entry node of the caller).

2.4.2.2 Relevant contexts

Rather than calculate all possible conditions, Points-to-Algo 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. Points-to-Algo lazily generates those object fields actually **used (read/written)** by this method directly or indirectly through calls; conditions are inferred for only those fields which are *read* in this sense. For example, *method1* in Figure 1.5 writes the *next* field of $a1_{init}$ and reads the *next* field of $a2_{init}$, but it does not use the *next* field of $a3_{init}$. Hence, at statement **4**, Points-to-Algo considers only $a2_{init}.next$ for potential modification and it does not consider $a3_{init}.next$ for potential modification. Intuitively, considering only read fields for potential modification is safe because value propagation within the lifetime of a method can take place only through such fields. If a field that is not read during the lifetime of a callee is read during the lifetime of a caller that invokes the callee, then this field is considered for potential modification in the caller at the call site that invokes the callee. For example, consider the method *caller* shown in Figure 2.8. It invokes *method1* defined in Figure 1.5. At call site **call** that

```
caller( A* s1, A* s2, A* s3 ) {
   call: method1(s1,s2,s3);
   a: s1 = s3→next;
}
```

Figure 2.8: Potential modification at a call site

invokes *method1*, the value of $a3_{init}$ is the unknown initial value $s3_{init}$ and $s3_{init}.next$ is read during the lifetime of *caller*, at program point **a**. Thus $s3_{init}.next$ is considered for potential modification at the call site **call** due to potential update of $s3_{init}.next$ at statement **4**. Further, it is safe to consider only the read fields of unknown initial values for potential modification because for any given invocation of a method, the heap-names allocated at a program point reachable from the entry node of the method represent objects allocated during the lifetime of the method and hence are different from any object in the environment that is active at the entry node of the method when the execution of the method begins. Also note that a potential alias between an unknown initial value u and other unknown initial values is generated only when a field of u is directly modified (i.e., ignoring potential modification due to potential aliasing) during the lifetime of a method and the same field of the other unknown initial values are read during the lifetime of the same method. As a result, even if $a3_{init}.next$ were read during the lifetime of *method1*, still the potential aliasing between $a2_{init}$ and $a3_{init}$ will not be relevant to *method1* because $a2_{init}.next$ is not directly modified during the lifetime of *method1*.

2.4.2.3 Mechanics of lazy generation of fields of unknown initial values

As stated above, Points-to-Algo (and *RCI* in general) lazily generates only those fields of an unknown initial value that are used (read/written) 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. When a field *f* of an unknown initial value is found to be used (read/written) for the first time in *M*, this fact (i.e., whether the field is read or written) 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, until a fixed point is reached and the worklist becomes empty. If through an actual-to-unknown-initial-value binding, access to f causes access to another field for the first time in a non-same-SCC caller, the field in the caller is marked when the corresponding actual-to-unknown initial value binding is computed at a call site in the non-same-SCC caller; thus such marking is done in a hierarchical, bottom-up manner without iteration. For example, in Figure 2.8, the use of $s2_{init}.next$ at call site **call** is implied by the use of $a2_{init}.next$ in *method1*.

2.4.2.4 Propagation of *PTA-dfelms*

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

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

The calculation of the points-to solution in terms of unknown initial values and the calculation of the pointer summary transfer function of a method are accomplished by the propagation of *PTA-dfelms* from method entry to exit on static paths using a worklist algorithm. Initially the solution at each node is empty; it grows monotonically as new *PTA-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. These node transfer functions are described in Section 2.8. In the following paragraphs we will intuitively describe the propagation of *PTA-dfelms* using the example given in Figure 1.5.

2.4.2.5 Examples of propagation of *PTA-dfelms*

Figure 2.9 shows the final Phase I-pta solution computed by Points-to-Algo at the top of each of the statements 3, 4, 5, 6, 7, 8 and 8a contained in *method1* defined in Figure 1.5. The *PTA-dfelms* at program point **3** say that all the variables have the same values as at the entry node of *method1*. Since statement **3** assigns the value of *a1* to *r*, **4:1**, the first *PTA-dfelm* at program point **4**, says that *r* points to the unknown initial value of *a1*. The relevant context of this *PTA-dfelm* is *Empty-context*, which means this *PTA-dfelm* holds in all contexts. Similarly, since statement **4** assigns the value of *a3* to the *next* field of the object to which *a1* points at the entry node of *method1*, **5:1** says that the *next* field of the unknown initial value of *a1* points to the unknown initial value of *a3*. Again, the relevant context is *Empty-context*, meaning the *PTA-dfelm* holds in all contexts.

At an assignment statement, for each set *s* of *PTA-dfelms* that gives the values of the left and right hand sides, the relevant context of the corresponding *PTA-dfelm* (implied by *s*) resulting from the assignment is the conjunction of the relevant contexts of the *PTA-dfelms* of *s*. For statement **4**, there is only one such *s* which consists of **3:1** and **3:3**, and the conjunction of the relevant contexts of the *PTA-dfelms* of *s* is *Empty-context*. As stated earlier, if the unknown initial values of *a1* and *a2* are the same, statement **4** also modifies the *next* field of the unknown initial value of *a2*. The *PTA-dfelms* **5:2** and **5:3** keep track of this potential modification. **5:2** is applicable to only those contexts in which $a1_{init}$ and $a2_{init}$ are equal, while **5:3** is applicable to only those contexts in which $a1_{init}$ and $a2_{init}$ are not equal.

As stated earlier, Points-to-Algo does not consider $a3_{init}.next$ for potential modification at statement **4** because it is not read in *method1* or in any method invoked during the

Solution at program point **3**:
{ 3:1 [*empty*,*Empty-context*,$\langle$*a1*,$a1_{init}\rangle$],
3:2 [*empty*,*Empty-context*,$\langle$*a2*,$a2_{init}\rangle$],
3:3 [*empty*,*Empty-context*,$\langle$*a3*,$a3_{init}\rangle$],
3:4 [*empty*,*Empty-context*,$\langle a1_{init}.next, a1_{init}.next_{init}\rangle$],
3:5 [*empty*,*Empty-context*,$\langle a2_{init}.next, a2_{init}.next_{init}\rangle$],
3:6 [*empty*,*Empty-context*,$\langle$*global1*,$global1_{init}\rangle$],
3:7 [*empty*,*Empty-context*,$\langle$*global2*,$global2_{init}\rangle$],
3:8 [*empty*,*Empty-context*,$\langle$*global3*,$global3_{init}\rangle$] **}**

Solution at program point **4**:
{ 4:1 [*empty*,*Empty-context*,$\langle r, a1_{init}\rangle$] **}**
∪ **{ 3:1** to **3:8 }**

Solution at program point **5**:
{ 5:1 [*empty*,*Empty-context*,$\langle a1_{init}.next, a3_{init}\rangle$],
5:2 [*empty*,$\langle$*(*$a1_{init}$ **eq** $a2_{init}$*),empty,empty*$\rangle$,$\langle a2_{init}.next, a3_{init}\rangle$],
5:3 [*empty*,$\langle$*(*$a1_{init}$ **neq** $a2_{init}$*),empty,empty*$\rangle$,$\langle a2_{init}.next, a2_{init}.next_{init}\rangle$] **}**
∪ **{ 3:1** to **3:3**, **3:6** to **3:8**, **4:1 }**

Solution at program point **6**:
{ 6:1 [*empty*,$\langle$*empty,(type(*$a1_{init}$*)*∈*A::update),empty*$\rangle$,$\langle$*global1*,$object_1\rangle$],
6:2 [*empty*,$\langle$*empty,(type(*$a1_{init}$*)*∈*A::update),empty*$\rangle$,$\langle$*global3*,$object_1\rangle$],
6:3 [*empty*,$\langle$*empty,(type(*$a1_{init}$*)*∈*C::update),empty*$\rangle$,$\langle$*global1*,$object_2\rangle$],
6:4 [*empty*,$\langle$*empty,(type(*$a1_{init}$*)*∈*C::update),empty*$\rangle$,$\langle$*global3*,$object_2\rangle$] **}**
∪ **{ 3:1** to **3:3**, **3:7**, **4:1**, **5:1** to **5:3 }**

Solution at program point **7**:
{ 7:1 [*empty*,$\langle$*(*$a1_{init}$ **eq** $a2_{init}$*),empty,empty*$\rangle$,$\langle$*global2*,$a3_{init}\rangle$],
7:2 [*empty*,$\langle$*(*$a1_{init}$ **neq** $a2_{init}$*),empty,empty*$\rangle$,$\langle$*global2*,$a2_{init}.next_{init}\rangle$] **}**
∪ **{ 3:1** to **3:3**, **4:1**, **5:1** to **5:3**, **6:1** to **6:4 }**

Solution at program point **8**:
{ 8:1 [*empty*,$\langle$*empty,(type(*$a1_{init}$*)*∈*A::update),empty*$\rangle$,$\langle u, object_1\rangle$],
8:2 [*empty*,$\langle$*empty,(type(*$a1_{init}$*)*∈*C::update),empty*$\rangle$,$\langle u, object_2\rangle$] **}**
∪ **{ 3:1** to **3:3**, **4:1**, **5:1** to **5:3**, **6:1** to **6:4**, **7:1**, **7:2 }**

Solution at program point **8a**:
{ 8a:1 [*empty*,*Empty-context*,$\langle$*global1*,$object_8\rangle$] **}**
∪ **{ 3:1** to **3:3**, **4:1**, **5:1** to **5:3**, **6:2**, **6:4**, **7:1**, **7:2**, **8:1**, **8:2 }**

Figure 2.9: Phase I-pta solution for method1

lifetime of *method1*. By generating the fields of unknown initial values lazily, Points-to-Algo infers only *relevant* potential aliases and potential non-aliases. For example, the relationships of $a3_{init}$ with $a1_{init}$ and $a2_{init}$ are not relevant to *method1* because first, $a3_{init}.next$ is not read during the lifetime of *method1* and second, $a2_{init}.next$ is not directly modified during the lifetime of *method1*. Obviously a potential alias between an unknown initial value u and other unknown initial values is generated only when a field of u is directly modified (i.e., ignoring potential modification due to potential aliasing) during the lifetime of a method and the same field of the other unknown initial values are read during the lifetime of the same method. As a result, even if $a3_{init}.next$ were read during the lifetime of *method1*, still the potential aliasing between $a2_{init}$ and $a3_{init}$ will not be relevant to *method1* because $a2_{init}.next$ is not directly modified during the lifetime of *method1*.

Statement **5** is a dynamically dispatched call site. It can invoke either *A::update* or *C::update* depending upon the concrete type of $a1_{init}$. As a result, Points-to-Algo computes the values of *global1* and *global3* at program point **6** conditioned on the potential concrete types of $a1_{init}$. The *PTA-dfelm* **6:1** says that *global1* points to $object_1$ in those contexts in which the concrete type of $a1_{init}$ belongs to the set of types represented by *A::update* (i.e., $\{A,B\}$). The meanings of **6:2**, **6:3** and **6:4** are similar. As before with potential aliases, Points-to-Algo only infers relevant potential concrete types; for example, the concrete types of $a2_{init}$ and $a3_{init}$ are not needed for points-to analysis of *method1* because there is no dynamic dispatch based on these values.

The *PTA-dfelms*-**a** **7:1** and **7:2** are implied respectively by **5:2** and **5:3**, and the *PTA-dfelms*-**a** **8:1** and **8:2** are implied respectively by **6:1** and **6:3**.

The pointer summary transfer function of *method1* consists of the *PTA-dfelms* at program point **8a** except **3:1**, **3:2**, **3:3**, **4:1**, **8:1** and **8:2**, as these *PTA-dfelms* represent values of variables local to *method1*.

Now consider the call sites **9** and **12** that invoke *method1* from *method2*. At call site 9, Points-to-Algo translates the *PTA-dfelms* at the exit node of *method1* by replacing

[*empty*,*Empty-context*,$\langle$*global1*,$object_8\rangle$]
[*empty*,*Empty-context*,$\langle a4_{init}.next, a6_{init}\rangle$]
[*empty*,$\langle$($a4_{init}$ **eq** $a5_{init}$),*empty*,*empty*$\rangle$,$\langle a5_{init}.next, a6_{init}\rangle$]
[*empty*,$\langle$($a4_{init}$ **neq** $a5_{init}$),*empty*,*empty*$\rangle$,$\langle a5_{init}.next, a5_{init}.next_{init}\rangle$]
[*empty*,$\langle$*empty*,(type($a4_{init}$)$\in$A::update),*empty*$\rangle$,$\langle$*global3*,$object_1\rangle$]
[*empty*,$\langle$*empty*,(type($a4_{init}$)$\in$C::update),*empty*$\rangle$,$\langle$*global3*,$object_2\rangle$]
[*empty*,$\langle$($a4_{init}$ **eq** $a5_{init}$),*empty*,*empty*$\rangle$,$\langle$*global2*,$a6_{init}\rangle$]
[*empty*,$\langle$($a4_{init}$ **neq** $a5_{init}$),*empty*,*empty*$\rangle$,$\langle$*global2*,$a5_{init}.next_{init}\rangle$]

Figure 2.10: Instantiated *PTA-dfelms* at call site **9**

[*empty*,*Empty-context*,$\langle$*global1*,$object_8\rangle$]
[*empty*,*Empty-context*,$\langle object_{10}.next, object_{11}\rangle$]
[*empty*,*Empty-context*,$\langle object_{11}.next, null\rangle$]
[*empty*,*Empty-context*,$\langle$*global3*,$object_1\rangle$]
[*empty*,*Empty-context*,$\langle$*global2*,*null*$\rangle$]

Figure 2.11: Instantiated *PTA-dfelms* at call site **12**

the unknown initial values of *a1*, *a2* and *a3* with their values at call site 9 (i.e., the unknown initial values of *a4*, *a5* and *a6* respectively). For example:

- **7:1** [*empty*,$\langle$($a1_{init}$ **eq** $a2_{init}$),*empty*,*empty*$\rangle$,$\langle$*global2*,$a3_{init}\rangle$]
  is translated to
  [*empty*,$\langle$($a4_{init}$ **eq** $a5_{init}$),*empty*,*empty*$\rangle$,$\langle$*global2*,$a6_{init}\rangle$].

- **6:4** [*empty*,$\langle$*empty*,(type($a1_{init}$)$\in$C::update),*empty*$\rangle$,$\langle$*global3*,$object_2\rangle$]
  is translated to
  [*empty*,$\langle$*empty*,(type($a4_{init}$)$\in$C::update),*empty*$\rangle$,$\langle$*global3*,$object_2\rangle$].

Figure 2.10 shows all the resulting *PTA-dfelms* when the unknown initial values in the *PTA-dfelms* contained in the pointer summary transfer function of *method1* are instantiated by their values at call site **9**.

At call site **12**, Points-to-Algo instantiates *PTA-dfelms* contained in the pointer summary transfer function of *method1* using $object_{10}$ as $a1_{init}$ and $object_{11}$ as $a2_{init}$ and $a3_{init}$. For example:

- **7:1** [*empty*,$\langle$($a1_{init}$ **eq** $a2_{init}$),*empty*,*empty*$\rangle$,$\langle$*global2*,$a3_{init}\rangle$]

is not applicable to this call site as ($object_{10}$ **eq** $object_{11}$) is false.

- **7:2** [*empty*,⟨($a1_{init}$ **neq** $a2_{init}$)*,empty,empty*⟩,⟨*global2*,$a2_{init}.next_{init}$⟩] **}**
  is translated to
  [*empty*,*Empty-context*,⟨*global2,null*⟩]
  because ($object_{10}$ **neq** $object_{11}$) is *true* and the value of $object_{11}.next$ is *null*
  at program point **12**.

- **6:2** [*empty*,⟨*empty*,(*type*($a1_{init}$)∈*A::update*),*empty*⟩,⟨*global3*,$object_1$⟩]
  is translated to
  [*empty*,*Empty-context*,⟨*global3*,$object_1$⟩]
  because B, the concrete type of $a1_{init}$, belongs to A::update, i.e., {A,B}.

Figure 2.11 shows all the resulting *PTA-dfelms* when the unknown initial values in the *PTA-dfelms* contained in the pointer summary transfer function of *method1* are instantiated by their values at call site **12**.

At each call site Points-to-Algo stores the bindings between the actuals and the unknown initial values of the methods invocable from the call site. Each binding has the form [*ecfi*,*rc*,⟨*actual,uiv-at-the-entry-of-a-target*⟩], where *ecfi* and *rc* are respectively an *ECFInfo* and relevant context under which the binding holds. *ecfi* **will be non-empty only at a call site directly contained in a finally statement**. For example, consider the call site **12** in Figure 1.5. [*empty*,*Empty-context*,⟨$object_{11}$,$a2_{init}$⟩] and [*empty*,*Empty-context*,⟨$object_{11}$,$a3_{init}$⟩] are two of the actual-to-uiv bindings computed at this call site. When the actual-to-uiv bindings are used for replacing unknown initial values with actuals in a *PTA-dfelm d1*, each resulting *PTA-dfelm d2* is associated with a new relevant context, that is the conjunction of the relevant contexts of the bindings used in generating *d2* and the relevant context of *d1*, instantiated with the actuals.

2.4.2.6 Examples of propagation of *PTA-dfelms* in the presence of exceptions

Consider *method3* defined in Figure 1.5. The hierarchy of exception types used in the example in Figure 1.5 is shown in Figure 2.13. Figure 2.12 shows the Phase I-pta solution computed by Points-to-Algo at the top of statements 14, 15, 16, 17, 18 and 19.

Solution at program point **14**:
{ 14:1 [*empty*,*Empty-context*,$\langle$*global3*,$global3_{init}$$\rangle$],
14:2 [*empty*,*Empty-context*,$\langle$*e1*,$e1_{init}$$\rangle$] **}**

Solution at program point **15**:
{ 15:1 [*empty*,*Empty-context*,$\langle$*global3*,$object_{14}$$\rangle$] **}**
$\cup$ **{ 14:2 }**

Solution at program point **16**:
{ 16:1 [*Empty-context*,$e1_{init}$],
16:2 [$e1_{init}$,*Empty-context*,$\langle$*global3*,$object_{14}$$\rangle$],
16:3 [$e1_{init}$,*Empty-context*,$\langle$*e1*,$e1_{init}$$\rangle$] **}**

Solution at program point **17**:
{ 16:1 to **16:3 }**

Solution at program point **18**:
{ 18:1 [*empty*,$\langle$*empty,empty,(type(*$e1_{init}$*)* $\leq$ *ET2)*$\rangle$,$\langle$*excp*,$e1_{init}$$\rangle$],
18:2 [*empty*,$\langle$*empty,empty,(type(*$e1_{init}$*)* $\leq$ *ET2)*$\rangle$,$\langle$*global3*,$object_{14}$$\rangle$],
18:3 [*empty*,$\langle$*empty,empty,(type(*$e1_{init}$*)* $\leq$ *ET2)*$\rangle$,$\langle$*e1*,$e1_{init}$$\rangle$] **}**

Solution at program point **19**:
{ 19:1 [$\langle$*empty,empty,(type(*$e1_{init}$*)* $\not\leq$ *ET2)*$\rangle$,$e1_{init}$],
19:2 [$e1_{init}$,$\langle$*empty,empty,(type(*$e1_{init}$*)* $\not\leq$ *ET2)*$\rangle$,$\langle$*global3*,$object_{14}$$\rangle$],
19:3 [$e1_{init}$,$\langle$*empty,empty,(type(*$e1_{init}$*)* $\not\leq$ *ET2)*$\rangle$,$\langle$*e1*,$e1_{init}$$\rangle$],
19:4 [*empty*,$\langle$*empty,empty,(type(*$e1_{init}$*)* $\leq$ *ET2)*$\rangle$,$\langle$*global3*,$object_{14}$$\rangle$],
19:5 [*empty*,$\langle$*empty,empty,(type(*$e1_{init}$*)* $\leq$ *ET2)*$\rangle$,$\langle$*u1*,$object_{14}$$\rangle$],
19:6 [*empty*,$\langle$*empty,empty,(type(*$e1_{init}$*)* $\leq$ *ET2)*$\rangle$,$\langle$*e1*,$e1_{init}$$\rangle$] **}**

Figure 2.12: Phase I-pta solution for method3

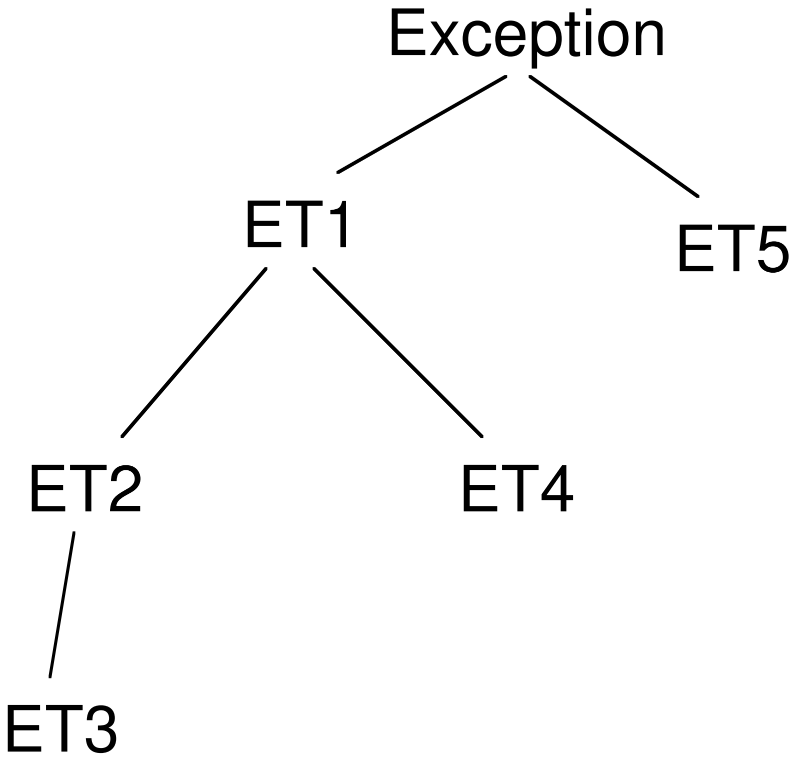

Figure 2.13: Hierarchy of exception types

The two *PTA-dfelms* at program point **14** (i.e., **14:1** and **14:2**) represent initial values of *global3* and *e1* respectively. The two *PTA-dfelms* at program point **15** also represent values of *global3* and *e1* respectively. **15:1** results from the assignment at program point **14**. Statement **15** throws the object to which *e1* points as the exception object. As a result, **16:1**, **16:2** and **16:3** are propagated to statement **16**, the exit node of the try statement. The *PTA-dfelm*-**b** **16:1** represents the exception thrown at program point **15**. The *PTA-dfelms*-**a** **16:2** and **16:3** are generated from **15:1** and **14:2** by the throw statement. Points-to-Algo uses the *ECFInfo*'s of the *PTA-dfelms* at the exit node of a try statement to decide the further propagation of the *PTA-dfelms* from the exit node. A *PTA-dfelm*-**a** is propagated to a catch statement if the exception represented by the *ECFInfo* of the *PTA-dfelm*-**a** can be caught by the catch statement. Similarly, a *PTA-dfelm*-**b** is propagated to a catch statement if the exception represented by the *PTA-dfelm*-**b** can be caught by the catch statement. As a result, **16:1**, **16:2** and **16:3** are propagated to the catch at program point **17**.

Now consider the three *PTA-dfelms*-**a** at program point **18**. Since the *ECFInfo*'s of **16:2** and **16:3** are both $e1_{init}$ and the exception is caught by the *catch* statement if and only if the type of the concrete value of $e1_{init}$ is *ET2* or a subtype of *ET2*, **18:2** and **18:3** are generated from **16:2** and **16:3** respectively. The *PTA-dfelm*-**b** **16:1** represents an uncaught exception, so it is used to instantiate the parameter of the *catch* statement. As a result, the *PTA-dfelm*-**a** **18:1** is generated.

Next, consider the solution at program point **19**. The *PTA-dfelms*-**a** **19:4**, **19:5** and **19:6** are propagated to program point **19** from the exit node of the catch statement in an obvious manner. The *PTA-dfelms* **19:1**, **19:2** and **19:3** are propagated to program point **19** from program point **16**. These are generated from **16:1**, **16:2** and **16:3** respectively. The *exception type contexts* of these *PTA-dfelms* indicate that the exception thrown at program point **15** escapes the catch statement if the type of the concrete value of $e1_{init}$ is neither *ET2* nor a subtype of *ET2*.

Now consider the Phase I-pta solution for *method4*. Figures 2.14 and 2.15 show the Phase I-pta solution at a few program points in *method4*.

Solution at program point **21**:
{ 21:1 [$empty$,$Empty$-$context$,$\langle global3, global3_{init}\rangle$],
21:2 [$empty$,$Empty$-$context$,$\langle e3, object_{20}\rangle$],
21:3 [$empty$,$Empty$-$context$,$\langle e2, e2_{init}\rangle$] **}**

Solution at program point **21a**:
{ 21a:1 [$ET4$,$Empty$-$context$,$\langle global3, object_{14}\rangle$],
21a:2 [$empty$,$Empty$-$context$,$\langle e3, object_{20}\rangle$],
21a:3 [$ET4$,$Empty$-$context$,$\langle e3, object_{20}\rangle$],
21a:4 [$empty$,$Empty$-$context$,$\langle e2, e2_{init}\rangle$],
21a:5 [$ET4$,$Empty$-$context$,$\langle e2, e2_{init}\rangle$],
21a:6 [$Empty$-$context$,$object_{20}$] **}**

Solution at program point **23**:
{ 23:1 [$empty$,$Empty$-$context$,$\langle global3, object_{14}\rangle$],
23:2 [$empty$,$Empty$-$context$,$\langle e3, object_{20}\rangle$],
23:3 [$empty$,$Empty$-$context$,$\langle e2, e2_{init}\rangle$],
23:4 [$empty$,$Empty$-$context$,$\langle excp1, object_{20}\rangle$] **}**

Figure 2.14: Phase I-pta solution for method4

Solution at program point **24**:
{ 24:1 [*empty*,*Empty-context*,$\langle$*global3*,$global3_{init}\rangle$],
24:2 [*empty*,*Empty-context*,$\langle$*e4*,$e2_{init}\rangle$],
24:3 [*empty*,*Empty-context*,$\langle$*e2*,$e2_{init}\rangle$] **}**

Solution at program point **24a**:
{ 24a:1 [$e2_{init}$,$\langle$*empty,empty,(type*($e2_{init}$) $\not\leq$ *ET2)*$\rangle$,$\langle$*global3*,$object_{14}\rangle$],
24a:2 [*empty*,$\langle$*empty,empty,(type*($e2_{init}$) $\leq$ *ET2)*$\rangle$,$\langle$*global3*,$object_{14}\rangle$],
24a:3 [*empty*,*Empty-context*,$\langle$*e4*,$e2_{init}\rangle$],
24a:4 [$e2_{init}$,$\langle$*empty,empty,(type*($e2_{init}$) $\not\leq$ *ET2)*$\rangle$,$\langle$*e4*,$e2_{init}\rangle$],
24a:5 [*empty*,*Empty-context*,$\langle$*e2*,$e2_{init}\rangle$],
24a:6 [$e2_{init}$,$\langle$*empty,empty,(type*($e2_{init}$) $\not\leq$ *ET2)*$\rangle$,$\langle$*e2*,$e2_{init}\rangle$],
24a:7 [$\langle$*empty,empty,(type*($e2_{init}$) $\not\leq$ *ET2)*$\rangle$,$e2_{init}$] **}**

Solution at program point **26**:
26:1 [*empty*,$\langle$*empty,empty,(type*($e2_{init}$) $\leq$ *ET4)*$\rangle$,$\langle$*global3*,$object_{14}\rangle$],
26:2 [*empty*,$\langle$*empty,empty,(type*($e2_{init}$) $\leq$ *ET4)*$\rangle$,$\langle$*e4*,$e2_{init}\rangle$],
26:3 [*empty*,$\langle$*empty,empty,(type*($e2_{init}$) $\leq$ *ET4)*$\rangle$,$\langle$*e2*,$e2_{init}\rangle$],
26:4 [*empty*,$\langle$*empty,empty,(type*($e2_{init}$) $\leq$ *ET4)*$\rangle$,$\langle$*excp2*,$e2_{init}\rangle$] **}**

Solution at program point **28**:
28:1 [*empty*,$\langle$*empty,empty,(type*($e2_{init}$) $\not\leq$ *ET2)*$\wedge$*(type*($e2_{init}$) $\not\leq$ *ET4)*$\rangle$,$\langle$*global3*,$object_{14}\rangle$],
28:2 [*empty*,$\langle$*empty,empty,(type*($e2_{init}$) $\not\leq$ *ET2)*$\wedge$*(type*($e2_{init}$) $\not\leq$ *ET4)*$\rangle$,$\langle$*e4*,$e2_{init}\rangle$],
28:3 [*empty*,$\langle$*empty,empty,(type*($e2_{init}$) $\not\leq$ *ET2)*$\wedge$*(type*($e2_{init}$) $\not\leq$ *ET4)*$\rangle$,$\langle$*e2*,$e2_{init}\rangle$],
28:4 [*empty*,$\langle$*empty,empty,(type*($e2_{init}$) $\not\leq$ *ET2)*$\wedge$*(type*($e2_{init}$) $\not\leq$ *ET4)*$\rangle$,$\langle$*excp3*,$e2_{init}\rangle$] **}**

Figure 2.15: Phase I-pta solution for method4

First, consider the solution at program point **21**. The *PTA-dfelms*-**a** **21:1** and **21:3** respectively represent the unknown initial values of *global3* and *e2* reaching program point **21** from the entry node of *method4*. The *PTA-dfelm*-**a** **21:2** results from the assignment to *e3* at program point **20**.

Next consider the solution at program point **21a**. The value of $e1_{init}$ at call site **21** is $object_{20}$, whose type is *ET4*. As a result, **19:2** is instantiated into **21a:1**. The relevant context of **21a:1** is *Empty-context* because *(type($object_{20}$) $\not\leq$ ET2)* is true and thus the relevant context of **19:2** evaluates to true after the instantiation. **21a:6** is generated from **19:1** in a similar manner. **21a:2** and **21a:4** are respectively generated from **21:2** and **21:3** because there are possible invocation contexts of *method3* in which the exit node of *method3* is reachable from the entry node of *method3* along a path without any uncaught exception. For example, if the concrete type of $e1_{init}$ is *ET2*, then the exception thrown at program point **15** will be caught at program point **17** and the exit node of *method3* will be reached without any uncaught exception. Although this invocation context is not applicable to call site **21**, the decision to retain the *ECFInfo* of a *PTA-dfelm*-**a** representing the value of a local variable across a call site is made based on whether the exit node of the target is reachable from the entry node of the target along a path without any uncaught exception in any invocation context. This results in a safe solution in all cases and in a precise solution in those cases where the problem is polynomial-time solvable (see Chapter 3). **21a:3** and **21a:5** are respectively generated from **21:2** and **21:3** because *method3* throws $e1_{init}$ as exception object and the concrete type of $e1_{init}$ at call site **21** is *ET4*. The *ECFInfo*'s of **21a:3** and **21a:5** store the signature of the exception thrown by *method3*.

Now consider the solution at program point **23**. The exception thrown by the call at program point **21** is caught by the catch at program point **22**. As a result, **23:1**, **23:2** and **23:3** are respectively generated from **21a:1**, **21a:3** and **21a:5**. The *ECFInfo*'s of **23:1**, **23:2** and **23:3** are *empty* because the exception of concrete type *ET4* has been caught by the catch at program point **22**. **23:4** is generated from **21a:6** because the exception object caught by the catch at program point **22** is assigned to the parameter of the catch (i.e., *excp1*).

Next consider the solution at program point **24**. The *PTA-dfelms*-**a** **24:1** and **24:3** respectively represent the unknown initial values of *global3* and *e2* reaching program point **24** from the entry node of *method4*. The *PTA-dfelm*-**a** **24:2** results from the assignment to *e4* at program point **23b**.

Now consider the solution at program point **24a**. The value of $e1_{init}$ at call site **24** is $e2_{init}$. As a result, **19:2**, **19:4** and **19:1** are respectively instantiated into **24a:1**, **24a:2** and **24a:7**. **24a:3** and **24a:5** are respectively generated from **24:2** and **24:3** because (as stated above) there are possible invocation contexts of *method3* in which the exit node of *method3* is reachable from the entry node of *method3* along a path without any uncaught exception. **24a:4** and **24a:6** are respectively generated from **24:2** and **24:3** because *method3* throws $e1_{init}$ as exception object under the context *(type(*$e1_{init}$*)* $\not\leq$ *ET2)* and the value of $e1_{init}$ at call site **24** is $e2_{init}$. The *ECFInfo*'s of **24a:4** and **24a:6** store the signature of the exception thrown by *method3*.

Next consider the solution at program point **26**. The exception object $e2_{init}$ thrown by the call at program point **24** is caught by the catch at program point **25** under the type constraint *(type(*$e2_{init}$*)* $\not\leq$ *ET2)* $\wedge$ *(type(*$e2_{init}$*)* $\leq$ *ET4)*. However, *(type(*$e2_{init}$*)* $\not\leq$ *ET2)* $\wedge$ *(type(*$e2_{init}$*)* $\leq$ *ET4)* is equivalent to *(type(*$e2_{init}$*)* $\leq$ *ET4)* because *(type(*$e2_{init}$*)* $\leq$ *ET4)* implies *(type(*$e2_{init}$*)* $\not\leq$ *ET2)*. As a result, **26:1**, **26:2** and **26:3** are respectively generated from **24a:1**, **24a:4** and **24a:6**. The *ECFInfo*'s of **26:1**, **26:2** and **26:3** are *empty* because the exception has been caught by the catch at program point **25**. **26:4** is generated from **24a:7** because the exception object caught by the catch at program point **25** is assigned to the parameter of the catch (i.e., *excp2*).

Finally consider the solution at program point **28**. The exception object $e2_{init}$ thrown by the call at program point **24** is caught by the catch at program point **27** under the type constraint *(type(*$e2_{init}$*)* $\not\leq$ *ET2)* $\wedge$ *(type(*$e2_{init}$*)* $\not\leq$ *ET4)* because the exception thrown at program point **15** has to escape the catches at program points **17** and **25** to be caught by the catch at program point **27**. As a result, **28:1**, **28:2** and **28:3** are respectively generated from **24a:1**, **24a:4** and **24a:6**. The *ECFInfo*'s of **28:1**, **28:2** and **28:3** are *empty* because the exception has been caught by the catch at program point

27. **28:4** is generated from **24a:7** because the exception object caught by the catch at program point **27** is assigned to the parameter of the catch (i.e., *excp3*).

2.4.2.7 Examples of propagation in the presence of finally statements

Consider *method5* and *method6* defined in Figure 1.5. First, we will give a brief overview of control flow through methods *method5* and *method6* and then, we will describe the Phase I-pta solutions for these methods.

method6 invokes *method5* from the call site **35**. *method5* allocates an exception object of type *ET1* at program point **31** and then throws this exception object at program point **32**. This exception is not caught in *method5* and it is propagated to the caller, in this case *method6*. The catch statement at program point **36** cannot catch this exception as *ET1* is not a subtype of *ET5*. Thus, the control flow goes to the finally statement at program point **37**. The finally statement at program point **37** contains a try statement nested inside it at program point **38**. This nested try statement throws an exception of type *ET1* at program point **40**. Since the catch statement at program point **41** catches all exceptions of type *ET1* or a subtype of *ET1*, the exception thrown at program point **40** is propagated to the catch statement at program point **41**. Thus at program point **41** there are two active exceptions: one thrown at program point **32** and the other thrown at program point **40**. The exception thrown at program point **40** is caught by the catch statement at program point **41**. After this, the control flow goes to the entry node of the finally statement at program point **42**. After the exit from the finally statement at program point **42**, the control flow goes to program point **44**, where only the exception thrown at program point **32** is active.

A part of the Phase I-pta solution for *method5* is shown below (we are ignoring part of the solution for the ease of presentation):

---

private: static void method5(A *param) {

```
ET1 *unexp;
```

30: `global1 = param;`

31: `unexp = new ET1();`

`/***` **32:1** [*empty*,*Empty-context*,⟨$global1,param_{init}$⟩] `***/`

`/***` **32:2** [*empty*,*Empty-context*,⟨$unexp,object_{31}$⟩] `***/`

32: `throw unexp;`

`/***` **33:1** [*Empty-context*,$object_{31}$] `***/`

`/***` **33:2** [*ET1*,*Empty-context*,⟨$global1,param_{init}$⟩] `***/`

`/***` **33:3** [*ET1*,*Empty-context*,⟨$unexp,object_{31}$⟩] `***/`

33: }

The throw statement at program point **32** throws $object_{31}$, whose run-time type is *ET1*. As a result, the *PTA-dfelms*-**a** **33:2** and **33:3** are generated from the *PTA-dfelms*-**a** **32:1** and **32:2** respectively, and the *ECFInfo*'s of **33:2** and **33:3** are *ET1*, the signature of the exception thrown at program point **32** (note the difference with the *PTA-dfelms*-**a** **16:2** and **16:3** defined in Figure 2.12). The *PTA-dfelm*-**b** **33:1** represents the exception thrown at program point **32**.

Next, consider the part of the Phase I-pta solution for *method6* shown below:

---

public: static void method6( ) {

`A *local;`

`ET1 *unexp1;`

34a: `unexp1 = 0;`

34: `local = new A();`

`try {`

`/***` **35:1** [*empty*,*Empty-context*,⟨$local,object_{34}$⟩] `***/`

`/***` **35:2** [*empty*,*Empty-context*,⟨*unexp1*,*null*⟩] `***/`

35: `method5( local );`

`/***` **35a:1** [*Empty-context*,$object_{31}$] `***/`

/*** **35a:2** [$ET1$,$Empty\text{-}context$,$\langle global1,object_{34}\rangle$] ***/

/*** **35a:3** [$ET1$,$Empty\text{-}context$,$\langle local,object_{34}\rangle$] ***/

/*** **35a:4** [$ET1$,$Empty\text{-}context$,$\langle unexp1,null\rangle$] ***/

35a: }

36: catch(ET5 *param1) {

}

37: finally {

/*** **38:1** [$Empty\text{-}context$,$object_{31}$] ***/

/*** **38:2** [$ET1$,$Empty\text{-}context$,$\langle global1,object_{34}\rangle$] ***/

/*** **38:3** [$ET1$,$Empty\text{-}context$,$\langle local,object_{34}\rangle$] ***/

/*** **38:4** [$ET1$,$Empty\text{-}context$,$\langle unexp1,null\rangle$] ***/

/*** actual-to-uiv-bindings:

38:5 [$ET1$,$Empty\text{-}context$,$\langle object_{34},38::global1_{init}\rangle$]

38:6 [$ET1$,$Empty\text{-}context$,$\langle object_{34},38::local_{init}\rangle$]

38:7 [$ET1$,$Empty\text{-}context$,$\langle null,38::unexp1_{init}\rangle$].

Here 38::$global1_{init}$, 38::$local_{init}$ and 38::$unexp1_{init}$ are respectively the unknown initial values of global1, local and unexp1 associated with program point **38**. ***/

38: try {

/*** **39:1** [$empty$,$Empty\text{-}context$,$\langle global1,38::global1_{init}\rangle$] ***/

/*** **39:2** [$empty$,$Empty\text{-}context$,$\langle local,38::local_{init}\rangle$] ***/

/*** **39:3** [$empty$,$Empty\text{-}context$,$\langle unexp1,38::unexp1_{init}\rangle$] ***/

39: unexp1 = new ET1();

/*** **40:1** [$empty$,$Empty\text{-}context$,$\langle global1,38::global1_{init}\rangle$] ***/

/*** **40:2** [$empty$,$Empty\text{-}context$,$\langle local,38::local_{init}\rangle$] ***/

/*** **40:3** [$empty$,$Empty\text{-}context$,$\langle unexp1,object_{39}\rangle$] ***/

40: throw unexp1;

/*** **40a:1** [$ET1$,$Empty\text{-}context$,$\langle global1,38::global1_{init}\rangle$] ***/

/*** **40a:2** [$ET1$,$Empty\text{-}context$,$\langle local,38::local_{init}\rangle$] ***/

/*** **40a:3** [$ET1$,$Empty\text{-}context$,$\langle unexp1,object_{39}\rangle$] ***/

```
    /*** 40a:4 [Empty-context,object39] ***/
  40a:  }
  /*** 41:1 [ET1,Empty-context,⟨global1,38::global1init⟩] ***/
  /*** 41:2 [ET1,Empty-context,⟨local,38::localinit⟩] ***/
  /*** 41:3 [ET1,Empty-context,⟨unexp1,object39⟩] ***/
  /*** 41:4 [Empty-context,object39] ***/
  41:  catch( ET1 *param2 ) {
    /*** 41a:1 [empty,Empty-context,⟨global1,38::global1init⟩] ***/
    /*** 41a:2 [empty,Empty-context,⟨local,38::localinit⟩] ***/
    /*** 41a:3 [empty,Empty-context,⟨unexp1,object39⟩] ***/
    /*** 41a:4 [empty,Empty-context,⟨param2,object39⟩] ***/
  41a:  }
  42:  finally {
    /*** 43:1 [44,Empty-context,⟨global1,38::global1init⟩] ***/
    /*** 43:2 [44,Empty-context,⟨local,38::localinit⟩] ***/
    /*** 43:3 [44,Empty-context,⟨unexp1,object39⟩] ***/
    43:
  }
/*** 44:1 [Empty-context,object31] ***/
/*** 44:2 [ET1,Empty-context,⟨global1,object34⟩] ***/
/*** 44:3 [ET1,Empty-context,⟨local,object34⟩] ***/
/*** 44:4 [ET1,Empty-context,⟨unexp1,object39⟩] ***/
44:  } // End of outer finally

/*** 45:1 [Empty-context,object31] ***/
/*** 45:2 [ET1,Empty-context,⟨global1,object34⟩] ***/
/*** 45:3 [ET1,Empty-context,⟨local,object34⟩] ***/
/*** 45:4 [ET1,Empty-context,⟨unexp1,object39⟩] ***/
45:  }
```

The *PTA-dfelm*-**a** **35:1** implies the actual-to-uiv binding

$[empty, Empty\text{-}context, \langle object_{34}, param_{init} \rangle]$

at call site **35**. As a result, the *PTA-dfelm*-**a** **33:2** is instantiated into the *PTA-dfelm*-**a** **35a:2**. The *PTA-dfelm*-**a** **35a:1** is generated from the *PTA-dfelm*-**a** **33:1** is an obvious manner. Since the exit node of *method5* is reachable from the entry node of *method5* only along a path that throws an uncaught exception of run-time type *ET1*, **35:1** and **35:2** are respectively transformed into **35a:3** and **35a:4** as **35:1** and **35:2** are propagated across the call site **35**.

Since the catch statement at program point **36** cannot catch an exception of type *ET1*, **35a:1**, **35a:2**, **35a:3** and **35a:4** are propagated to the entry node of the finally statement at program point **37**, and **38:1**, **38:2**, **38:3** and **38:4** reach program point **38**. A try statement directly contained in a finally statement is treated like a call to an anonymous procedure because it can cause exceptions (and other reasons for entering a finally statement) to stack up. In order to do analysis inside the anonymous procedure, the global variables in the *used-set* of the method containing the try statement and the all the local variables of the method containing the try statement are assumed to have unknown initial values at the entry node of the nested try statement. The bindings between the values of the globals and locals and their unknown initial values associated with the entry node of the try statement are stored as actual-to-uiv bindings at the call to the anonymous procedure. Thus, the try statement at program point **38** is treated like a call to an anonymous procedure. The actual-to-uiv bindings **38:5**, **38:6** and **38:7** store the bindings between the actuals and the unknown initial values of *global1*, *local* and *unexp1* (i.e., $38{::}global1_{init}$, $38{::}local_{init}$ and $38{::}unexp1_{init}$) at this call site. These actual-to-uiv bindings are used for instantiating $38{::}global1_{init}$, $38{::}local_{init}$ and $38{::}unexp1_{init}$ in *PTA-dfelms* when these *PTA-dfelms* are propagated from the nested try statement (at program point **38**) into the outer finally statement (that starts at program point **37**). This way, the stack of exceptions (and other reasons for entering a finally statement) is maintained implicitly by remembering them in the corresponding actual-to-uiv bindings. For example, at program point **40a** there are two active exceptions: one thrown at program point **32** and the other thrown at program point **40**. However, the *ECFInfo*'s of the *PTA-dfelms* at program point **40a** need to store only the signature of the exception thrown at program point **40** as the signature of the other exception is stored in the actual-to-uiv bindings at program point **38**. The exception thrown at program point **40** is caught by the catch statement at program point **41**. As a result, **41a:1** to **41a:4** are generated from **41:1** to **41:4**. **41a:1** to **41a:3** are propagated from the program point **41a** to the entry node of the finally statement at program point **42**. As a result, **43:1** to **43:3** are propagated to program point **43**. Note that the *ECFInfo*'s of **43:1** to **43:3** are the label *44* because the finally statement has been entered due to falling through the catch statement at program point **41** and hence the control should go to program point **44** from the exit of the finally statement at program point **42**. Recall that an *ECFInfo* remembers the reason for entering a finally statement, which in this case happens to be falling through the catch statement at program point **41**. At program point **43**, the actual-to-uiv bindings **38:5** and **38:6** are used for instantiating $38{::}global1_{init}$ and $38{::}local_{init}$ in **43:1** and **43:2**. As a result, **44:2** and **44:3** are generated from **43:1** and **43:2** respectively. Note that the *ECFInfo*'s of **44:2** and **44:3** are *ET1* because the *ECFInfo*'s of the actual-to-uiv bindings **38:5** and **38:6** are *ET1*. **44:4** is generated from **43:3** because program point **38** is reachable only along a path that has an uncaught exception of run-time type *ET1* (thrown at program point **32**). **44:1** is generated from **38:1** because program point **43** is reachable from program point **38** along a path without any uncaught exception or incomplete, pending transition to a statement outside the outer finally statement.

Although the example in Figure 1.5 has recursive types, only a finite number of unknown initial values are accessed in it, so representative names are not needed.

2.4.3 Phase II-pta

In this phase, Points-to-Algo traverses the SCC-DAG in a topological order (top-down) and propagates the *concrete values* of unknown initial values to the entry nodes of methods. A concrete value of an unknown initial value is a heap-name or an interface initial value. For example, in Figure 1.5, Phase II-pta propagates $object_{10}$ from call site **12** to the entry of *method1* as a concrete value of $a1_{init}$. Points-to-Algo treats an interface initial value like a concrete value and propagates an **interface initial value** to the entry nodes of other methods if, through an actual-to-uiv binding at a call site, the interface initial value is the value of an unknown initial value at the entry of a target. For example, at call site **9** in Figure 1.5, Points-to-Algo propagates the interface initial value $a4_{init}$ to the entry of *method1* as a concrete value of $a1_{init}$.

The propagation within an SCC is done iteratively until a fixed point is reached, while propagation across SCC's is done in a top-down manner without iteration.

This phase involves only the entry nodes and call nodes, and as the empirical results in Chapter 5 show, it is extremely fast.

Figure 2.16 shows the Phase II-pta solution at the entry node of *method1* defined in Figure 1.5. Each concrete value-to-uiv binding computed at the entry node of a method has the form:

[*rc*,*cv*,*uiv*],

where *cv* is the concrete value of an unknown initial value *uiv* at the entry node of a method and *rc* is either *Empty-context* or it is a relevant context that involves only the interface initial values of a single *public interface method* and that can evaluate to true by making conservative, worst-case assumptions about the interface initial values, meaning the binding holds at the entry node of the method in those contexts in which *rc* holds at the entry node of the *public interface method*. For example, consider the relevant context ⟨*empty*,*(type(a4)∈A::update)*,*empty*⟩ of **method1:17** in Figure 2.16. It involves only the interface initial value $a4_{init}$ of a single *public interface method method2* and it can evaluate to true by making conservative, worst-case assumption about $a4_{init}$. On the other hand, if the declared type of $a4_{init}$ were C instead of A, then the relevant context ⟨*empty*,*(type(a4)∈A::update)*,*empty*⟩ would not evaluate to true by making any assumption about $a4_{init}$. Note that in general to name an unknown initial value uniquely, we need to store in the name of the unknown initial value the name

/*** The bindings **method1:1** to **method1:8** are propagated from call site **9**. ***/
/*** Recall that $a4_{init}$, $a5_{init}$, $a6_{init}$, $4_{init}.next_{init}$ and $a5_{init}.next_{init}$ are
interface initial values at the entry of the *public interface method* method2 ***/
method1:1 [*Empty-context*,$a4_{init}$,$a1_{init}$]
method1:2 [*Empty-context*,$a5_{init}$,$a2_{init}$]
method1:3 [*Empty-context*,$a6_{init}$,$a3_{init}$]
method1:4 [*Empty-context*,$a4_{init}.next_{init}$,$a1_{init}.next_{init}$]
method1:5 [*Empty-context*,$a5_{init}.next_{init}$,$a2_{init}.next_{init}$]
/*** method2::$global1_{init}$ is the interface initial value of global1 at the entry of
method2 and method1::$global1_{init}$ is the unknown initial value of global1 at
the entry of method1 ***/
method1:6 [*Empty-context*,*method2::*$global1_{init}$,*method1::*$global1_{init}$]
method1:7 [*Empty-context*,*method2::*$global2_{init}$,*method1::*$global2_{init}$]
method1:8 [*Empty-context*,*method2::*$global3_{init}$,*method1::*$global3_{init}$]

/*** The bindings **method1:9** to **method1:18** are propagated from call site **12**. ***/
method1:9 [*Empty-context*,$object_{10}$,$a1_{init}$]
method1:10 [*Empty-context*,$object_{11}$,$a2_{init}$]
method1:11 [*Empty-context*,$object_{11}$,$a3_{init}$]
method1:12 [*Empty-context*,*null*,$a1_{init}.next_{init}$]
method1:13 [*Empty-context*,*null*,$a2_{init}.next_{init}$]
method1:14 [*Empty-context*,$object_8$,*method1::*$global1_{init}$]
method1:15 [$\langle$*(*$a4_{init}$ **eq** $a5_{init}$*),empty,empty*$\rangle$,$a6_{init}$,*method1::*$global2_{init}$]
method1:16 [$\langle$*(*$a4_{init}$ **neq** $a5_{init}$*),empty,empty*$\rangle$,$a5_{init}.next_{init}$,*method1::*$global2_{init}$]
method1:17 [$\langle$*empty,(type(*$a4_{init}$*)*$\in$*A::update),empty*$\rangle$,$object_1$,*method1::*$global3_{init}$]
method1:18 [$\langle$*empty,(type(*$a4_{init}$*)*$\in$*C::update),empty*$\rangle$,$object_2$,*method1::*$global3_{init}$]

Figure 2.16: Phase II-pta solution at the entry node of method1

of the method with whose entry node the unknown initial value is associated. In our earlier examples, for simplicity we did not do this as there was no ambiguity and the method name was clear from context. However, in Figure 2.16 we need to distinguish between the unknown initial values of the same variable associated with entry nodes of two different methods. Hence, in Figure 2.16 the unknown initial values of globals have been named by prefixing the name of the method with whose entry node these unknown initial values are associated.

2.4.4 Phase III-pta

This phase involves only non-entry nodes. In this phase the unknown initial values in the parametrized points-to solution computed during Phase I-pta are **instantiated** by their concrete values computed in Phase II-pta. Those instantiations of the unknown initial values in parametrized points-to information for which the conditions associated with the points-to information evaluate to true, yield the final points-to solution. Note that a potential non-alias evaluates to false only if both operands of the potential non-alias are instantiated by the same interface initial value. Even if both operands of a potential non-alias are instantiated by the same heap-name, the potential non-alias evaluates to true because the heap-name can represent more than one run-time object. When a condition involves an interface initial value, conservative, worst-case assumptions are made about the interface initial value to evaluate the condition. 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, heap-names and interface initial values.

Figure 2.17 shows the Phase III-pta solution at a few program points in *method1*, which is defined in Figure 1.5. As discussed earlier, we show only the solution for variables read at a program point. Each element of the final points-to solution at a node n has the form:

$[rc, \langle var,\ value \rangle]$,

where *var* is a local pointer variable, a global pointer variable, a heap-name's field of pointer type or an interface initial value's field of pointer type, and *value* is a heap-name or an interface initial value. *rc* is either *Empty-context* (meaning the relevant context before the instantiation evaluated to true after the instantiation) or it is a relevant context that involves only the interface initial values of a single *public interface method* and that can evaluate to true by making conservative, worst-case assumptions about these interface initial values, meaning the points-to holds at n if *rc* holds at the entry node of the *public interface method.*

/*** We only show solution for those variables that are read at a program point. ***/

Solution at program point **3**:
[*Empty-context*, ⟨*a1*, $a4_{init}$⟩]
[*Empty-context*, ⟨*a1*, $object_{10}$⟩]

Solution at program point **7**:
[⟨*empty*, *(type(*$a5_{init}$*)∈A::update)*, *empty*⟩, ⟨*global1*, $object_1$⟩]
[⟨*empty*, *(type(*$a5_{init}$*)∈C::update)*, *empty*⟩, ⟨*global1*, $object_2$⟩]
/*** The relevant context of the following solution element is empty because when $a1_{init}$ in *(type(*$a1_{init}$*)∈A::update)* is instantiated with $object_{10}$, the type constraint evaluates to true. ***/
[*Empty-context*, ⟨*global1*, $object_1$⟩]

Figure 2.17: Phase III-pta solution at a few program points in method1

The set of points-tos computed by Phase III-pta for a program is a superset of the precise set of points-tos for the program; this is expected because computing the precise solution is undecidable [Lan92b].

The reason for retaining the relevant contexts in the final points-to solution is that these could be useful for applications like program testing (see Chapter 4), program understanding etc.

2.5 Advantages of using unknown initial values

In this section we discuss some additional advantages of bottom-up analysis using unknown initial values compared to traditional, top-down analysis using concrete values.

2.5.1 Kills or strong updates

The use of unknown initial values as compared to heap-names provides additional opportunities to perform kills or strong updates. The previous value of a field of a heap-name cannot be killed (without computing additional information) at a program point that updates that field because a heap-name may represent more than one run-time object. In contrast, it is possible to kill the previous values of the fields of unknown initial

```
void method7( ) {
   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();
}
```

Figure 2.18: Propagation using heap-names

```
void method8( 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();
}
```

Figure 2.19: Propagation using unknown initial values

values because for a particular call to a method, an unknown initial value represents the same run-time object throughout that method execution [WL95]. The following example illustrates this advantage of using unknown initial values over heap-names:

First, consider *method7*[4] defined in Figure 2.18. Class A is as defined in Figure 1.5. At program point **3**, both p and q may point to the same heap-name $object_2$; however, they may point to different run-time objects. Thus, the assignment to the *next* field of $object_2$ 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 *method7* defined in Figure 2.19.

---

[4]For clarity this example is not written in *RL*.

In *method8*, at program point **3**, both p and q may point to the same unknown initial value prm_{init}. Although for different calls to *method8*, prm_{init} may represent different run-time objects, for a given call to *method8*, prm_{init} represents the same run-time object throughout the method. Thus, the assignment to the *next* field of prm_{init} at statement **3** can be safely killed by the assignment to the same field at statement **4**.

2.5.2 Gain in efficiency due to abstraction

In programs written in object-oriented languages usually there are many invocation contexts per method (e.g., a constructor is invoked from many different contexts). As a result, there is a significant gain in efficiency by analyzing the method using unknown initial values instead of the concrete values propagated from different invocation contexts. Consider the example given in Figure 2.20.

At program point **exit-proc0**, p may point to any $object_{li}$, $i \in \{1, ..., n\}$. In many traditional, top-down techniques [LR92], n different data-flow elements are stored at program point **exit-proc0** to represent n different values of p. Each of these data-flow elements has some contextual information (e.g., reaching alias[5] in [LR92]) associated with it. When such a data-flow element reaches program point **exit-proc0**, the contextual information of the data-flow element is checked at each call site of *proc0* to find out whether the data-flow element needs to be propagated to the call site. This results in quadratic behavior for this example. In contrast, Points-to-Algo (and *RCI* in general) stores the n different values of p abstractly using the PTA-*dfelm*-**a** [*empty*,*Empty-context*,$\langle p, q_{init}\rangle$] at program point **exit-proc0**. This single PTA-*dfelm*-**a** is then passed to the return-site of each *ci*, where it is instantiated with the value of q before the call. This results in a linear behavior for this example.

[5] A reaching alias is an alias reaching the entry node of a method. In this example, reaching aliases will represent values of q propagated to the entry node of *proc0* from each call site **ci**, $i \in \{1, ..., n\}$.

```
class F {};

class test {
  public: static F *p, *q;

  public: static void proc0( ) {
    n0: p = q;

    exit-proc0:
  };

  public: static void proc1( ) {
    l1: q = new A();
    c1: proc0();
    n1:
  };

  ....

  public: static void procn( ) {
    ln: q = new A();
    cn: proc0();
    nn:
  };
}
```

Figure 2.20: Power of abstraction

2.6 Recursive Types

2.6.1 The space of initial values *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 {           class F { };
  public:  F *f1;      public:  E *f;
  public:  F *f2;    };
};

proc( D *a ) {
}
```

space(a_{init}) consists of a_{init}, $a_{init}.f_{init}$, $a_{init}.f_{init}.f1_{init}$ and $a_{init}.f_{init}.f2_{init}$.

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:              public:             };
    G *parent;           H *child;
};                   };
```

where *space(p_{init})* consists of p_{init}, $p_{init}.child_{init}$, $p_{init}.child_{init}.parent_{init}$, $p_{init}.child_{init}.parent_{init}.child_{init}$ and so on.

2.6.2 Representative names

Since Points-to-Algo potentially needs to represent any arbitrary element of a *space(unknown initial value)*, in the presence of recursive types, Points-to-Algo 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

PTA-dfelm involving a *rep* represents a set of *PTA-dfelms* containing one *PTA-dfelm* for each instantiation of the *rep* by a member of the corresponding subset.

To intuitively describe the algorithm used by Points-to-Algo to generate a *rep*, consider an unknown initial value
$p_{init}.f1_{init}.f2_{init} \ldots f_{k_{init}}$ in *space(p_{init})*. To find the corresponding *rep*, Points-to-Algo 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., p_{init}. 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 name of the above form. Points-to-Algo 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; Points-to-Algo'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 an unknown initial value $p_{init}.f1_{init}.f2_{init} \ldots f_{k_{init}}$, Points-to-Algo 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 f_i in turn, for $i = 1..k$. Points-to-Algo builds the corresponding *rep* during this tree traversal by discarding repetitive subpaths it explores. Given the selected field f_i, Points-to-Algo 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 f_i, say it is f_s. Then Points-to-Algo skips $f_{i+1init}, \ldots, f_{sinit}$ and restarts traversal from f_{s+1}. This process continues until Points-to-Algo reaches the end of the path. Now Points-to-Algo has the *rep* for the original unknown initial value. For example, consider the following example:

```
class T {          class S {             proc( S *p ) {
  public:            public:             };
    U *f2;             T *f1;
};                 };
class U {
```

```
  public:
    T *f3;
};
```

Suppose we need the *rep* for the unknown initial value $p_{init}.f1_{init}.f2_{init}.f3_{init}$. Now $f1_{init}$ and $f3_{init}$ have the same type, i.e., T. As a result, in the *rep* the subpath $f2_{init}.f3_{init}$ is discarded.

A *rep* of an unknown initial value $p_{init}.f_{1_{init}}.f_{2_{init}} \ldots f_{k_{init}}$ has the form $v_0.v_1 \ldots v_t$, where

- v_0 is p_{init} or $[p_{init}]$ and
- for each v_i, $i = 1..k$, there exist a $l \leq k$ such that v_i is $f_{l_{init}}$ or $[f_{l_{init}}]$.

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

2.6.3 Mapping between actuals and unknown initial values

Let *approx-space(uv)* be the approximation of *space(uv)* constructed by Points-to-Algo 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*, Points-to-Algo 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 an unknown initial value or a *rep*. Points-to-Algo 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)*, Points-to-Algo 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.

```
class A1 {
   public: B1 *f1;
   public: B1 *f2;
   public: A1( ) {
      f1 = f2 = 0;
   };
};

class B1 {
   public: C1 *f1;
   public: A1 *f2;
   public: B1() {
      f1 = f2 = 0;
   };
};

class C1 { };

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:
   }

   public: static void proc1( A1 *param ) {
      A1 *tmp;

      tmp = param;

      while( tmp != 0 ) {
         tmp→f2 = x;

         if ( tmp→f1 != 0 )
            tmp = tmp→f1→f2;
         else
            break;
      }

      10:
   }
}
```

Figure 2.21: Collapsing

2.6.3.1 Example

The example[6] in Figure 2.21 shows how *rep*'s are used to represent the pointer summary transfer function of a method.

The *PTA-dfelm*-**a** [*empty*,*Empty-context*,⟨*[param$_{init}$].f2*,*x_{init}*⟩] is part of the pointer summary transfer function of *proc1* and Points-to-Algo propagates it first to program point **10** and from there to program point **9**. [*param$_{init}$*] is a *rep* and the subset represented by it (i.e., **{** $param_{init}.f1_{init}.f2_{init}$, $param_{init}.f1_{init}.f2_{init}.f1_{init}.f2_{init}$, ...

[6]For clarity this example is not written in *RL*.

}) binds to { $object_4$, $object_7$ } at the call site **8**. Figures 2.22 and 2.23 show the bindings between the elements of *approx-space(*$param_{init}$*)* and the actuals at call site **8**. As a result, this *PTA-dfelm*-**a** expands to

{ [*empty*,*Empty-context*,$\langle object_4.f2, x_{init}\rangle$],

[*empty*,*Empty-context*,$\langle object_7.f2, x_{init}\rangle$] } at program point **9**.

2.7 Reachability

Points-to-Algo (and *RCI* in general) only performs analysis on *reachable* nodes in the ICFG. Points-to-Algo computes reachability information in two steps. First, during Phase I-pta, Points-to-Algo marks a statement as reachable if it is reachable from the entry node of the method containing the statement along a realizable path[7]. This is done by the function *mark-reachable* explained later in Section 2.8.2.3. The propagation during Phase I-pta is done only through nodes marked reachable by *mark-reachable*. Second, during Phase II-pta, Points-to-Algo marks the entry node of a method as reachable if it is reachable from the entry node of a *public interface method* along a realizable path. Phase II-pta is done only for those methods whose entry nodes are found to be reachable. This avoids propagation of concrete values from unreachable methods. Finally, a statement is reachable if and only if it has been marked reachable by *mark-reachable* and the entry node of the method containing the statement has been marked reachable by Phase II-pta. Intuitively, if there is a realizable path r from the entry node s of a *public interface method* to a statement l contained in a method M, then r is a concatenation of two paths *r1* and *r2*, where *r1* is a realizable path from s to the entry node of M and *r2* is a realizable path from the entry node of M to l. Reachability due to paths like *r2* is computed during Phase I-pta using *mark-reachable* and reachability due to paths like *r1* is computed during Phase II-pta.

Obviously, considering every node to be reachable will result in a safe solution. However, considering only those nodes during Phase I-pta that are marked reachable by

[7] As explained in Section 1.2, a realizable path is an interprocedural path with matching calls and returns.

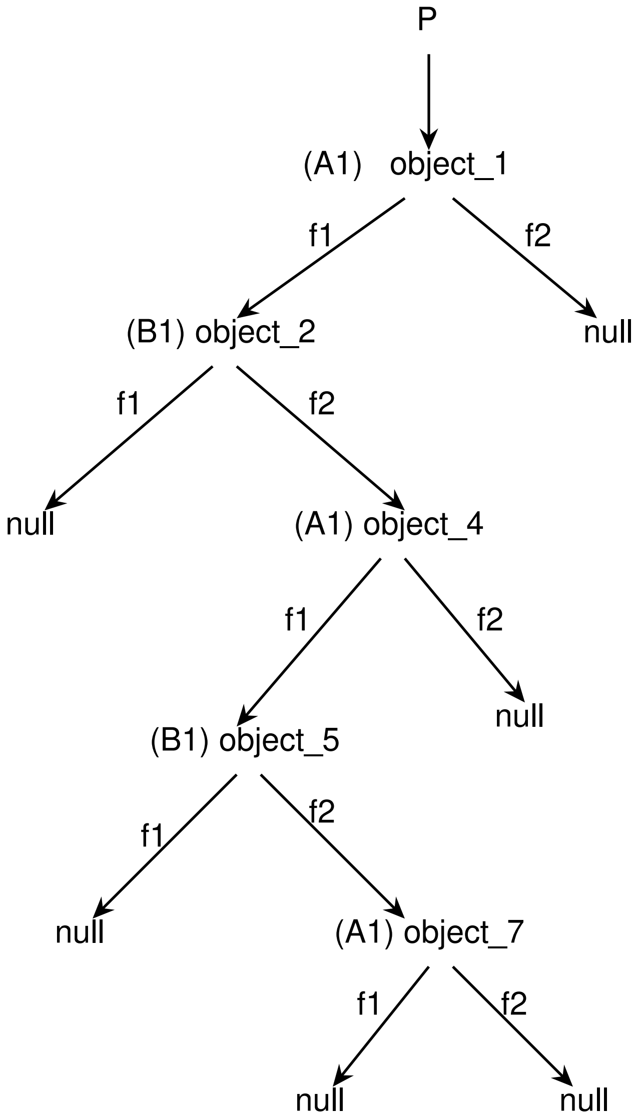

Figure 2.22: Actuals

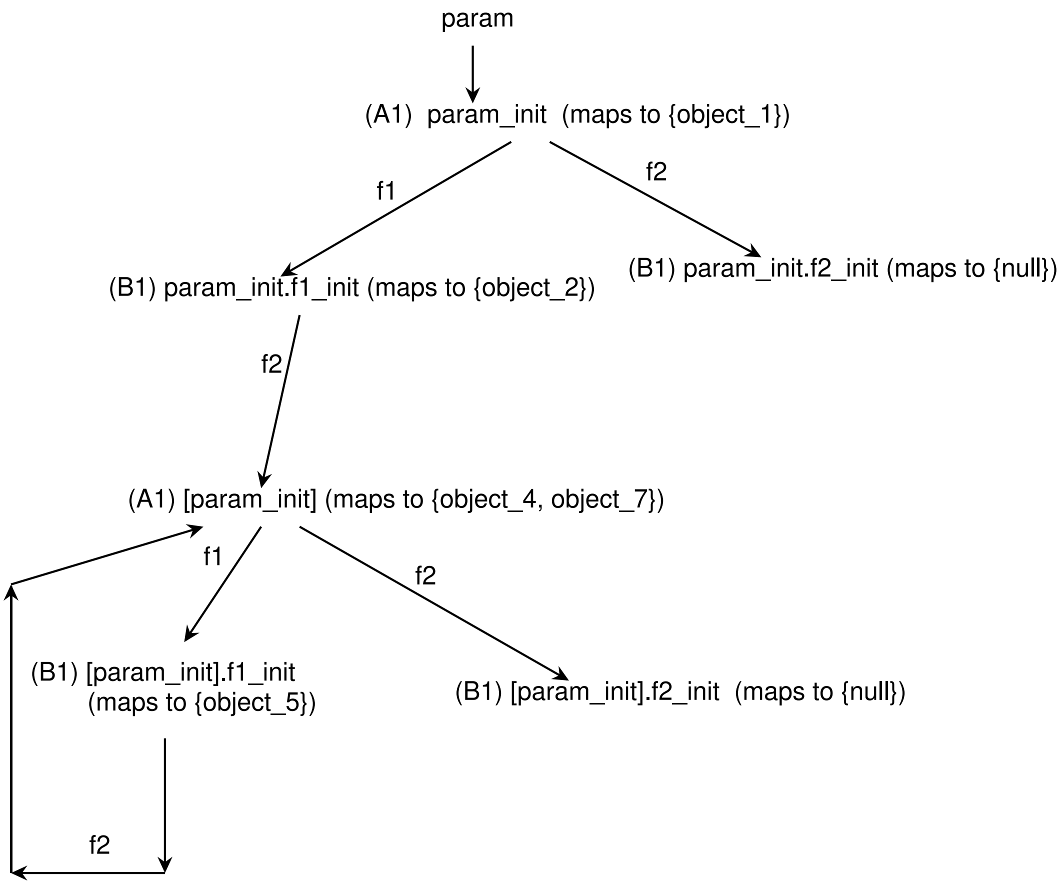

Figure 2.23: *approx-space(*$param_{init}$*)*

mark-reachable and considering only those methods during Phase II-pta that are found reachable during Phase II-pta, result in a precise solution in those cases for which points-to analysis is polynomial-time solvable (see Chapter 3).

2.8 Propagation of *PTA-dfelms* in Phase I-pta

Our goal in this section is to present a high-level view of the information flow during Phase I-pta. The pseudocode of most of the functions used during Phase I-pta is given in appendices A, B and C. The pseudocode for the rest of the functions used during Phase I-pta is given in this section. The pseudocode is illustrated using the example in Figure 1.5.

2.8.1 Terminology related to ICFG nodes

Before proceeding further, we will briefly describe some terms related to an ICFG node and we will refer to these terms in the rest of this thesis.

If an ICFG node n has a unique successor, *n.successor* represents this successor node.

For each ICFG node n, *n.method* represents the method containing n.

Each call site is represented by a pair of ICFG nodes: a call node that represents the method invocation and a return-site that is the successor of the call node. Intuitively, data-flow information is propagated from the call node to the entry node of the target method and from the exit node of the target method to the return-site. If an ICFG node n represents a call node, *n.actual-uiv-bindings* stores the bindings between actuals at n and the corresponding unknown initial values at the entry nodes of targets of n. Recall that the elements of *n.actual-uiv-bindings* are computed during Phase I-pta.

Each method has an entry node and an exit node.

A condition node represents the test expression of an *if* or *while* statement and a condition node has two successors, one of the successors represents code that is executed if the test expression evaluates to true and the other successor represents code that is executed if the test expression evaluates to false.

```
for each SCC in reverse topological order
   mark-reachable()
   initialize-worklist-I-pta()
   process-worklist-I-pta()
for each return-site n
   n.reaching-PTA-dfelms = n.reaching-PTA-dfelms ∪ n.successor.reaching-PTA-dfelms
```

Figure 2.24: Phase I-pta

If an ICFG node n represents an assignment statement, then *n.lhs* and *n.rhs* respectively represent the left-hand-side expression and the right-hand-side expression of the assignment statement represented by n.

For each ICFG node n that is not the entry node of a method, *n.reaching-PTA-dfelms* represents the set of *PTA-dfelms* that have been found (so far) to reach the top of n.

2.8.2 Overall structure of Phase I-pta

A high-level pseudocode for Phase I-pta is given in Figure 2.24. As stated earlier, Phase I-pta traverses the SCC-DAG in a reverse topological order and analyzes the procedures in a SCC using an iterative, worklist algorithm until a fixed point is reached. For example, for $L_{example}$ each SCC contains exactly one method and **{** *A::update, C::update, method1, method2, method3, method4, method5, method6* **}** is a reverse topological ordering of the SCC's. In order to avoid propagation from unreachable nodes (as stated earlier in Section 2.7), before starting the worklist algorithm on the procedures contained in the current SCC, *mark-reachable* is used to mark those ICFG nodes that are reachable along realizable paths from the entry nodes of the methods containing the ICFG nodes. Recall that as stated earlier, considering every node to be reachable during Phase I-pta will result in a safe solution. However, considering only those nodes during Phase I-pta that are marked reachable by *mark-reachable*, results in a precise solution in those cases for which points-to analysis is polynomial-time solvable (see Chapter 3).

initialize-worklist-I-pta and *process-worklist-I-pta* implement the iterative, worklist algorithm. First, *initialize-worklist-I-pta* is used for initializing the worklist with *PTA-dfelms*-**a** representing (1) the unknown initial values of globals and parameters of pointer types, at the entry node of each method in the current SCC, and (2) *initial points-tos* generated at reachable (marked by *mark-reachable*) assignment nodes, object creation sites and call nodes in the current SCC. Intuitively, a node generates an *initial points-to* if the points-to can be computed without using the points-to solution reaching the top of the node. For example, the statement $p = 0;$ generates the initial points-to $\langle p,\ null\rangle$. In contrast, the statement $p = q;$ does not generate an initial points-to because we need the values of q reaching the top of this statement to compute the values of p. *initialize-worklist-I-pta* puts the above *PTA-dfelms*-**a** on the worklist for the propagation of these *PTA-dfelms*-**a** to the successors of the ICFG nodes where these *PTA-dfelms*-**a** are generated. Next, *process-worklist-I-pta* performs iterative analysis using the worklist, until the worklist becomes empty. Each worklist entry is a pair of the form *(node, dfe)*, where *node* is an ICFG node and *dfe* is a new *PTA-dfelm* reaching *node*. Intuitively, *process-worklist-I-pta* works as follows. It repeatedly performs the following three steps until the worklist becomes empty.

1. Delete an entry $w =$ *(n,rdfe)* from the worklist.

2. Compute the effect (for points-to analysis) of the code associated with the ICFG node n on the *PTA-dfelm rdfe*.

3. Add to the worklist, new worklist entries containing (i) the *PTA-dfelms* resulting from applying the semantics of the code associated with n on *rdfe* and (ii) the successor nodes of n to which the resulting *PTA-dfelms* need to be propagated.

The steps 2 and 3 of *process-worklist-I-pta* are performed using the Phase I-pta node transfer functions of ICFG nodes. Intuitively, a Phase I-pta transfer function of an ICFG node computes the points-to-analysis effect of the code associated with the node on a new *PTA-dfelm* reaching the node (i.e., *rdfe*) and adds the resulting *PTA-dfelms* and the successors of the node to the worklist for further propagation. Thus, *process-worklist-I-pta* iteratively deletes a node w from the worklist and applies the Phase I-pta

```
                       Phase I-pta
                            |
     -------------------------------------------------
     |                    |                          |
mark-reachable    initialize-worklist-I-pta     process-worklist-I-pta
                          |                        /      / \
                          |                       /      /   \
                          |                      /      / ... \
                          |                     /     Phase I-pta
                          |                    /          node
                          |                   /     transfer functions
                          |                  /          /
                          |                 /          /
                          |                /          /
                          |               /          /
           add-to-soln-and-worklist-if-needed-I-pta
```

Figure 2.25: Overview of the call graph for Phase I-pta

node transfer function of the ICFG node contained in w on the *PTA-dfelm* contained in w, until the worklist becomes empty.

The reason why in Figure 2.24, the solution of each ICFG node that represents the return-site of a call node is augmented by the solution at the successor of the node is that during Phase I-pta, certain *PTA-dfelms* propagated across a call node are propagated to the successor of the return-site of the call node, bypassing the return-site. This is done for the ease of analysis. This is a low-level technical detail which is explained further in Section C.9.

Figure 2.25 shows an overview of the call graph for Phase I-pta. The call graphs for *initialize-worklist-I-pta* and Phase I-pta node transfer functions are only partially shown here. For the ease of presentation, we will present the complete call graphs of *initialize-worklist-I-pta* and Phase I-pta node transfer functions in the appendices B and C, which respectively contain pseudocode of *initialize-worklist-I-pta* and the Phase I-pta node transfer functions.

Table 2.1 shows the section numbers and figure numbers containing the explanation/pseudocode of the functions invoked by Phase I-pta.

Function Name	Sections Containing Explanation/Pseudocode	Figures Containing Pseudocode
mark-reachable	Sec 2.8.2.3	Figs A.1 to A.9 in Appendix A
initialize-worklist-I-pta	Appendix B	Figs B.2 to B.4 in Appendix B
process-worklist-I-pta	Sec 2.8.2.1	Figs 2.26, 2.27
add-to-soln-and-worklist-if-needed-I-pta	Sec 2.8.2.2	Fig 2.27
Phase I-pta node transfer functions	Appendix C	Figs C.2 to C.23

Table 2.1: Guide to functions invoked by Phase I-pta

2.8.2.1 process-worklist-I-pta

A high-level pseudocode for *process-worklist-I-pta* is given in Figure 2.26. As stated above, *process-worklist-I-pta* essentially invokes Phase I-pta transfer functions for different kinds of ICFG nodes.

rdfe is either (1) (*first case*) a *PTA-dfelm*-**a** [*ecfi*,*rc1*,⟨*var*,*val*⟩] or (2) (*second case*) a *PTA-dfelm*-**b** [*rc1*,*excp-obj*].

The Phase I-pta node transfer functions are defined in Figures C.2 to C.23 contained in Appendix C.

2.8.2.2 add-to-soln-and-worklist-if-needed-I-pta

The function *add-to-soln-and-worklist-if-needed-I-pta* (Figure 2.27) is used for adding a ICFG node and new *PTA-dfelms* reaching that ICFG node to the worklist for future processing. Figure 2.27 presents pseudocode for this function.

Here *node* is an ICFG node, *dfelms* is a set of *PTA-dfelms* and *dfe* is a *PTA-dfelm*.

```
process-worklist-I-pta() {
while worklist is not empty {
  wl-node = delete node from worklist
  n = wl-node.node; rdfe = wl-node.dfe

  if (n represents an assignment statement)
    1: process-assignment-pta(n, rdfe)

// A condition node represents the test expression of an if or while
  if (n is a condition node)
    2: for each successor succ of n
          add-to-soln-and-worklist-if-needed-I-pta(succ,{rdfe})

  if (n is a throw node)
    3: process-throw-pta( n, rdfe )
  if (n is the entry node of a try)
    4: process-try-entry-pta( n, rdfe )
  if (n is the exit node of a try block)
    5: process-try-exit-pta( n, rdfe )
  if (n is the entry node of a catch)
    6: process-catch-entry-pta( n, rdfe )
  if (n is the exit node of a catch)
    7: process-catch-exit-pta( n, rdfe )
  if (n is the entry node of a finally) // n.successor is the successor of n
    8: add-to-soln-and-worklist-if-needed-I-pta(n.successor,{rdfe})
  if (n is the exit node of a finally)
    9: process-finally-exit-pta( n, rdfe )
  if (n is a call node)
    10: process-call-pta( n, rdfe )
  if (n is the exit node of a method)
    11: process-method-exit-pta( n, rdfe )
  if (n represents a break statement)
    12: process-break-pta( n, rdfe )
  if (n represents a continue statement)
    13: process-continue-pta( n, rdfe )
  if (n represents a return statement)
    14: process-return-statement-pta( n, rdfe )
  if (n is the return-site of a call node) // i.e., n is the successor of a call node
    15: process-return-site-pta( n, rdfe )
// a new-node is an object creation site
  if (n is a new-node)
    16: process-new-node-pta( n, rdfe )
} }
```

Figure 2.26: process worklist for Phase I-pta

add-to-soln-and-worklist-if-needed-I-pta(node, dfelms) {
for each $dfe \in dfelms$
if $dfe \notin node.reaching\text{-}PTA\text{-}dfelms$
$node.reaching\text{-}PTA\text{-}dfelms = node.reaching\text{-}PTA\text{-}dfelms \cup \{\text{dfe}\}$
wl-node = new worklist node (*node, dfe*)
add *wl-node* to *worklist* **}**

Figure 2.27: Pseudocode for adding information to the worklist during Phase I-pta

2.8.2.3 mark-reachable

mark-reachable marks reachable ICFG nodes in the current SCC using a simple iterative worklist algorithm. It assumes the entry node of each method in the current SCC is reachable. The pseudocode of *mark-reachable* is given in Appendix A.

For the example in Figure 1.5, all nodes except those contained in the catch statement at program point **36** are marked reachable by *mark-reachable*. The catch statement at program point **36** is not reachable because it catches exceptions of type *ET5* or a subtype of *ET5* and the call to *method5* at program point **35** throws exception of type *ET1*.

Each data-flow element computed by *mark-reachable* at a program point n contained in a method M has one of the following three forms:

1. ⟨*empty,reachable*⟩, meaning n is reachable from the entry node of M along a path that does not have any uncaught exception, or any pending transition due to a break statement, a continue statement, a return statement or falling through a try statement or catch statement associated with a finally statement.

2. ⟨*excp-type,reachable*⟩, meaning n is reachable from the entry node of M along a path that has an uncaught exception of type *excp-type* or a subtype of *excp-type*.

3. ⟨*label,reachable*⟩, meaning n is reachable from the entry node of M along a path that has a pending transition (due to a break statement, a continue statement, a return statement or falling through a try statement or catch statement associated with a finally statement) to statement number *label*.

for each SCC in topological order
  initialize-worklist-II-pta()
  process-worklist-II-pta()

Figure 2.28: Phase II-pta

For every node (except (1) an exit node of a try statement, a catch statement, a finally statement or a method and (2) the return site of a call node) that is not directly contained in a finally statement, all data-flow elements computed by *mark-reachable* are of first kind. For every node that is directly contained in a finally statement, all data-flow elements computed by *mark-reachable* are of the second or third kind; this is because there is always a reason for entering a finally statement and the first component of a data-flow element computed by *mark-reachable* stores this reason.

2.9 Phase II-pta

Phase II-pta is defined in Figures 2.28, 2.29, 2.30 and 2.31. Following is an informal overview of Phase II-pta.

For each concrete value of an unknown initial value, computed at the entry node of a method M during this phase, Points-to-Algo visits each of the reachable (marked by *mark-reachable*) call sites in M and does the following:

- For each reachable dynamically dispatched call site C in M, Points-to-Algo 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 PTA-*dfelms*-**a** computed during Phase I-pta that represent the values of p at C. Points-to-Algo evaluates the PTA-*dfelms*-**a** of S by instantiating unknown initial values with their concrete values computed at the entry node of M. Those instantiations of the PTA-*dfelms*-**a** for which the relevant contexts evaluate to *true* or those instantiations of the PTA-*dfelms*-**a** for which the relevant contexts are instantiated into relevant contexts that involve interface initial values of a single *public interface method* and that can evaluate to true by making conservative, worst-case assumptions about

initialize-worklist-II-pta() {
worklist = ϕ

for each reachable method m in the current SCC
   initialize-worklist-II-pta-for-method(m)
}

---

initialize-worklist-II-pta-for-method(m) }
for each el $\in$ m.cv-to-uiv-bindings
   wl-node = new worklist node (m, el)
   add wl-node to worklist

for each reachable call node c in m {
   if (c is statically dispatched)
      mark-if-new-reachable-target(target of c)
   for each actual-to-uiv binding [*ecfi1*,*rc1*,$\langle$*actual1*,*uiv1*$\rangle$] $\in$ c.actual-uiv-bindings {
      if ((*actual1* is a heap-name or an interface initial value)
            and (*rc1* is *Empty-context* or can-evaluate-to-true(m, *rc1*)))
         // *uiv1.method* is the target (of c) with
         // whose entry node *uiv1* is associated
         mark-if-new-reachable-target(*uiv1.method*)
         add-to-soln-and-worklist-if-needed-II-pta(uiv1.method, {[*rc1*,*actual1*,*uiv1*]})
   }
}**}**

Figure 2.29: worklist initialization for Phase II-pta

process-worklist-II-pta() {

```
while worklist is not empty {

  wl-node = delete from work-list
  m = wl-node.method
  cv-uiv-binding = wl-node.binding
  let cv-uiv-binding be [rc1,cv1,uiv1]

  for each reachable call node c in m
    mark-new-reachable-targets( c, cv-uiv-binding )
    for each actual-to-uiv binding s ∈ c.actual-to-uiv-bindings such that s
          contains uiv1
      /*** let s be [ecfi2,rc2,⟨actual2,uiv2⟩] ***/
      result = instantiate-actual-to-uiv-binding(m, s, cv-uiv-binding)
      /*** here uiv2.method is the target with whose entry node uiv2 is associated
      ***/
      add-to-soln-and-worklist-if-needed-II-pta(uiv2.method, result)
} }
```

---

instantiate-actual-to-uiv-binding(method, act-to-uiv-binding, cv-to-uiv-binding){

```
result = φ

let cv-to-uiv-binding be [rc1,cv1,uiv1]
let act-to-uiv-binding be [ecfi2,rc2,⟨actual2,uiv2⟩]

/*** either rc2 contains uiv1 or actual2 is uiv1 ***/

for (each instantiation of UIV's in ⟨rc2,actual2⟩ with their concrete values computed
      so far at method.entry such that only the cv-to-uiv-binding [rc1,cv1,uiv1] is
      used for instantiating uiv1) {
  /*** only cv1 should be used as value of uiv1 ***/
  let rc2_inst be the instantiated form of rc2
  if (can-be-true(rc2_inst))
    let actual3 be the value of actual2 after the instantiation
    rc3 = drop-true-conjuncts(rc2_inst)
    result = result ∪ {[rc3,actual3,uiv2]}
}
return result }
```

Figure 2.30: worklist processing for Phase II-pta

can-evaluate-to-true(method, rc){
if (rc involves only interface initial values of a single *public interface method* and
        can-be-true(rc))
  return true
else
  return false }

can-be-true(rc_{inst}**)** {
// rc_{inst} is a relevant context or a relevant context in which unknown initial values
// have been instantiated by their concrete values
if (rc_{inst} can be true by making conservative, worst-case assumptions
      about the interface initial values in rc_{inst})
  return true
else
  return false }

add-to-soln-and-worklist-if-needed-II-pta(method, bindings) {
for each el ∈ bindings
  if el ∉ method.cv-to-uiv-bindings
    method.cv-to-uiv-bindings = method.cv-to-uiv-bindings ∪ {el}
    if (method is in the current SCC)
      wl-node = new worklist node (method, el)
      add wl-node to worklist
}

Figure 2.31: Phase II-pta auxiliary functions

these interface initial values, 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-pta, Points-to-Algo produces the **final call graph** which is a significant refinement of the initial call graph. At a dynamically dispatched call site in the final call graph, the only targets considered invocable are those that are invocable using the concrete values of the receiver computed at the call site during Phase II-pta.

- At each call site C, Points-to-Algo uses the actual-to-uiv bindings computed during Phase I-pta to propagate concrete values to the methods invocable from C. Points-to-Algo evaluates an actual-to-uiv binding by substituting the unknown initial values in the binding with their concrete values computed at the entry node of M. An instantiation of a binding is used for propagation if and only if either (1) the relevant context of the binding evaluates to true after the instantiation or (2) the relevant context of the binding is instantiated into an relevant context that involves the interface initial values of a single *public interface method* and that can evaluate to true by making conservative, worst-case assumptions about these interface initial values.

In order to avoid propagation of concrete values from unreachable methods, Points-to-Algo computes an initial set of *reachable* methods, and then incrementally expands this set during Phase II-pta. The initial set of reachable methods consists of all the *public interface methods*[8]. When Points-to-Algo visits each node of SCC-DAG in a topological order during Phase II-pta, it considers only those methods in the current SCC that have been marked reachable. Whenever Points-to-Algo finds an unmarked method to be invocable from a call site in a reachable method, it uses *mark-if-new-reachable-target* and *mark-new-reachable-targets* to mark the new method also as reachable. If the newly marked method m is contained in the current SCC, *mark-if-new-reachable-target* and *mark-new-reachable-targets* call *initialize-worklist-II-pta-for-method(m)* to perform worklist initialization for m.

[8]For complete programs the initial set consists of only *main*.

For example, consider Phase II-pta for the example in Figure 1.5. *method2* is a *public interface method* and hence it is contained in the initial set of reachable methods. As a result, *method1* is also reachable as it is invoked from a reachable call site in *method2*. At call site 12, Phase I-pta stores an actual-to-uiv binding [*empty*,*Empty-context*,$\langle object_{10}, a1_{init}\rangle$]. Since the relevant context *Empty-context* trivially evaluates to *true*, *initialize-worklist-II-pta-for-method(method2)* propagates $object_{10}$ as a concrete value of $a1_{init}$ to the entry node of *method1*. At the call site **5**, the value of the receiver is the value of *r*, and the value of *r* is given by the *PTA-dfelm*-**a** [*empty*,*Empty-context*,$\langle r, a1_{init}\rangle$]. *mark-new-reachable-targets(5,*[*empty*,$object_{10}$,$a1_{init}$]*)* substitutes $object_{10}$ for $a1_{init}$ in this *PTA-dfelm*-**a** to obtain $object_{10}$ as a concrete value of the receiver. This implies that *A::update* is invocable from statement **5** and the final call graph has an edge from statement **5** to *A::update*.

2.10 Phase III-pta

In this phase unknown initial values in *PTA-dfelms*-**a** are instantiated by their concrete values computed in Phase II-pta. Recall that an interface initial value can be the concrete value of an unknown initial value. Those instantiations of *PTA-dfelms*-**a** for which relevant contexts either (1) evaluate to true or (2) they are instantiated into relevant contexts involving only the interface initial values of a single *public interface method* and the instantiated relevant contexts can evaluate to true by making conservative, worst-case assumptions about these interface initial values, yield the final points-to solution. Recall each element of the final points-to solution at a node *n* has the form:

- [*rc*, $\langle$*var*, *value*$\rangle$],

where *var* is a local pointer variable, a global pointer variable, a heap-name's field of pointer type or a interface initial value's field of pointer type, and *value* is a heap-name or an interface initial value. *rc* is either *Empty-context* (meaning the relevant context before the instantiation evaluated to true after the instantiation) or it is a relevant context involving only the interface initial values of a single *public interface method* that can evaluate to true by making conservative, worst-case assumptions about these interface initial values.

For example, suppose Phase III-pta needs to be done at program point **6** in Figure 1.5. For this, the unknown initial values in each PTA-*dfelm*-**a** at program point **6** are instantiated by their concrete values computed at the entry node of *method1*. For example, $[$*empty*,$\langle$*empty,(type(*$a1_{init}$*)*$\in$*A::update),empty*$\rangle$,$\langle$*global1,object*$_1\rangle] \in$ *6.reaching-PTA-dfelms* and $\langle$*Empty-context*, $a4_{init}$, $a1_{init}$ $\rangle\in$ *method1.cv-to-uiv-bindings*. As a result, $a1_{init}$ is instantiated by the interface initial value $a4_{init}$ and $[$*empty*,$\langle$*empty,(type(*$a4_{init}$*)*$\in$*A::update),empty*$\rangle$,$\langle$*global1,object*$_1\rangle]$ is generated, which belongs to the final points-to solution at program point **6**.

Chapter 3

Complexity Characterizations

In this chapter we characterize the complexity of points-to analysis in the presence of object-oriented language constructs: exceptions and dynamic dispatch. Our results [CRL98a, CRL97] clearly identify the difficult features and indicate the approximations any efficient algorithm has to make. Our results are summarized in Table 3.1.

The hierarchy of complexity classes is shown in Figure 3.1. It is known that **NC** is a proper subset of **P-Space**. Apart from this, it is not known if any of the containments shown in Figure 3.1 is proper.

3.1 Definitions

This section presents some definitions needed to explain the results in this chapter.

Single-level type: A single-level type is one of the following:

1. a primitive type defined in [GJS96] (e.g., *int*, *float* etc.) or
2. a class that has all non-static data-members of primitive types (e.g., *class A { int i,j; }*).

Subtype: We use the same definition of subtyping as in C++ and Java: a class A is a subtype of another class B if and only if A extends B, either directly or indirectly through inheritance.

exception type: An *exception type* is *Exception* or a subtype of *Exception*.

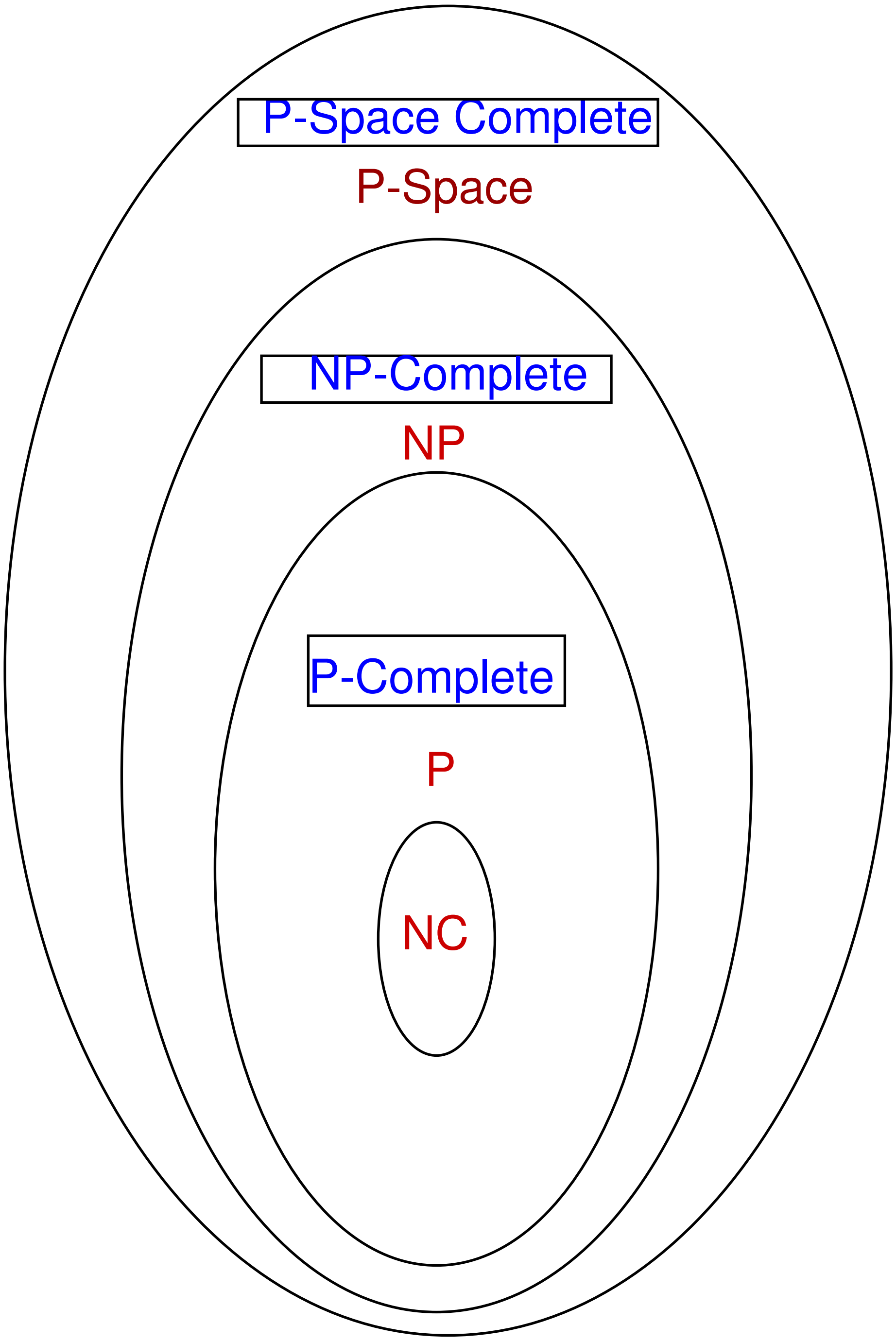

Figure 3.1: Hierarchy of complexity classes

results	chapter section	single-level types	exceptions without subtypes	exceptions with subtypes	dynamic dispatch
interprocedural points-to analysis in **P**, $O(n^7)$	sec 3.3	x	x		
intraprocedural points-to analysis **PSPACE**-complete	sec 3.2	x		x	
interprocedural points-to analysis **PSPACE**-hard	sec 3.2	x		x	
interprocedural points-to analysis **PSPACE**-hard	sec 3.2	x			x
interprocedural points-to analysis in **P**, $O(n^4)$	sec 3.3.2	x			
intraprocedural points-to analysis in **NC**	sec 3.3.3	x			

Table 3.1: Complexity results for points-to analysis summarized

Direct containment in a finally: A program point l is said to be directly contained in a finally if and only if *innermost-enclosing-exception-block(l)* is a finally statement.

Containment in a nested try-catch-finally: A program point l is said to be contained in a nested try-catch-finally if and only if (1) l is contained in the body b of a try, catch or finally statement and (2) *innermost-enclosing-exception-block(b)* is a finally statement.

start-node: A *start-node* is the entry node of a *public interface method*.

realizable path: A realizable path [SP81, LR91, RHS95] is an interprocedural path in the ICFG in which calls and returns are properly matched.

realizable-path-from-a-start-node(r,s,l): Let l be a program point and s be a start-node. realizable-path-from-a-start-node(r,s,l) denotes a realizable path r from the start-node s to l.

realizable-path-from-entry-node(r,M,l): Let l be a program point in a method M. realizable-path-from-entry-node(r,M,l) denotes a realizable path r from the entry-node of M to l. Here M can be an anonymous procedure that represents a *try-catch-finally* directly contained in a finally statement. If M represents an anonymous procedure, then it is the innermost such procedure containing l.

realizable-path-from-entry-node-without-exception-and-label(r,M,l): Let l be a program point in a method M. realizable-path-from-entry-node-without-exception-and-label(*r,M,l*) denotes a realizable-path-from-entry-node(*r,M,l*) such that when r is followed, at l, (1) there is no uncaught exception object that is thrown[1] at a program point on r and (2) there is no incomplete, pending transition that is generated in r and that is due to a break statement, a continue statement, a return statement or falling through a try statement or a catch statement associated with a finally statement.

realizable-path-from-entry-node-with-exception(r,M,l,excp-type): Let l be a program point in a method M. realizable-path-from-entry-node-with-exception(*r,M,l,excp-type*) denotes a realizable-path-from-entry-node(*r,M,l*) such that when r is followed, at l, there is an uncaught exception object (1) that is thrown at a program point on r, (2) whose type is *excp-type* and (3) that is allocated at a program point on r.

realizable-path-from-entry-node-with-exception(r,M,l,uiv): Let l be a program point in a method M and *uiv* be an unknown initial value. realizable-path-from-entry-node-with-exception(*r,M,l,uiv*) denotes a realizable-path-from-entry-node(*r,M,l*) such that when r is followed, at l, there is an uncaught exception object *uiv* that is thrown at a program point on r.

realizable-path-from-entry-node-with-label(r,M,l,label): Let l be a program point in a method M. realizable-path-from-entry-node-with-label(*r,M,l,label*) denotes a realizable-path-from-entry-node(*r,M,l*) such that when r is followed, at l, there is an incomplete, pending transition to statement numbered *label* that is generated in r and that is due to a break statement, a continue statement, a return statement or falling through a try statement or a catch statement associated with a finally statement.

[1]Due to a call or try statement nested inside a finally statement, there may be active, uncaught exceptions at the entry node of M.

3.2 PSPACE-hardness of points-to analysis in the presence of exceptions and dynamic dispatch

In this section we show that intraprocedural points-to analysis for programs written in a subset of *RL* that has only single-level types and exceptions with subtyping is **PSPACE**-complete, while the interprocedural case is **PSPACE**-hard, even without dynamic dispatch. We also show that points-to analysis for programs written in a subset of *RL* that has only single-level types and dynamic dispatch (and does not have exceptions) is **PSPACE**-hard.

3.2.1 PSPACE-hardness of points-to analysis in the presence of exceptions

In [Lan92a], Landi showed that intraprocedural may-alias analysis for a subset of C that has only pointers with at most 4 levels of indirection[2] and that does not have dynamic allocation is **PSPACE**-complete. The only control-flow statements used in the proof are *if* and *while*. The same proof shows that intraprocedural points-to analysis for the same subset of C (PTA for short)[3] is **PSPACE**-complete.

We will reduce PTA in polynomial time to the problem of intraprocedural points-to analysis for programs written in a subset of *RL* that has only single-level types (here we allow dynamic allocation because we are considering *RL* programs and in *RL*, the only way to initialize a pointer variable is through dynamic allocation) and exceptions with subtyping (TIA for short). For ease of presentation we will show the reduction for an instance of PTA with at most 2 levels of indirection. The reduction of an instance of PTA with higher levels of indirection is a straightforward extension of the reduction for the case with at most 2 levels of indirection. The key idea in the reduction is to model the address of a variable using subtyping and to model pointer dereference by throwing an exception which is caught by a catch statement with appropriate declared

---

[2] e.g., int $****p,\ ***q$;

[3] Given a program point l in a procedure *Proc*, and variables *v1* and *v2*, does there exist a path from the entry-node of *Proc* to l such that *v1* points to *v2* at l when this path is followed.

type of the parameter.

Consider an instance of PTA:

```
void proc1( )
{
  int r1, ..., rm;       // m variables
  int *q1, ..., *qn;     // n variables
  int **p1, ..., **ps;   // s variables

  ...
   // pointer assignments, ifs and whiles

}
```

The corresponding instance of TIA in *RL* is as follows:

```
class int_type {
};
class multi_level_pointer :public Exception {
   // We assume that Exception is a single-level type
};
class q1_type :  public multi_level_pointer {
};
...
class qn_type :  public multi_level_pointer {
};



class TIA {
  static void proc2( )
```

```
  {
    int_type *r1, ..., *rm;
    int_type *q1_var, ..., *qn_var;
    q1_type *q1_addr;
    ...
    qn_type *qn_addr;

    multi_level_pointer *p1, ..., *ps;

    l1:  r1 = new int_type();  // corresponds to r1 in PTA
    ...
    lm:  rm = new int_type();  // corresponds to rm in PTA

    t1:  q1_addr = new q1_type();
     // corresponds to address of q1 in PTA
    ...
    tn:  qn_addr = new qn_type();
     // corresponds to address of qn in PTA

    ...
  }
}
```

The statements in PTA are translated to equivalent statements in TIA as follows:

A statement of the form fk: qi = &rj is translated to:

```
    fk:  qi_var = rj;
```

A statement of the form fk: qi = qj is translated to:

```
    fk:  qi_var = qj_var;
```

A statement of the form fk: pi = &qj is translated to:

```
fk:  pi = qj_addr;
```

*A statement of the form fk: *pi = qj is translated to:*

*// a statement of the form fk: *pi = &rj is translated similarly*

```
fk:  try {
  throw pi;
}
catch( q1_type *excp ) {
  q1_var = qj_var;
}
...
catch( qn_type *excp ) {
  qn_var = qj_var;
}
```

*Similarly, a statement of the form fk: qi = *pj is translated to:*

```
fk:  try {
  throw pj;
}
catch( q1_type *excp ) {
  qi_var = q1_var;
}
...
catch( qn_type *excp ) {
  qi_var = qn_var;
}
```

Combining the above two cases a statement of the form `fk: *pi = *pj` *is translated to:*

```
fk: try {
  throw pi;
}
catch( q1_type *excp ) {
  try {
    throw pj;
  }
  catch( q1_type *excp ) {
    q1_var = q1_var;
  }
  ...
  catch( qn_type *excp ) {
    q1_var = qn_var;
  }
}
...
catch( qn_type *excp ) {
  try {
    throw pj;
  }
  catch( q1_type *excp ) {
    qn_var = q1_var;
  }
  ...
  catch( qn_type *excp ) {
    qn_var = qn_var;
  }
}
```

ifs and *whiles* are translated verbatim, except the test expressions, which we assume (without loss of generality) to be side-effect free and hence ignore in our analysis. Other statements that do not modify any pointer variable are translated into empty statements. This does not preserve program semantics, but it does not affect the mapping between the the points-to solutions of the two problems.

The following lemma is immediate from the above construction:

Lemma 1 *Let f_k be a program point in proc1. Then:*

1. *$\langle q_i, r_j \rangle$ holds at f_k if and only if $\langle q_i\_var, object_{lj} \rangle$ holds at f_k in proc2.*

2. *$\langle p_i, q_j \rangle$ holds at f_k if and only if $\langle p_i, object_{tj} \rangle$ holds at f_k in proc2.*

If the instance of PTA has $O(n)$ statements (and hence at most $O(n)$ variables as we need to consider only those variables that are used in the procedure) then the corresponding instance of TIA, constructed above, has $O(n^3)$ statements, as each statement in PTA is replaced by at most $O(n^2)$ statements. As a result, we have a polynomial-time reduction from PTA to TIA. Note that if we start with an instance of PTA with pointers having at most 4 levels of indirection, then using the scheme described above we will get an equivalent instance of TIA of size at most $O(n^7)$. In general, if we start with at most k levels of indirection, we will get an equivalent instance of TIA of size at most $O(n^{2(k-1)+1})$. As a result, the above construction shows that intraprocedural TIA is **PSPACE**-hard.

Now we will show that intraprocedural TIA is in **PSPACE**, and hence **PSPACE**-complete. Savitch's theorem [HU79] implies that **PSPACE**= **NPSPACE**[4]. Hence, it is enough to show that intraprocedural TIA is in **NPSPACE**. Our proof is similar to Landi's proof [Lan92a] that PTA is in **NPSPACE**. We present a non-deterministic

[4]languages accepted by non-deterministic Turing machines using polynomial amount of space [Pap94].

algorithm for TIA, which uses a polynomial amount of space. The input to the algorithm is the CFG of a method M, a node l in the CFG and a points-to $\langle p,\ obj\ \rangle$. It outputs *yes* if and only if this points-to holds at the top of l (with respect to some path from the entry-node of M to l), otherwise it outputs *no*. The following is a brief outline of the algorithm.

The algorithm maintains a set S of points-tos computed so far and a stack St of *labels* and exception objects representing reasons for entering nested finally statements. It uses two special variables: *excp-var* and *label-var*, for storing the reason for exit from the last try or catch statement. *excp-var* points to the last uncaught exception object if the exit was due to an exception, while *label-var* stores the number of a target statement if the exit was due to a labelled *break* or *continue*, *return* or by falling through. Note that at any instant at most one of these variables has a valid value. At each node the algorithm computes a new S by considering the effect of the node on the old S, and then non-deterministically (except for a throw statement or the exit-node of a try statement, a catch statement or a finally statement) chooses one of the successors of the node and continues with the new S from this successor. Whenever the algorithm reaches l, it checks whether $\langle p,\ obj\ \rangle$ is present in S. If yes, it accepts the input and stops; otherwise it continues as above. It also keeps track of the length of the path traversed so far. If the path length exceeds nn^n2^m (n is the number of statements and m is the maximum number of possible points-tos), it rejects the input and stops. Since we are considering only single-level types, both the number of variables (including *excp-var* and *label-var*) and the number of objects we need to consider are bounded by $n+2$. Hence m is at most $(n+2)^2$. Moreover, the height of St is bounded by n and each element of St can have at most n different values. As a result, the maximum length of a shortest path associated with a points-to that does not occur on any shorter path is at most $nn^n2^{(n+2)^2}$. This justifies not considering paths longer than this.

When a *try-catch-finally* directly contained in a finally statement is entered, the value of *excp-var* or *label-var*, depending upon which has a valid value, is pushed on to St and these variables are reinitialized to have null values.

Now we will describe how the algorithm chooses a successor at a (I) throw statement, (II) the exit node of a try statement, (III) the exit node of a catch statement, (IV) a break statement, (V) a continue statement, (VI) a return statement and (VII) the exit node of a finally statement. At the exit node of each try statement, the algorithm maintains an array (*catch-table*) indexed by exception types, which stores for each exception type a pointer to (1) the entry of an associated catch statement, (2) the entry node of the associated finally statement (if any) or (3) the exit node of *innermost-enclosing-exception-block(try)* where control should go if the exit node of the try statement is reached with an exception object of this type. These tables can be easily built in polynomial time by making a prepass through the method.

(I) At a *throw* statement *l*, the algorithm instantiates *excp-var* with the thrown exception object and chooses exit node of *innermost-enclosing-exception-block(l)* as the successor.

(II) At the exit node *l* of a try statement *t*, the algorithm chooses the successor as follows. If *excp-var* is not null, then the algorithm looks up the *catch-table* using the type of the exception object (to which *excp-var* is pointing) to determine the successor. In the rest of this paragraph we will deal with different cases that can arise depending upon (a) whether the exit is due to an exception or due to an incomplete, pending transition or due to falling through, (b) whether there is a finally statement associated with *t* or not and (c) whether *innermost-enclosing-exception-block(t)* is a finally statement or not. If *excp-var* is not null and the successor is the exit node of *innermost-enclosing-exception-block(t)* and *innermost-enclosing-exception-block(t)* is a finally statement, then the algorithm pops the top of the stack *St* (as the current exception overrides the previous reason on the top of *St*). At this point in the argument, a procedure is introduced that is identical to what is needed later on, when we describe how to process the exit node of a catch statement. The phrases *Begin common* and *End common* delimit the start and end of this procedure. *Begin common:*

If *label-var* is not null and there is a finally statement associated with *t*, then the the algorithm chooses the entry node of the finally statement as the successor. If *label-var* is not null and there is no finally statement associated with *t* and the statement whose number is stored in *label-var* is contained in *innermost-enclosing-exception-block(t)* and *innermost-enclosing-exception-block(t)* is not a finally statement, then the the algorithm chooses the statement whose number is stored in *label-var* as the successor and reinitializes *label-var* to null. If *label-var* is not null and there is no finally statement associated with t and the statement whose number is stored in *label-var* is contained in *innermost-enclosing-exception-block(t)* and *innermost-enclosing-exception-block(t)* is a finally statement, then the the algorithm chooses the statement whose number is stored in *label-var* as the successor, reinitializes *label-var* to null, pops the top of the stack St and assigns this value to *label-var* or *excp-var* depending upon whether the top of the stack was a label or an exception. If *label-var* is not null and there is no finally statement associated with t and the statement whose number is stored in *label-var* is not contained in *innermost-enclosing-exception-block(t)* and *innermost-enclosing-exception-block(t)* is not a finally statement, then the the algorithm chooses the exit node of *innermost-enclosing-exception-block(t)* as the successor. If *label-var* is not null and there is no finally statement associated with t and the statement whose number is stored in *label-var* is not contained in *innermost-enclosing-exception-block(t)* and *innermost-enclosing-exception-block(t)* is a finally statement, then the algorithm chooses the exit node of *innermost-enclosing-exception-block(t)* as the successor and pops the top of the stack St (as the current reason for exiting the finally statement overrides the previous reason on the top of St). If both *label-var* and *excp-var* are null and there is a finally statement associated with t, the algorithm stores the number of the successor of the try-catch-finally construct associated with t in *label-var* and chooses the entry node of the finally statement as the successor. If both *label-var* and *excp-var* are null and there is no finally statement associated with t and *innermost-enclosing-exception-block(t)* is not a finally statement, the algorithm chooses the successor of the try-catch-finally construct associated with t as the successor. If both *label-var* and *excp-var* are null and there is no finally statement associated with t and *innermost-enclosing-exception-block(t)* is a finally statement, the algorithm chooses the successor of the try-catch-finally construct associated with t as the successor, pops the top of the stack St and assigns this value to *label-var* or *excp-var* depending upon whether the top of the stack was a label or an exception.

End common

(III) At the exit node l of a catch statement c, the algorithm chooses the successor as follows. If *excp-var* is not null and there is a finally statement associated with c, then the algorithm chooses the entry node of the finally statement as the successor. If *excp-var* is not null and there is no finally statement associated with c and *innermost-enclosing-exception-block(c)* is not a finally statement, then the algorithm chooses the exit node of *innermost-enclosing-exception-block(c)* as the successor. If *excp-var* is not null and there is no finally statement associated with c and *innermost-enclosing-exception-block(c)* is a finally statement, then the algorithm chooses the exit node of *innermost-enclosing-exception-block(c)* as the successor and pops the top of the stack *St* (as the current exception overrides the previous reason on the top of *St*). The rest of this paragraph is same as the last portion of the above paragraph marked by *Begin common* and *End common*, with the try statement t replaced by the catch statement c.

(IV), (V) At a break statement or a continue statement l, the algorithm chooses the successor as follows. Let t be the target of the break statement or the continue statement. If t is contained in *innermost-enclosing-exception-block(l)* , then the algorithm chooses t as the successor and the values of *excp-var* and *label-var* remain unchanged. If t is not contained in *innermost-enclosing-exception-block(l)* , then the algorithm chooses the exit node of *innermost-enclosing-exception-block(l)* as the successor, stores t in *label-var* and reinitializes *excp-var* with null.

(VI) At a return statement l, the algorithm chooses the successor as follows. Let t be the number of the exit node of the method. If *innermost-enclosing-exception-block(l)* is the method body, then the algorithm chooses t as the successor. If *innermost-enclosing-exception-block(l)* is not the method body, then the algorithm chooses the exit node of *innermost-enclosing-exception-block(l)* as the successor, assigns t to *label-var* and reinitializes *excp-var* with null.

(VII) At the exit node l of a finally statement f, the algorithm chooses the successor as follows. If *excp-var* is not null and *innermost-enclosing-exception-block(f)* is not a finally statement, then the algorithm chooses the exit node of *innermost-enclosing-exception-block(f)* as the successor. If *excp-var* is not null and *innermost-enclosing-exception-block(f)* is a finally statement, then the algorithm chooses the exit node of *innermost-enclosing-exception-block(f)* as the successor and pops the top of the stack *St* (as the current exception overrides the previous reason on the top of *St*). If *label-var* is not null and the statement whose number is stored in *label-var* is contained in *innermost-enclosing-exception-block(f)* and *innermost-enclosing-exception-block(f)* is not a finally statement, then the algorithm chooses the statement whose number is stored in *label-var* as the successor and reinitializes *label-var* to null. If *label-var* is not null and the statement whose number is stored in *label-var* is contained in *innermost-enclosing-exception-block(f)* and *innermost-enclosing-exception-block(f)* is a finally statement, then the the algorithm chooses the statement whose number is stored in *label-var* as the successor, reinitializes *label-var* to null, pops the top of the stack *St* and assigns this value to *label-var* or *excp-var* depending upon whether the top of the stack was a label or an exception. If *label-var* is not null and the statement whose number is stored in *label-var* is not contained in *innermost-enclosing-exception-block(f)* and *innermost-enclosing-exception-block(f)* is not a finally statement, then the the algorithm chooses the exit node of *innermost-enclosing-exception-block(f)* as the successor. If *label-var* is not null and the statement whose number is stored in *label-var* is not contained in *innermost-enclosing-exception-block(f)* and *innermost-enclosing-exception-block(f)* is a finally statement, then the the algorithm chooses the exit node of *innermost-enclosing-exception-block(f)* as the successor and pops the top of the stack *St* (as the current reason for exiting the finally statement overrides the previous reason on the top of *St*). Note that at the exit node of a finally statement, either *excp-var* is not null or *label-var* is not null, because they store either the reason for entering the finally statement or the reason for exiting the finally statement created within the finally statement.

It is easy to see that the algorithm uses polynomial amount of space. Moreover, it has an accepting path if and only if there exists a path from entry-node of M to l along which $\langle p,\ obj \rangle$ reaches l.

Since the intraprocedural TIA is a subproblem of interprocedural TIA, as a corollary we get: interprocedural TIA is **PSPACE**-hard. Hence we have the following theorem:

Theorem 2 *Intraprocedural points-to analysis for programs with only single-level types and exceptions with subtyping is* **PSPACE***-complete, while the interprocedural case is* **PSPACE***-hard, even without dynamic dispatch.*

3.2.2 Complexity of points-to analysis for programs with only single-level types and dynamically-dispatched calls

Using a construction similar to the one given above, we can show the following[5]:

Theorem 3 *points-to analysis for programs with only single-level types and dynamic dispatch is* **PSPACE***-hard, even when exceptions are not allowed.*

In this construction, pointer dereferences are simulated through dynamically-dispatched calls instead of exceptions. Also it uses static (global) variables because they have to be modified through dynamically-dispatched calls instead of exceptions. The following is the reduction of PTA to the problem of points-to analysis of programs with only single-level types and dynamic dispatch, and without exceptions or threads (TIA1). As before, for the ease of presentation we show the reduction for an instance of PTA with at most 2 levels of indirection. The reduction of an instance of PTA with higher levels of indirection is a straightforward extension of the reduction for the case with at most 2 levels of indirection. The key idea in the reduction is to model the address of a variable using subtyping and to model pointer dereference using dynamically dispatched calls.

Again consider an instance of PTA:

```
void proc1( )
{
  int r1, ..., rm;       // m variables
  int *q1, ..., *qn;     // n variables
```

[5][PR96] contains an NP-hardness proof for a more restricted case.

```
  int **p1, ..., **ps;  // s variables

  ...
   // pointer assignments, ifs and whiles

}
```

The corresponding instance of TIA1 in the above mentioned subset of *RL* is as follows:

```
class int_type {
};
class multi_level_pointer {
  ...
};
class q1_type : public multi_level_pointer {
  ...
};
...
class qn_type : public multi_level_pointer {
  ...
};


class TIA1 {
  static int_type *r1, ..., *rm;
  static int_type *q1_var, ..., *qn_var;
  static q1_type *q1_addr;
    ...
  static qn_type *qn_addr;
  static multi_level_pointer *p1, ..., *ps;
```

```
  static void proc2( )
  {

    l1:  r1 = new int_type();  // corresponds to r1 in PTA
    ...
    ln:  rm = new int_type();  // corresponds to rm in PTA

    t1:  q1_addr = new q1_type();
     // corresponds to address of q1 in PTA
    ...
    tn:  qn_addr = new qn_type();
     // corresponds to address of qn in PTA

    ...
  }
}
```

The statements in PTA are translated to equivalent statements in TIA1 as follows:

A statement of the form fk: qi = &rj is translated to:

```
fk:  qi_var = rj;
```

A statement of the form fk: qi = qj is translated to:

```
fk:  qi_var = qj_var;
```

A statement of the form fk: pi = &qj is translated to:

```
fk:  pi = qj_addr;
```

*A statement of the form fk: *pi = qj is translated to:*

*// a statement of the form fk: *pi = &rj is translated similarly*

```
fk:  pi→func_fk();
```

Where func_fk is a virtual function of multi_level_pointer which is overriden in each qi_type as defined below:

```
void multi_level_pointer::func_fk( ) {
  func_fk is a virtual function of multi_level_pointer
}
void q1_type::func_fk( ) {
  q1_var = qj_var;
}
...
void qn_type::func_fk( ) {
  qn_var = qj_var;
}
```

*Similarly, a statement of the form fk: qi = *pj is translated to:*

```
fk:  pj→func_fk();
void multi_level_pointer::func_fk( ) {
  func_fk is a virtual function of multi_level_pointer
}
void q1_type::func_fk( ) {
  qi_var = q1_var;
}
...
void qn_type::func_fk( ) {
  qi_var = qn_var;
}
```

Combining the above two cases a statement of the form

`fk:  *pi = *pj` *is translated to:*

```
fk:  pi→func_fk();
```

```
void multi_level_pointer::func_fk( ) {
  func_fk is a virtual function of multi_level_pointer
}
void q1_type::func_fk( ) {
  pj→func_fk1();
}
...
void qn_type::func_fk( ) {
  pj→func_fkn();
}
void multi_level_pointer::fk1( ) {
  func_fk1 is a virtual function of multi_level_pointer
}
...
void multi_level_pointer::func_fkn( ) {
  func_fkn is a virtual function of multi_level_pointer
}
void q1_type::func_fk1( ) {
  q1_var = q1_var;
}
...
void qn_type::func_fk1( ) {
  q1_var = qn_var;
}
...
...
void q1_type::func_fkn( ) {
  qn_var = q1_var;
}
...
```

```
void qn_type::func_fkn( ) {
  qn_var = qn_var;
}
```

As before *ifs* and *whiles* are translated verbatim, except the test expressions, which we assume (without loss of generality) to be side-effect free and hence ignore in our analysis. Other statements that do not modify any pointer variable are translated into empty statements. This does not preserve program semantics, but it does not affect the mapping between the the points-to solutions of the two problems.

This shows that in order to explore the additional complexity for points-to analysis due to exceptions, we need to start with programs with only single-level types and without dynamic dispatch - as this seems to be the only known natural special case in **P**.

3.3 What is solvable in polynomial-time ?

In this section we show that for programs written in a subset of *RL* that has only single-level types and exceptions without subtyping, and that excludes dynamic dispatch, the problem of points-to analysis is in **P**. We will call this subset of *RL*, **SRL**. First we prove that Points-to-Algo, described in Chapter 2, computes the precise solution for programs written in *SRL*, and then we prove that the Points-to-Algo's worst-case complexity for this case is $O(n^7)$, where n is roughly the program size, hence proving that this case is in **P**. In practice we expect the performance of Points-to-Algo to be much better. Here our goal is only to prove that Points-to-Algo's worst-case complexity is polynomial-time and thus for simplicity we use program size as a loose upper bound for several quantities which we expect to be constant in practice.

Since any user-defined exception type has to extend the *Exception* class, we do not rule out subtyping completely when we say exceptions without subtyping. What is meant is that (1) no two user-defined *exception types* are related by subtype-supertype relationship and (2) there is no user-defined variable of type *Exception **. The second condition is needed because all the user-defined exception types are subtypes of *Exception*.

3.3.1 Precision of Points-to-Algo

This section shows that Points-to-Algo, described in Chapter 2, computes the precise solution for programs written in *SRL*.

Unfortunately, all realizable paths in a program are not necessarily executable and determining whether a particular branch of an *if statement* is executable is undecidable. Barth [Bar78] defined *precise up to symbolic execution* to be the precise solution under the assumption that all realizable program paths are executable (i.e., the result of a test is independent of previous tests and all the branches are possible). In the rest of this chapter, we use *precise* to mean *precise up to symbolic execution*.

Lemma 2 *Let l be a program point in a method M, let e be the entry node of M, let s be a start-node and let r be a realizable-path-from-a-start-node(r,s,l). Then there exists a unique decomposition of r such that (1) $r = r1 + r2$ (+ indicates concatenation), (2) r1 is a realizable-path-from-a-start-node(r,s,e) and (3) r2 is a realizable-path-from-entry-node(r,M,l).*

proof The proof is by straightforward induction on the length of r. □

3.3.1.1 Precision of Mark-reachable

This section proves that for programs written in *SRL*, *mark-reachable* defined in Appendix A computes the precise reachability information. Here by *precise reachability* we mean the following: a node l contained in a method M is said to be reachable from the entry node of M if and only if there exists a realizable path from the entry node of M to l.

Lemma 3 *Let l be a program point in a method M.*

1. *If ⟨empty,reachable⟩ is computed by mark-reachable at l, then there exists a realizable-path-from-entry-node-without-exception-and-label(r,M,l).*

2. *If ⟨excp-type,reachable⟩ is computed by mark-reachable at l, then either (1) there exists a realizable-path-from-entry-node-with-exception(r,M,l,excp-type) or (2) there exists a realizable-path-from-entry-node-with-exception(r,M,l,uiv) such that the typeof(uiv) is excp-type.*

3. *If ⟨label,reachable⟩ is computed by mark-reachable at l, then there exists a realizable-path-from-entry-node-with-label(r,M,l,label).*

proof The 3 claims can be proved simultaneously using straightforward induction on the number of iterations needed to compute ⟨*empty,reachable*⟩, ⟨*excp-type,reachable*⟩ or ⟨*label,reachable*⟩. □

Lemma 4 *Suppose there exists a realizable-path-from-entry-node(r,M,l). The following hold with respect to it.*

1. *If r is a realizable-path-from-entry-node-without-exception-and-label(r,M,l), then ⟨empty,reachable⟩ is computed by mark-reachable at l.*

2. *If r is a realizable-path-from-entry-node-with-exception(r,M,excp-type), then ⟨excp-type,reachable⟩ is computed by mark-reachable at l.*

3. *If r is a realizable-path-from-entry-node-with-exception(r,M,uiv), then ⟨typeof(uiv),reachable⟩ is computed by mark-reachable at l.*

4. *If r is a realizable-path-from-entry-node-with-label(r,M,label), then ⟨label,reachable⟩ is computed by mark-reachable at l.*

proof The 4 claims can be proved simultaneously by straightforward induction on the length of r. □

Theorem 4 *mark-reachable computes the precise reachability solution for programs written in SRL.*

proof: Lemma 3 implies that the solution computed by *mark-reachable* is a subset of the precise solution; while lemma 4 implies that the precise solution is a subset of the solution computed by *mark-reachable*. Hence the theorem. □

Intuitively, the reason why *mark-reachable* can compute the precise solution for programs written in *SRL* is that in the absence of dynamic dispatch and exceptions with subtyping, reachability does not depend upon the points-to solution and can be computed precisely without computing the points-to solution. In the absence of dynamic dispatch, the target of a call is statically known and in the absence of exceptions with subtyping, the signature of an exception thrown by a throw statement is also known statically from the declared type of the variable used by the throw statement. However, in the presence of dynamic dispatch and exceptions with subtyping, reachability depends upon the values of pointers, in this situation it can be shown that *mark-reachable* computes a safe solution (see Appendix E.2).

Now we show that Points-to-Algo computes a precise solution for programs written in *SRL*. The overall structure of the proof is as follows. First, we show that Phase I-pta is precise. Next, assuming Phase I-pta is precise, we prove that Phase II-pta is precise. Finally, assuming the first two phases are precise, we show that Phase III-pta is precise.

3.3.1.2 Precision of Phase I-pta

To prove that phase I-pta is precise, we need to define what we mean by *precision* of this phase because it computes in terms of unknown initial values. Let l be a program point in a method M. Let v be a user-defined variable of pointer type (recall we are considering *SRL* and only user-defined variables can be of pointer type) that is read at l. Let *object* be a heap-name or an unknown initial value of a parameter or global at the entry node of M. $\langle v, object\rangle$ belongs to the precise Phase I-pta solution at l if and only if there exists a realizable-path-from-entry-node(r,M,l) such that if r is followed, at l, v points to *object*.

Lemma 5 *Let l be a program point in a method M.*

1. *If* [*empty*,*Empty-context*,⟨*var,val*⟩] *is computed by Phase I-pta at l, then there exists a realizable-path-from-entry-node-without-exception-and-label(r,M,l) such that if r is followed, at l, var points to val.*

2. *If* [*excp-type*,*Empty-context*,⟨*var,val*⟩] *is computed by Phase I-pta at l, then there exists a realizable-path-from-entry-node-with-exception(r,M,l, excp-type) such that if r is followed, at l, var points to val.*

3. *If* [*uiv*,*Empty-context*,⟨*var,val*⟩] *is computed by Phase I-pta at l, then there exists a realizable-path-from-entry-node-with-exception(r,M,l, uiv) such that if r is followed, at l, var points to val.*

4. *If* [*label*,*Empty-context*,⟨*var,val*⟩] *is computed by Phase I-pta at l, then there exists a realizable-path-from-entry-node-with-label(r,M,l,label) such that if r is followed, at l, var points to val.*

proof The 4 claims can be proved simultaneously using straightforward induction on the number of iterations needed to compute [*empty*,*Empty-context*,⟨*var,val*⟩], [*excp-type*,*Empty-context*,⟨*var,val*⟩], [*uiv*,*Empty-context*,⟨*var,val*⟩] or [*label*,*Empty-context*,⟨*var,val*⟩]. □

Lemma 6 *Suppose there exists a realizable-path-from-entry-node(r,M,l). Let var be one of the following: (a) a parameter of M of pointer type that is read in M, (b) a global variable of pointer type that is in the used set*[6] *of M (defined in Figure B.3) or (c) a local variable of pointer type of M. Further suppose when r is followed, at l, var points to val. The following hold with respect to r.*

1. *If the realizable-path-from-entry-node(r,M,l) is the realizable-path-from-entry-node-without-exception-and-label(r,M,l), then* [*empty*,*Empty-context*,⟨*var,val*⟩] *is computed by Phase I-pta at l.*

[6] Recall that a global variable is considered used if it is either read or it is written.

2. *If the realizable-path-from-entry-node(r,M,l) is the realizable-path-from-entry-node-with-label(r,M,l,label), then* [*label*,*Empty-context*,⟨*var,val*⟩] *is computed by Phase I-pta at l.*

3. *If the realizable-path-from-entry-node(r,M,l) is the realizable-path-from-entry-node-with-exception(r,M,l,excp-type), then* [*excp-type*,*Empty-context*,⟨*var,val*⟩] *is computed by Phase I-pta at l.*

4. *If the realizable-path-from-entry-node(r,M,l) is the realizable-path-from-entry-node-with-exception(r,M,l,uiv), then* [*uiv*,*Empty-context*,⟨*var,val*⟩] *is computed by Phase I-pta at l.*

proof The 4 claims can be proved simultaneously using straightforward induction on the length of r.

Note that non-empty relevant contexts are generated only in the presence of modification through a pointer dereference (e.g., $p \rightarrow f = q;$), dynamic dispatch or exceptions with subtyping. Since all these are not present in programs written in *SRL*, the relevant contexts of the *PTA-dfelms* mentioned in the above two lemmas are *Empty-context*.

Theorem 5 *Points-to-Algo computes the precise Phase I-pta solution for programs written in SRL.*

proof: Let l be a program point in a method M. Suppose variable *var* is read at l. Suppose [*ecfi*,*Empty-context*,⟨*var,object*⟩] is computed by Points-to-Algo at l. Then Lemma 5 implies that there exists a realizable-path-from-entry-node(r,M,l) such that if r is followed, at l, *var* points to *object*. Thus the solution computed by Phase I-pta is a subset of the precise Phase I-pta solution. Now suppose that there exists a realizable-path-from-entry-node(r,M,l) such that if r is followed, at l, *var* points to *object*. Then lemma 6 implies that [*ecfi*,*Empty-context*,⟨*var,object*⟩] is computed by the Points-to-Algo at l for some *ecfi*. Thus the precise Phase I-pta solution is a subset of the solution computed by Phase I-pta. Hence the theorem. □

Intuitively, the reason why Points-to-Algo can compute the precise solution for programs written in *SRL* is that in the absence of multi-level pointers, dynamic dispatch and exceptions with subtyping, a *PTA-dfelm* generated at a node depends upon only one *PTA-dfelm* reaching that node and Points-to-Algo nevers needs to know if two *PTA-dfelms* hold simultaneously at a node. In the presence of multi-level pointers, dynamic dispatch or exceptions with subtyping, this is no longer true and as a result points-to analysis is no longer polynomial-time solvable in the presence of these constructs. It can be shown that for programs written in *RL*, Points-to-Algo computes a safe solution (see Appendix E.3).

3.3.1.3 Precision of Phase II-pta

To prove that phase II-pta is precise, first we need to define what we mean by *precision* of this phase. Let e be the entry-node of a method M. Let v be either a global of pointer type that is in the used set of M or let v be a parameter of M of pointer type that is read in M. Let *object* be a heap-name or an interface initial value. $\langle v_{init}, object\rangle$ belongs to the precise Phase II-pta solution at e if and only if there exists a start-node s and a realizable-path-from-a-start-node(r,s,e) such that if r is followed, at e, v points to *object*.

Lemma 7 *Let e be the entry node of a method M. If M is marked reachable by Phase II-pta, then there exist a start-node s and a realizable-path-from-a-start-node(r,s,e).*

proof The above claim can be proved using Theorem 4 and straightforward induction on the number of iterations needed to to mark M reachable. □

Lemma 8 *Let e be the entry node of a method M. If* [*Empty-context*,var_{init},*object*] *is computed by Phase II-pta at e, then there exists a start-node s and a realizable-path-from-a-start-node(r,s,e) such that if r is followed, at e, var points to object.*

proof The above claim can be proved using Lemma 7, Theorem 5 and straightforward induction on the number of iterations needed to compute [*Empty-context*,var_{init},*object*]. □

Lemma 9 *Let e be the entry-node of a method M. Let s be a start-node. Suppose there exists a realizable-path-from-a-start-node(r,s,e). Then M is marked reachable by Phase II-pta.*

proof The above claim can be proved using Theorem 4 and straightforward induction on the length of r. □

Lemma 10 *Let e be the entry-node of a method M. Let v be either a global of pointer type that is in the used set of M or let var be a parameter of M of pointer type that is read in M. Let object be a heap-name or an interface initial value. Let s be a start-node. Suppose there exists a realizable-path-from-a-start-node(r,s,e). Then the following hold with respect to r.*

If when r is followed, at e, v points to object, then [*Empty-context*,var_{init},*object*] *is computed by Phase II-pta of Points-to-Algo at e.*

proof The above claim can be proved using Lemma 9, Theorem 5 and straightforward induction on the length of r. □

Theorem 6 *Points-to-Algo computes the precise Phase II-pta solution for programs written in SRL.*

proof: Lemma 8 implies that the solution computed by Phase II-pta is a subset of the precise Phase II-pta solution; while Lemma 10 implies that the precise Phase II-pta solution is a subset of the solution computed by Phase II-pta. Hence the theorem. □

3.3.1.4 Precision of Phase III-pta

To prove that phase III-pta is precise, first we need to define what we mean by *precision* of this phase. Let l be a program point in a method *M*. Let v be a user-defined variable of pointer type that is read at l. Let *object* be a heap-name or an interface initial value. $\langle v, object\rangle$ belongs to the precise Phase III-pta solution at l if and only if there exists a start-node s and a realizable-path-from-a-start-node(r,s,l) such that if r is followed, at l, v points to *object*.

Lemma 11 *Let l be a program point in a method M. Let e be the entry node of M. Let v be a user-defined variable of pointer type that is read at l. If ⟨v,object⟩ is computed by Phase III-pta of Points-to-Algo, then there exists a start-node s and a realizable-path-from-a-start-node(r,s,l) such that if r is followed, at l, v points to object.*

proof Suppose [*ecfi*,*Empty-context*,⟨*v,object*⟩] ∈ *l.reaching-PTA-dfelms*. Using Lemma 5 there exists a realizable-path-from-entry-node(r2,M,l) such that if *r2* is followed, at *l*, *v* points to *object*. Phase III-pta is done only in methods marked reachable by Phase II-pta. As a result, using Lemma 7, there exists a start node *s* and a realizable-path-from-a-start-node(r1,s,e). Hence, concatenation of r1 and r2 gives r.

Suppose [*ecfi*,*Empty-context*,⟨v,x_{init}⟩] ∈ *l.reaching-PTA-dfelms*
and [*Empty-context*,*object*,x_{init}] ∈ *M.cv-to-uiv-bindings*. Now using Lemma 5 there exists a realizable-path-from-entry-node(r2,M,l) such that if *r2* is followed, at *l*, *v* points to x_{init}. Next using Lemma 8 there exists a start node *s* and a realizable-path-from-a-start-node(r1,s,e), such that if *r1* is followed, at *e*, *x* points to *object*. Hence, concatenation of r1 and r2 gives r. □

Lemma 12 *Let l be a program point in a method M. Let e be the entry node of M. Let v be a user-defined variable of pointer type that is read at l. Let object be a heap-name or an interface initial value. Suppose there exists a start-node s and a realizable-path-from-a-start-node(r,s,l) such that if r is followed, at l, v points to object. Then ⟨v,object⟩ is computed by Phase III-pta of Points-to-Algo.*

proof Using Lemma 2, *r* = *r1* + *r2*, where *r1* is realizable-path-from-a-start-node(r1,s,e) and *r2* is a realizable-path-from-entry-node(r2,M,l).

First, using Lemma 9, *M* is marked reachable by Phase II-pta.

Suppose when *r2* is followed, at *l*, *v* points to *object*. Then using Using Lemma 6, [*ecfi*,*Empty-context*,⟨*v,object*⟩] ∈ *l.reaching-PTA-dfelms* and hence Phase III-pta computes ⟨*v,object*⟩ at program point *l*.

Suppose when *r1* is followed, at *e*, *object* is the concrete value of a variable *x* and when *r2* is followed, at *l*, *v* points to x_{init}. Now using Lemma 10, [*Empty-context*,*object*,x_{init}]

$\in$ *M.cv-to-uiv-bindings*, and using Lemma 6, [*ecfi*,*Empty-context*,$\langle v, x_{init}\rangle$] $\in$ *l.reaching-PTA-dfelms*. As a result, Phase III-pta computes $\langle v, object\rangle$ at program point *l*. □

Theorem 7 *Points-to-Algo computes the precise Phase III-pta solution for programs written in SRL.*

proof: Lemma 11 implies that the solution computed by Points-to-Algo is a subset of the precise Phase III-pta solution; while Lemma 12 implies that the precise Phase III-pta solution is a subset of the solution computed by Points-to-Algo. Hence the theorem. □

3.3.2 Complexity of Points-to-Algo

In this section we show that the worst-case complexity of Points-to-Algo for programs written in *SRL* is $O(n^7)$.

We will use the following notations:

- Let the total number of statements in the program be n_1,
- let the sum of the numbers of arguments passed at call sites be n_2,
- let n be $n_1 + n_2$,
- let the maximum number of arguments passed at a call site be A,
- let the total number of user-defined variables of pointer type be $\underline{N_{var}}$,
- let the total number of dynamically created objects (identified by their creation sites) be $\underline{N_{hobj}}$,
- let the total number of call sites be $\underline{N_{call}}$,
- let the total number of method exit nodes be $\underline{N_{proc-exit}}$,
- let the total number of exception types be $\underline{N_{excp-type}}$,

- let the total number of exception objects be $\underline{N_{excp-obj}}$ (identified by their creation sites),
- let the maximum number of labels generated[7] by the Points-to-Algo in a method be N_{label},
- let the maximum number of unknown initial values generated in a method be N_{uiv},
- let the maximum number of catch statements associated with a try statement be N_{catch}, and
- let l be a program point in a method M.

Complexity of Phase 0: The initial call graph is constructed in linear time because *SRL* does not have dynamic dispatch. The decomposition of the initial call graph into strongly connected components is also done in linear time using the algorithm given in [CLR92]. Thus Phase 0 is done in linear time.

Complexity of Phase I-pta: First we consider the complexity of *mark-reachable*. The maximum number of data-flow elements (computed by *mark-reachable*) at a node is $O(N_{excp-type}+N_{label}+1)$ because each data-flow element has the form $\langle$*ecfi,reachable*$\rangle$, where *ecfi* is an exception type, a label or *empty*. At each node, except a call node, the exit node of a method or the exit node of a try statement, a constant amount of work is done for each new data-flow element reaching that node. At each exit node of a try statement, at most $O(N_{catch})$ amount of work is done in propagating a data-flow element to the appropriate catch statements. For each data-flow element along an interprocedural edge, at most $O(N_{label} + 1)$ amortized amount of work is done. Thus the worst-case complexity of *mark-reachable* is $O(n * (N_{excp-type} + N_{label} + 1)(N_{catch} + 1) + N_{call}(N_{excp-type} + N_{label} + 1)(N_{label} + 1))$. Again note that this is only a rough upper bound and we expect N_{label} and the number of data-flow elements at a program point to be constant in practice.

---

[7] Recall that a label is generated by the Points-to-Algo, as an *ECFInfo*, at a labelled break statement, a labelled continue statement, a return statement or while falling through the body of a try or catch which has a finally associated with it.

Now we consider the complexity of the rest of Phase I-pta. Each *PTA-dfelm* at l has one of the following two forms:

- [*ecfi*,*Empty-context*,$\langle$*var*,*val*$\rangle$]
- [*Empty-context*,*exception-object*]

Let the maximum number of *PTA-dfelms*-**a** at any node be $\underline{N_{PTA-dfelm-\mathbf{a}}}$ and let the maximum number of *PTA-dfelms*-**b** at any node be $\underline{N_{PTA-dfelm-\mathbf{b}}}$. Thus, $N_{PTA-dfelm-\mathbf{a}}$ is $O((N_{label}+1+N_{excp-type}+N_{uiv})N_{var}(N_{hobj}+N_{uiv}))$ and $N_{PTA-dfelm-\mathbf{b}}$ is $O((N_{excp-obj}+N_{uiv}))$.

First consider a node l that is not a call node, throw node, exit node of a method or the exit node of an anonymous procedure representing a try statement nested inside a finally statement. For each *PTA-dfelm* reaching l a constant amount of work is done. Hence, the total amount of work done for such nodes is bounded by $O(n(N_{PTA-dfelm-\mathbf{a}}+N_{PTA-dfelm-\mathbf{b}}))$.

Now consider a node l that represents a throw statement. The number objects thrown by l is at most $O(N_{excp-obj}+N_{uiv})$. Thus, for each *PTA-dfelm*-**a** reaching l at most $O(N_{excp-obj}+N_{uiv})$ work is done in computing the *ECFInfo*'s of the resulting *PTA-dfelms*-**a**. Hence, the total amount of work done for throw nodes is bounded by $O(n(N_{PTA-dfelm-\mathbf{a}}(N_{excp-obj}+N_{uiv})))$.

Now consider a call node l. Suppose the reaching *PTA-dfelm* is a *PTA-dfelm*-**a**. The work done in computing new actual-to-uiv-bindings implied by the *PTA-dfelm*-**a** is $O(A+1)$. At most $O(N_{excp-obj}+N_{uiv}+N_{label}+1)$ work is done in computing *ECFInfo*'s of *PTA-dfelms*-**a** propagated across the call by *propagate-across-call-PTA-dfelm*-**a**. Similarly, for each *PTA-dfelm*-**b** reaching l, $O(N_{label}+1)$ is done by *propagate-across-call-PTA-dfelm*-**b**.

The total amount of work done in propagating *PTA-dfelms* from the exit node of the target of a call node to the return site of the call node is $O((N_{excp-type}+N_{label}+N_{uiv}+1)(N_{excp-obj}+N_{uiv})(N_{hobj}+N_{uiv})N_{PTA-dfelm-\mathbf{a}}+(N_{excp-obj}+N_{uiv})N_{PTA-dfelm-\mathbf{b}})$. This is because (a) an unknown initial value in a points-to can have at most $N_{hobj}+N_{uiv}$

values at a call site, (b) an unknown initial value in an *ECFInfo* can have at most $N_{excp-obj}+N_{uiv}$ values at a call site, (c) each binding between an actual and an unknown initial value at a call site can hold under at most $N_{excp-type}+N_{label}+N_{uiv}+1$ *ECFInfo*'s and (d) for each instantiation of unknown initial values in a *PTA-dfelm*-**a**, all the actual-to-uiv bindings used in the instantiation must have the same *ECFInfo*. A part of this work may be done at the call node and rest may be done at the exit node.

Thus the total amount of work done at call nodes and method exit nodes is $O(N_{call}((N_{excp-type}+N_{label}+N_{uiv}+1)(N_{excp-obj}+N_{uiv})(N_{hobj}+N_{uiv})N_{PTA-dfelm-\mathbf{a}}+(N_{excp-obj}+N_{uiv})N_{PTA-dfelm-\mathbf{b}}+A))$.

Now each of N_{var}, $N_{excp-type}$, N_{label}, N_{hobj}, N_{call}, N_{uiv}, $N_{excp-obj}$ and A is $O(n)$. Hence the overall, worst-case complexity of Phase I-pta is $O(n((N_{excp-type}+N_{label}+N_{uiv}+1)(N_{excp-obj}+N_{uiv})(N_{hobj}+N_{uiv})N_{PTA-dfelm-\mathbf{a}}+(N_{excp-obj}+N_{uiv})N_{PTA-dfelm-\mathbf{b}}+A))$ or $O(n^7)$. Again note that this is only a rough upper bound and we expect several quantities like N_{label}, N_{uiv} and the number of points-tos at a program point to be constant in practice.

Complexity of Phase II-pta: For programs written in *SRL*, each cv-to-uiv binding has the form [*Empty-context*,*cv*,*uiv*]. Thus, there are at most $O((N_{hobj}+N_{uiv})N_{uiv})$ cv-to-uiv bindings at the entry node of a method. For programs written in *SRL*, each actual-to-uiv binding at a call site has the form [*ecfi*,*Empty-context*,$\langle$*actual,uiv*$\rangle$]. Thus, for each cv-to-uiv binding reaching the entry node of a method, at most $O(N_{uiv})$ amount of work is done at a call site contained in the method in propagating cv-to-uiv bindings to the entry node of the target of the call site. Thus, for each cv-to-uiv binding reaching the entry node of a method *M*, at most $O(N_{uiv}*C_M)$ work is done at the call sites contained in *M*, where C_M is the total number of the call sites in *M*. Thus the worst-case complexity of Phase II-pta is $O((N_{hobj}+N_{uiv})N_{uiv}*N_{uiv}*N_{call}+N_m(N_{hobj}+N_{uiv})N_{uiv})$, where N_m is the total number of methods in the program, that is the worst-case complexity of this phase is $O(n^4)$.

Complexity of Phase III-pta: As stated above, the maximum number of *PTA-dfelms*-**a** at a node is $O((N_{label}+1+N_{excp-type}+N_{uiv})N_{var}(N_{hobj}+N_{uiv}))$. This means we need to instantiate at most $O(N_{var}(N_{hobj} + N_{uiv}))$ points-tos at a node. Each unknown initial value in a points-to can have at most $O(N_{uiv} + N_{hobj})$ concrete values. Thus at most $O(N_{var}(N_{hobj}+N_{uiv})(N_{uiv}+N_{hobj}))$ amount of work is done at each node. Hence the worst-case complexity of this phase is $O(n(N_{var}(N_{hobj}+N_{uiv})(N_{uiv}+N_{hobj})))$ or $O(n^4)$.

Since the worst-case complexity of each of the phases is $O(n^7)$, we have the following theorem:

Theorem 8 *The worst-case complexity of Points-to-Algo for programs written in SRL is $O(n^7)$.*

As a result of Theorems 7 and 8, we have the following theorem:

Theorem 9 *points-to analysis of programs written in SRL is in* **P** *and can be precisely solved in $O(n^7)$ worst-case time.*

Using an analysis very similar to the above analysis, it can be shown that for programs that are written in *SRL* and that do not have any exception (i.e., without try, catch, throw and finally statements), the worst-case complexity of Points-to-Algo is $O(n^4)$. As a result, we have the following theorem:

Theorem 10 *points-to analysis of programs that are written in SRL and that do not have any exception (i.e., without try, catch, throw and finally statements) can be precisely solved in $O(n^4)$ worst-case time.*

Theorem 10 improves the $O(n^7)$ worst-case bound obtainable using previous techniques [LR91, RHS95].

Theorems 2, 3 and 9 show that in the presence of dynamic dispatch and exceptions, among all the reasonable special cases that we have considered, programs written in *SRL* comprise the only natural special case that is in **P**. Note that just adding subtyping for exception types and allowing overloaded *catch* clauses increase complexity from P to **PSPACE**-hard.

3.3.3 Complexity of intraprocedural points-to analysis for programs with only single-level types and without dynamic dispatch or exceptions

Theorem 11 *Intraprocedural points-to analysis for programs with only single-level types and without dynamic dispatch or exceptions is in non-deterministic log-space and hence* **NC**.

Recall that non-deterministic log-space is the set of languages accepted by non-deterministic Turing machines using logarithmic space [Pap94], **NC** is the class of efficiently parallelizable problems and **NC** contains non-deterministic log-space.

Proof Given a program point l in a method M and a points-to u, to check whether u holds at l, we non-deterministically (i.e., predecessors are chosen non-deterministically) search backwards starting at l. As we move from a node to its predecessor, we replace u by another points-to which is necessary and sufficient for u to hold at the top of the current node. We can do this because we have only single-level types. Thus, at any instant, we have only one points-to to check. Finally, either we reach a node which creates this points-to or the search fails. Also note that, in this case, shortest path associated with a points-to is at most $O(n^3)$ in length, where n is the number of statements in M. $\Box$

Let us denote intraprocedural points-to analysis for programs with only single-level types and without dynamic dispatch or exceptions by *TIA2*. Now we will show that *TIA2* is non-deterministic log-space hard. Consider the graph reachability problem (R_G): given a directed graph G and two vertices v_1 and v_2 in G, is there a directed path from v_1 to v_2 ? This problem is known to be non-deterministic log-space complete [Jon75]. It is easy to reduce[8] R_G (using logarithmic space) to *TIA2*. Consider an instance of $R_G : \langle G, V, E, v_1, v_2 \rangle$, where G is a directed graph, V is the set of vertices of G, E is the set of edges of G and the goal is to find whether there is a directed path from vertex v_1 to v_2. The corresponding instance of *TIA2* has one variable for each

[8]This was pointed out to me by Prof. Eric Allender.

vertex in V and one assignment statement for each edge in E. Each of these assignment statements is contained in an *if* statement and all the *if* statements are contained in a single *while* statement. Thus the instance of *TIA2* has the following form:

class T { };

void TIA2( ) {

  T $*v_1$, ..., $*v_n$; // for each $v_i \in V$

  1: v_1 = new T();

    $v_2 = 0$;

    ...

    $v_n = 0$;

  while(_) {

    if (_)

      $v_{k1} = v_{l1}$; // assuming the edge $e_1 = (v_{l1}, v_{k1})$

    ...

    if (_)

      $v_{km} = v_{lm}$; // assuming the edge $e_m = (v_{lm}, v_{km})$

  }

  2:

}

It is easy to see that there exists a directed path from v_1 to v_2 in G if and only if v_2 can point to $object_1$ at program point **2**.

The above reduction and Theorem 11 show that *TIA2* is non-deterministic log-space complete. Hence we have the following theorem:

Theorem 12 *Intraprocedural points-to analysis for programs with only single-level types and without dynamic dispatch or exceptions is non-deterministic log-space complete.*

Chapter 4

Def-Use Analysis and Application to Data-flow-based Program Testing

In this chapter, we apply *RCI* to finding def-uses in object-oriented libraries (i.e., libraries written in *RL*). We will call the instantiation of *RCI* for finding def-uses, **Def-Use-Algo** [CR99]. We also show how the information computed by Def-Use-Algo can be used for generating relevant test cases. The rest of this chapter is organized as follows. Section 4.1 discusses the new challenges in applying data-flow-based testing to object-oriented libraries. Section 4.2 defines a def-use association. Section 4.3 presents some additional definitions needed for describing Def-Use-Algo. Section 4.4 describes the data-flow elements computed during Phase I of Def-Use-Algo. Section 4.5 presents an overview of the different steps of Def-Use-Algo. Section 4.6 shows how the information computed by Def-Use-Algo can be used for generating relevant test cases. Section 4.7 presents some details of Phase I of Def-Use-Algo. Finally, Section 4.8 presents details of Phase II of Def-Use-Algo.

We will skip the proofs of correctness and complexity of Def-Use-Algo as these are very similar to the proofs of correctness and complexity of Points-to-Algo given in Chapter 3 and Appendix E.

4.1 Introduction

Intuitively, there exists a def-use association between two program points if and only if there exists a program execution for which a location is defined at the first program point and later this location is read at the second program point. The goal of data-flow-based testing [RW85, FW88, OW88, LCS89, Ost90, OW91, FW93, HR94, HFGO94, FI98] is to write test cases such that the possible def-use associations in a program are exercised by the test cases. Data-flow-based testing is based on the intuition that until the result of a computation has been used during testing, a program has not been tested with respect to this computation. Many useful data-flow testing criteria can be defined using def-use relationships. [RW85] showed that for a simple Pascal-like language these criteria form a hierarchy of testing criteria between the all-paths criterion and all-nodes criterion. In this hierarchy, a testing criterion t_1 subsumes another testing criterion t_2 if and only if coverage of t_1 implies coverage of t_2. The advantage of testing criteria based on data-flow information and other control flow graph characteristics (e.g., edges, nodes etc) is that these criteria do not depend upon any specification and their satisfaction can be automatically checked (at least to a great extent), unlike testing for adherence to the functional specification of a system. Since most software systems lack formal specification, this approach to program testing is very attractive.

In this chapter, we are interested in the data-flow-based **unit testing** of O-O libraries. As stated earlier, since a complete program can be considered to be a library with a single entry point (i.e., *main*), the results of this chapter also apply to complete programs. Due to exceptions, dynamic dispatch and potential aliasing at the entry node of a *public interface method*, there are several new challenges in computing def-use associations in an object-oriented library. Consider the example library $L_{example}$ given in Figure 1.5. It has three *public interface methods*: *method2*, *method4* and *method6*. If $a1_{init}$ and $a2_{init}$, the unknown initial objects to which *a1* and *a2* point at the entry node of *method1*, are the same object, statement **4** modifies the *next* field of $a2_{init}$ in addition to the *next* field of $a1_{init}$. As a result, there is a def-use relationship between statements **4** and **6** if and only if $a1_{init}$ and $a2_{init}$ are the same. Consequently, even if all-path coverage is attained by a set of test cases for $L_{example}$, unless these test cases make $a1_{init}$ and $a2_{init}$ identical, the def-use relationship between statements **4** and **6** will not be executed. *This shows that due to potential def-use associations implied by potential aliasing, all-path coverage does not imply all-def-use coverage for a library.* In addition to paths, the context - aliasing between the initial values at the entry nodes of *public interface methods* of a library and the concrete types of these initial values - also needs to be considered in choosing test data. This problem was mentioned in [HR94], but the algorithm presented there cannot compute such potential def-uses.

When the value of a location is read at a program point and used for a computation other than the evaluation of a test expression governing branching, the location is said to have a *c-use* [RW85] at the program point. In contrast, When the value of a location is read at a program point and used for evaluating a test expression used for deciding branching (e.g., in a test expression of an *if* statement), the location is said to have a *p-use* at the program point. At a dynamically dispatched call site, the target method is chosen using the type of the receiver object. As a result, there is a new kind of *p-use* of the receiver variable along every interprocedural edge from a dynamically dispatched call site to the entry node of a method that can be potentially invoked from the call site. Consider statement **5** in Figure 1.5. There is a p-use of r along the interprocedural edge e_1 from statement **5** to the entry node of $A :: update$, and consequently there is a def-use relationship between statement **3** and the p-use of r on the edge e_1 if and only if the concrete type of $a1_{init} \in \{A, B\}$. Similarly, there is a p-use of r along the interprocedural edge e_2 from statement **5** to the entry node of $C :: update$, and consequently there is a def-use relationship between statement **3** and the p-use of r on the edge e_2 if and only if the concrete type of $a1_{init}$ is C.

When an exception object is thrown by a *throw* statement and then later caught by a *catch* statement, there exists a def-use relationship between the throw statement and the entry node of the catch statement where the parameter of the catch statement is assigned the exception object. Such def-uses can also depend upon context if the exception object is an unknown initial value. Consider statement **15** in Figure 1.5. The exception object thrown by statement **15** (i.e.,$e1_{init}$) is caught by the catch statement at program point **17** if and only if the concrete type of $e1_{init}$ is *ET2* or a subtype of *ET2*. As a result, there exists a def-use relationship between statement **15** and the entry node of the catch statement if and only if the concrete type of $e1_{init}$ is *ET2* or a subtype of *ET2*. Similarly, if the exception is caught by the catch statement, there also exists a def-use relationship between statements **14** and **18**. For def-use coverage, such def-uses between throws and catches and other def-uses arising from flow due to exceptions also need to be exercised.

The above discussion shows that the notion of def-use associations needs to be extended for O-O libraries. In this chapter, we present a new def-use algorithm that can compute the above kinds of def-use relationships (besides other ordinary def-use relationships) in libraries written in *RL*. We also show how the contexts associated with def-use relationships can be used for generating relevant test cases.

4.2 Def-use Associations

This section defines what we mean by a def-use association. Formally, a def-use association is a triplet ⟨*loc, def-point, use-point*⟩, where *loc* is

1. a user-defined variable,
2. a field of an unknown initial value or
3. a field of a heap-name

of **any type**. A def-use association ⟨*loc, def-point, use-point*⟩ belongs to the *precise* def-use solution of a library L if and only if there exists an execution path p from the entry node of a *public interface method* of L to *use-point* (contained in L) and an environment e at the entry node of the *public interface method*, such that

- if p is executed starting with the environment e at the entry node of the *public interface method*, *loc* is defined at *def-point* and this definition of *loc* is used at *use-point*.

Traditionally [RW85], *def-point* is an assignment statement that assigns to the location *loc*, and *use-point* is either a statement which reads location *loc* and uses (*c-use*) its value in a computation other than the evaluation of a test expression which governs branching or *use-point* is an edge out of a condition node and the location *loc* is read (*p-use*) at the condition node for evaluating the test expression governing branching at the condition node. Due the reasons described in Section 4.1, for object-oriented libraries, in addition to the above mentioned standard interpretations of *loc*, *def-point* and *use-point*, we allow *loc*, *def-point* and *use-point* to have the following additional interpretations. A *loc* can also be an exception object, which could be a heap-name or an unknown initial value. A *def-point* can also be a throw statement. A *use-point* can also represent the entry node of a catch statement and an interprocedural edge out of a dynamically dispatched call site. As a result, we get the following new kinds of def-use associations:

1. When *loc* is an exception object, *def-point* is a throw statement and *use-point* is the entry node of a catch statement, we have a def-use association between a throw statement that throws an exception object and the entry node of the corresponding catch statement that catches the exception object. Since the parameter of the catch statement is assigned the caught exception object, this kind of def-use association is a natural extension of the def-use association that exists between two program points when an assignment is made to a variable at the first program point and at the second program point the value of this variable is assigned to another variable. A throw statement can be considered to be making an implicit assignment to a location used for keeping track of uncaught exceptions and the entry node of the catch statement can be considered to be assigning the value of this location to the parameter of the catch statement. Thus, we also have a *c-use* at the entry node of the catch statement.

2. When *loc* points to the receiver object of a dynamically dispatched call site, *def-point* is a program point that assigns to *loc* and *use-point* is an interprocedural edge between the dynamically dispatched call site and a possible target of the call site, we have a new kind of def-use association between *def-point* and *use-point.* This situation is analogous to the case where a def-use association arises due to the use of a location in evaluating the test expression governing branching at a condition node. Here the value of *loc* gives the receiver object whose concrete type is used to determine the target/branching at the call site. Thus, it is natural to classify the use of *loc* in this situation as a *p-use*.

4.3 Definitions

This section presents some definitions needed to explain Def-Use-Algo.

Unknown initial def: We use $var_{initdef}$ to represent all the definitions of *var* reaching the entry node of a method. Here *var* can be either a global or a field of a unknown initial value, and it can be of any type.

Interface initial def: An unknown initial def at the entry node of a *public interface method* is called an interface initial def.

Safe def-use solution: A def-use solution that is a superset of the precise solution is called a *safe* solution. Like Points-to-Algo , Def-Use-Algo computes a safe solution.

4.4 Phase I data-flow elements

The Phase I of Def-Use-Algo consists of two subphases: Phase I-pta (this is same as the Phase I-pta described in Chapter 2) and Phase I-dua, during which Def-Use-Algo does def-use analysis.

Def-Use-Algo computes two kinds of data-flow elements during phase I: **PTA-dfelms** and **DUA-dfelms**, shown in Figure 4.1. *PTA-dfelms* are same as described in Section 2.3. *DUA-dfelms* are used for def-use analysis, are computed during Phase I-dua, and represent variable definitions and def-use associations. We will use **dfelm** to denote both *PTA-dfelm* and *DUA-dfelm*.

4.4.1 Data-flow elements for def-use analysis (*DUA-dfelms*)

4.4.1.1 DUA-dfelm-a

A *DUA-dfelm*-**a** has the form [*ECFInfo*, *relevant context-1*, ⟨*var, def-point,* ⟨*relevant context-2, value*⟩⟩], where (1) *var* is a user-defined variable, a field of an unknown initial value or a field of a heap-name of **any type** and (2) *def-point* is a program point (i.e., a definition point) or an unknown initial def. *relevant context-1* and *relevant context-2* are relevant contexts as defined in the case of *PTA-dfelms*. *relevant context-1* represents the

1. There are two kinds of **PTA-dfelms**:
PTA-dfelm-a. [*ECFInfo*, *relevant context*, *points-to*]
PTA-dfelm-b. [*relevant context*, *exception object*]

2. There are five kinds of **DUA-dfelms**:
DUA-dfelm-a. [*ECFInfo*, *relevant context-1*,
$\langle$*var, def-point,* $\langle$*relevant context-2, value*$\rangle\rangle$]
DUA-dfelm-b. [**excp-def,** *relevant context*, *exception object*, *throw-point*]
DUA-dfelm-c. [*relevant context*, $\langle$*var, def-point, use-point*$\rangle$]
DUA-dfelm-d. [*relevant context*, $\langle$*exception-object, throw-point, catch-point*$\rangle$]
DUA-dfelm-e. [*relevant context*, $\langle$*var, def-point, use-edge*$\rangle$]

Figure 4.1: Phase I data-flow elements

contexts in which *var* is defined at *def-point*. If *var* is of pointer type, *value* is the object that was assigned to *var* at *def-point* and *relevant context-2* represents the contexts in which *var* takes this *value*. If *var* is not a pointer, *relevant context-2* is *Empty-context* and *value* is *don't-care*. As we will see later, *value* is needed for computing *p-uses* on interprocedural edges at dynamically dispatched call sites. *DUA-dfelms*-**a** are used for propagating variable definitions to their use points. For example, [*empty*,$\langle$*(*$a1_{init}$ **eq** $a2_{init}$*),empty,empty*$\rangle$,$\langle$$a2_{init}$*.next,4,*$\langle$*Empty-context,*$a3_{init}$$\rangle\rangle$] represents the potential definition of $a2_{init}$*.next* at statement **4** in Figure 1.5. Intuitively, *rc1* governs the occurrence of the definition which requires an alias condition to occur; while *rc2* governs the identity of the object assigned. At statement **4**, there is a direct assignment so *rc2* is *Empty-context*. In contrast, the *DUA-dfelm*-**a** [*empty*,*empty*,$\langle$*global2,6,*$\langle\langle$*(*$a1_{init}$ **neq** $a2_{init}$*), empty, empty*$\rangle$,$a2_{init}$*.next*$_{init}$$\rangle\rangle$] represents the definition of *global2* at statement **6**.

4.4.1.2 DUA-dfelm-b

A *DUA-dfelm*-**b** has the form [**excp-def,** *relevant context*, *exception object*, *throw-point*], where (1) *throw-point* is the program point at which *exception object* has been thrown, (2) *exception object* is an unknown initial value or a heap-name and (3) **excp-def** is a keyword. *relevant context* represents the contexts in which *exception object* is thrown at *throw-point*. For example, [**excp-def,***Empty-context*,$e1_{init}$,*15*] represents the exception thrown at statement **15** in Figure 1.5. *DUA-dfelms*-**b** are used for propagating exception definitions from throw statements to corresponding catch statements.

4.4.1.3 DUA-dfelm-c

A *DUA-dfelm*-**c** has the form [*relevant context*, ⟨*var, def-point, use-point*⟩], where (1) *var* and (2) *def-point* are same as in *DUA-dfelm*-**a**, and (3) *use-point* is a program point. *DUA-dfelms*-**c** capture def-uses of variables, although these are parametrized in terms of unknown initial defs and unknown initial values. Intuitively, *var* is a variable which is defined at *def-point* in contexts represented by *relevant context* and this definition of *var* is used at *use-point*. For example, the *DUA-dfelm*-**c** [⟨$(a1_{init}$ **eq** $a2_{init})$, *empty, empty*⟩,⟨$a2_{init}.next$,*4,6*⟩] represents the def-use of $a2_{init}.next$ between statements **4** and **6** in Figure 1.5.

4.4.1.4 DUA-dfelm-d

A *DUA-dfelm*-**d** has the form [*relevant context*, ⟨*exception-object, throw-point, catch-point*⟩], where (1) *exception-object* is an unknown initial value or a heap-name, (2) *throw-point* is the number of a throw statement and (3) *catch-point* is the number of the entry node of a catch statement. A *DUA-dfelm*-**d** represents a def-use association between a throw statement and a catch statement, it means *exception-object* thrown at *throw-point* is caught at *catch-point* in contexts represented by *relevant context*. For example, [⟨*empty,empty,*$(type(e1_{init}) \leq ET2)$⟩,⟨$e1_{init}$*,15,17*⟩] represents the def-use between the throw at statement **15** and the catch at statement **17** in Figure 1.5.

4.4.1.5 DUA-dfelm-e

A *DUA-dfelm*-**e** has the form [*relevant context*, ⟨*var, def-point, use-edge*⟩], where (1) *use-edge* is an edge out of a condition node (representing the test expression of an *if* or *while*) or it is an interprocedural edge between a dynamically dispatched call site and one of its targets, and (2) *var* and (3) *def-point* are same as in *DUA-dfelm*-**a**. These *DUA-dfelms* represent *p-uses*. For example,

[$\langle$*empty,(type(*$a1_{init}$*)*$\in$*A::update),empty*$\rangle$,$\langle$*r,3,*$\langle$*5,A::update*$\rangle\rangle$] and [$\langle$*empty,(type(*$a1_{init}$*)*$\in$*C::update),empty*$\rangle$,$\langle$*r,3,*$\langle$*5,C::update*$\rangle\rangle$] represent the *p-uses* due to dynamic dispatch at statement **5** in Figure 1.5.

4.4.2 Limiting relevant context

Let d be a *dfelm* at the exit node of a method M, C be a call site that invokes M and rc be the relevant context of d. If rc evaluates to *true* at C, any relevant context t that is contained in rc (i.e., the set of conjuncts of t is a subset of the set of conjuncts of rc) also evaluates to *true* at C. As a result, we have the following theorem which is a generalization of the Theorem 1 given in Section 2.3:

Theorem 13 *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 13, instead of the complete relevant context, Def-Use-Algo can store 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). 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 generated conjuncts of this kind associated with a *dfelm*; the rest of the conjuncts are dropped.

4.5 Overview of def-use algorithm

The def-use algorithm presented in this chapter is an application of relevant context inference to finding def-use associations; it uses the points-to algorithm presented in Chapter 2. First we will give a brief overview of the various steps of the def-use algorithm and then we will present the details of these steps.

The Def-Use-Algo performs the following four phases on the SCC-DAG.

4.5.1 Phase 0

This is same as described in Section 1.6.

4.5.2 Phase I

Def-Use-Algo traverses the SCC-DAG in a reverse topological order (bottom-up) and analyzes each method assuming parameters and global variables have unknown initial values. It performs the following two subphases on each SCC.

Phase I-pta: This is same as described in Chapter 2.

Phase I-dua: During this subphase Def-Use-Algo finds def-use associations using the points-to solution computed in Phase I-pta. For each method Def-Use-Algo computes, in terms of unknown initial values, a safe approximation to the method's complete transfer function for def-use analysis. We will call this approximation, the **def-use summary transfer function**. The def-use summary transfer function of a method M is the set of data-flow elements (representing variable definitions) that reach the exit node of M and that do not represent definitions of local variables of M. This function summarizes the possible effects of method invocation on variable definitions. Again, the def-use summary transfer functions of methods in the same SCC have cyclic dependences, so they are computed simultaneously by fixed point iteration. In contrast, the def-use 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 of SCC-DAG, without iteration.

The results of this subphase on a SCC are two-fold: (i) a def-use solution at each node of every method in the SCC and (ii) a def-use summary transfer function for every method in the SCC, both of which are parametrized by unknown initial values, unknown initial defs and conditions on unknown initial values.

4.5.2.1 Conditional contexts

Like *points-tos*, the def-use associations are also calculated dependent on conditions on (1) the aliasing between unknown initial values of parameters and globals, and (2) the concrete types of these unknown initial values. For example, in Figure 1.5, the def-use association between the statements **4** and **6** due the definition of $a2_{init}.next$ at statement **4** and later its use at statement **6** is computed dependent on the condition that $a1_{init}$ and $a2_{init}$ are the same object at the entry of *method1*. Similarly, the def-use association between the statements **1** and **7** due to the definition and use of *global1* is computed dependent on the condition that the concrete type of $a1_{init} \in \{A,B\}$, and the def-use association between the statements **2** and **7** due to the definition and use of *global1* is computed dependent on the condition that the concrete type of $a1_{init} \in \{C\}$.

4.5.2.2 Unknown initial def

When a global or a field of an unknown initial value is read at a statement in a method and there exists a definition clear path from the entry of the method to the statement, the definition point of the corresponding def-use association at the statement is represented by the unknown initial def of the global or the field of the unknown initial value at the method entry. For example, in Figure 1.5, the initial value of $a2_{init}.next$ is read at statement **6** if *a1* and *a2* are not aliased at the entry of *method1*, and hence the def-use association $[\langle(a1_{init}\ \mathbf{neq}\ a2_{init}),\ empty,\ empty\rangle,\langle a2_{init}.next, a2_{init}.next_{initdef}, 6\rangle]$ (*DUA-dfelm*-**c**) is computed at program point **6**. The definition point of this def-use association is parametrized by the unknown initial def of $a2_{init}.next$, i.e., $a2_{init}.next_{initdef}$.

4.5.2.3 Examples of propagation of *DUA-dfelms*

The Phase I-dua solution for *method1* defined in Figure 1.5 is given in Figures 4.2 and 4.3. The solution at each program point is shown in two parts: the first part shows the *DUA-dfelms* that represent reaching definitions and the second part shows the *DUA-dfelms* that represent def-use associations. The *DUA-dfelms* that represent the reaching definitions are computed at the top of a statement, while the def-use

Solution at program point **3**:
1. Defs reaching program point **3**:
{ 3:1 [*empty*,*Empty-context*,$\langle$*a1,method1-entry*,$\langle$*Empty-context*,$a1_{init}\rangle\rangle$],
3:2 [*empty*,*Empty-context*,$\langle$*a2,method1-entry*,$\langle$*Empty-context*,$a2_{init}\rangle\rangle$],
3:3 [*empty*,*Empty-context*,$\langle$*a3,method1-entry*,$\langle$*Empty-context*,$a3_{init}\rangle\rangle$],
3:4 [*empty*,*Empty-context*,$\langle a2_{init}.next, a2_{init}.next_{initdef}, \langle$*Empty-context*,$a2_{init}.next_{init}\rangle\rangle$],
3:5 [*empty*,*Empty-context*,$\langle$*global1*,$global1_{initdef}$,$\langle$*Empty-context*,$global1_{init}\rangle\rangle$],
3:6 [*empty*,*Empty-context*,$\langle$*global2*,$global2_{initdef}$,$\langle$*Empty-context*,$global2_{init}\rangle\rangle$],
3:7 [*empty*,*Empty-context*,$\langle$*global3*,$global3_{initdef}$,$\langle$*Empty-context*,$global3_{init}\rangle\rangle$] **}**
2. Def-use associations:
{ 3:8 [*Empty-context*,$\langle$*a1,method1-entry,3*$\rangle$] **}**

Solution at program point **4**:
1. Defs reaching program point **4**:
{ 4:1 [*empty*,*Empty-context*,$\langle$*r,3*,$\langle$*Empty-context*,$a1_{init}\rangle\rangle$] **}**
$\cup$ **{ 3:1** to **3:7 }**
2. Def-use associations:
{ 4:2 [*Empty-context*,$\langle$*a1,method1-entry,4*$\rangle$],
4:3 [*Empty-context*,$\langle$*a3,method1-entry,4*$\rangle$] **}**

Solution at program point **5**:
1. Defs reaching program point **5**:
{ 5:1 [*empty*,*Empty-context*,$\langle a1_{init}.next, 4, \langle$*Empty-context*,$a3_{init}\rangle\rangle$],
5:2 [*empty*,$\langle$($a1_{init}$ **eq** $a2_{init}$), *empty, empty*$\rangle$,$\langle a2_{init}.next, 4, \langle$*Empty-context*,$a3_{init}\rangle\rangle$],
5:3 [*empty*,$\langle$($a1_{init}$ **neq** $a2_{init}$), *empty, empty*$\rangle$,
$\langle a2_{init}.next,\ a2_{init}.next_{initdef},\ \langle$*Empty-context*,$a2_{init}.next_{init}\rangle\rangle$]
}
$\cup$ **{ 3:1** to **3:3**, **3:5** to **3:7**, **4:1 }**
2. Def-use associations:
{ 5:4 [*Empty-context*,$\langle$*r,3,5*$\rangle$],
5:5 [$\langle$*empty*,(*type*($a1_{init}$)$\in$*A::update*),*empty*$\rangle$,$\langle$*r,3*,$\langle$*5,A::update*$\rangle\rangle$],
5:6 [$\langle$*empty*,(*type*($a1_{init}$)$\in$*C::update*),*empty*$\rangle$,$\langle$*r,3*,$\langle$*5,C ::update*$\rangle\rangle$] **}**

Figure 4.2: Phase I-dua solution for method1

Solution at program point **6**:
1. Defs reaching program point **6**:
{ 6:1 [*empty*,⟨*empty,(type*($a1_{init}$)∈*A::update),empty*⟩,⟨*global1,1,*⟨*Empty-context,object*$_1$⟩⟩],
6:2 [*empty*,⟨*empty,(type*($a1_{init}$)∈*A::update),empty*⟩,⟨*global3,1a,*⟨*Empty-context,object*$_1$⟩⟩],
6:3 [*empty*,⟨*empty,(type*($a1_{init}$)∈*C::update),empty*⟩,⟨*global1,2,*⟨*Empty-context,object*$_2$⟩⟩],
6:4 [*empty*,⟨*empty,(type*($a1_{init}$)∈*C::update),empty*⟩,⟨*global3,2a,*⟨*Empty-context,object*$_2$⟩⟩] **}**
∪ **{ 3:1** to **3:3**, **3:6**, **4:1**, **5:1** to **5:3 }**
2. Def-use associations:
{ 6:5 [*Empty-context*,⟨*a2,method1-entry,6*⟩]
6:6 [⟨($a1_{init}$ **eq** $a2_{init}$), *empty, empty*⟩,⟨$a2_{init}$*.next,4,6*⟩],
6:7 [⟨($a1_{init}$ **neq** $a2_{init}$), *empty, empty*⟩,⟨$a2_{init}$*.next*,$a2_{init}$*.next*$_{initdef}$*,6*⟩] **}**

Solution at program point **7**:
1. Defs reaching program point **7**:
{ 7:1 [*empty,Empty-context*,⟨*global2,6,*⟨⟨($a1_{init}$ **eq** $a2_{init}$), *empty, empty*⟩,$a3_{init}$⟩⟩],
7:2 [*empty,Empty-context*,⟨*global2,6,*⟨⟨($a1_{init}$ **neq** $a2_{init}$), *empty, empty*⟩,$a2_{init}$*.next*$_{init}$⟩⟩] **}**
∪ **{ 3:1** to **3:3**, **4:1**, **5:1** to **5:3**, **6:1** to **6:4 }**
2. Def-use associations:
{ 7:3 [⟨*empty,(type*($a1_{init}$)∈*A::update),empty*⟩,⟨*global1,1,7*⟩],
7:4 [⟨*empty,(type*($a1_{init}$)∈*C::update),empty*⟩,⟨*global1,2,7*⟩] **}**

Solution at program point **8**:
1. Defs reaching program point **8**:
{ 8:1 [*empty,Empty-context*,⟨*u,7,*⟨⟨*empty,(type*($a1_{init}$)∈*A::update),empty*⟩,*object*$_1$⟩⟩],
8:2 [*empty,Empty-context*,⟨*u,7,*⟨⟨*empty,(type*($a1_{init}$)∈*C::update),empty*⟩,*object*$_2$⟩⟩] **}**
∪ **{ 3:1** to **3:3**, **4:1**, **5:1** to **5:3**, **6:1** to **6:4**, **7:1**, **7:2 }**
2. Def-use associations:
ϕ

Solution at program point **8a**:
1. Defs reaching program point **8a**:
{ 8a:1 [*empty,Empty-context*,⟨*global1,8,*⟨*Empty-context,object*$_8$⟩⟩] **}**
∪ **{ 3:1** to **3:3**, **4:1**, **5:1** to **5:3**, **6:2**, **6:4**, **7:1**, **7:2**, **8:1**, **8:2 }**
2. Def-use associations:
ϕ

Figure 4.3: Phase I-dua solution for method1

associations computed at a statement represent the def-use associations for the uses at the statement.

Consider the defs reaching program point **3**. These represent (1) the initial definitions of the parameters that are read in *method1*, (2) the initial definitions of the globals that are in the *used-set* of *method1* and (3) the initial definitions of the fields of unknown initial values that are read in *method1* or any method invoked during the lifetime of *method1*. Here *method1-entry* is the number of the entry node of *method1*. The only def-use association computed at program point **3** represents the definition of the parameter *a1* at the entry node of *method1* and later its use at program point **3**.

Next consider the solution at program point **4**. Statement **3** is a definition point for the variable *r*. As a result, we have the *DUA-dfelm*-**a 4:1** at program point **4**, which represents the definition of *r* at program point **3**. The *DUA-dfelms*-**c 4:2** and **4:3** respectively represent the def-use associations due to the definitions of *a1* and *a3* at the entry node of *method1* and their uses at program point **4**.

Now consider the solution at program point **5**. Statement **4** updates the *next* field of $a1_{init}$ in all contexts. As a result, we have the *DUA-dfelm*-**a 5:1** which represents the definition of $a1_{init}.next$ at program point **4**. As stated earlier, statement **4** also updates $a2_{init}.next$ if *a1* and *a2* are aliased at the entry of *method1*. As a result, we have the *DUA-dfelm*-**a 5:2** which represents the potential definition of $a2_{init}.next$ at program point **4**. Similarly **4** does not update $a2_{init}.next$ if *a1* and *a2* are not aliased at the entry of *method1*. As a result, we have the *DUA-dfelm*-**a 5:3** which says that the unknown initial def of $a2_{init}.next$ (i.e., $a2_{init}.next_{initdef}$) propagates across program point **4** if *a1* and *a2* are not aliased at the entry of *method1*. Since the value of *r* is used as the receiver at program point **5**, there is a def-use association between program points **3** and **5** due to the definition and use of *r*, this is represented by the *DUA-dfelm*-**c 5:4**. As mentioned earlier (see Section 4.1), there are *p-uses* of *r* along the interprocedural edges between program point **5** and the methods *A::update* and *C::update*. The *DUA-dfelms*-**e 5:5** and **5:6** represent these *p-uses*. The *value* field of **4:1** (i.e., $a1_{init}$) is used to compute the type contexts of **5:5** and **5:6**. This is the reason why the *value* field is propagated in a *DUA-dfelm*-**a**. The *value* field gives the receiver object and hence the type contexts under which different targets will be invoked at a dynamically dispatched call site can be computed using the *value* field.

Next consider the solution at program point **6**. If the concrete type of $a1_{init}$, the receiver at program point **5**, belongs to { A, B }, the definitions of *global1* and *global3* at program points **1** and **1a** respectively, are propagated to program point **6**. The *DUA-dfelms*-**a** **6:1** and **6:2** represent these definitions of *global1* and *global3* respectively. On the other hand, if the concrete type of $a1_{init}$, the receiver at program point **5**, belongs to { C }, the definitions of *global1* and *global3* at program points **2** and **2a** respectively, are propagated to program point **6**. The *DUA-dfelms*-**a** **6:3** and **6:4** represent these definitions of *global1* and *global3* respectively. *a2* and $a2_{init}.next$ are read at program point **6**. As a result, the *DUA-dfelms*-**c** **6:5**, **6:6** and **6:7** are computed using the *DUA-dfelms*-**a** **3:2**, **5:2** and **5:3** respectively, in an obvious manner.

Now consider the solution at program point **7**. Statement **6** updates *global2*. If *a1* and *a2* are aliased at the entry of *method1*, the value of $a2_{init}.next$ at program point **6** is $a3_{init}$, as $a2_{init}.next$ is updated at program point **4** in this alias context. As a result, we have the *DUA-dfelm*-**a** **7:1**. If *a1* and *a2* are not aliased at the entry of *method1*, the value of $a2_{init}.next$ at program point **6** is $a2_{init}.next_{init}$, as $a2_{init}.next$ is not updated at program point **4** in this alias context. As a result, we have the *DUA-dfelm*-**a** **7:2**. *global1* is read at program point **7**. As a result, we have the *DUA-dfelm*-**c** **7:3** and **7:4** which represent the def-use associations between program points **1** and **2** and program point **7** due to the definition and use of *global1* and which are computed using **6:1** and **6:3** respectively.

The reasoning behind the solutions at program points **8** and **8a** is similar to the reasoning given above.

The defs reaching program point **8a** except **3:1**, **3:2**, **3:3**, **4:1**, **8:1** and **8:2** (as these represent defs of variables local to *method1*) comprise the def-use summary transfer function of *method1*.

Now consider the call sites **9** and **12** that invoke *method1* from *method2*. At call site 9,

$[empty, Empty\text{-}context, \langle global1, 8, \langle Empty\text{-}context, object_8\rangle\rangle]$
$[empty, Empty\text{-}context, \langle a4_{init}.next, 4, \langle Empty\text{-}context, a6_{init}\rangle\rangle]$
$[empty, \langle (a4_{init}\ \mathbf{eq}\ a5_{init}), empty, empty\rangle, \langle a5_{init}.next, 4, \langle Empty\text{-}context, a6_{init}\rangle\rangle]$
$[empty, \langle (a4_{init}\ \mathbf{neq}\ a5_{init}), empty, empty\rangle,$
$\quad \langle a5_{init}.next,\ a5_{init}.next_{initdef},\ \langle Empty\text{-}context,\ a5_{init}.next_{init}\ \rangle\rangle]$
$[empty, \langle empty, (type(a4_{init}) \in A::update), empty\rangle, \langle global3, 1a, \langle Empty\text{-}context, object_1\rangle\rangle]$
$[empty, \langle empty, (type(a4_{init}) \in C::update), empty\rangle, \langle global3, 2a, \langle Empty\text{-}context, object_2\rangle\rangle]$
$[empty, Empty\text{-}context, \langle global2, 6, \langle\langle (a4_{init}\ \mathbf{eq}\ a5_{init}), empty, empty\rangle, a6_{init}\rangle\rangle]$
$[empty, Empty\text{-}context, \langle global2, 6, \langle\langle (a4_{init}\ \mathbf{neq}\ a5_{init}), empty, empty\rangle, a5_{init}.next_{init}\rangle\rangle]$

Figure 4.4: Instantiated *DUA-dfelms* at call site **9**

Def-Use-Algo translates the *DUA-dfelms* at the exit node of *method1* by replacing the unknown initial values of *a1*, *a2* and *a3* with their values at call site 9 (i.e., the unknown initial values of *a4*, *a5* and *a6* respectively) and by replacing the unknown initial def of $a2_{init}.next$ with its value at call site 9 (i.e., $a5_{init}.next_{initdef}$). For example:

- **5:3** $[empty,\ \langle (a1_{init}\ \mathbf{neq}\ a2_{init}), empty, empty\rangle,$
  $\langle a2_{init}.next,\ a2_{init}.next_{initdef},\ \langle Empty\text{-}context,\ a2_{init}.next_{init}\ \rangle\rangle]$
  is translated to
  $\langle empty,,\ \langle (a4_{init}\ \mathbf{neq}\ a5_{init}), empty, empty\rangle,$
  $\langle a5_{init}.next,\ a5_{init}.next_{initdef},\ \langle Empty\text{-}context,\ a5_{init}.next_{init}\ \rangle\rangle]$

- **7:1** $[empty, Empty\text{-}context, \langle global2, 6, \langle\langle (a1_{init}\ \mathbf{eq}\ a2_{init}), empty, empty\rangle, a3_{init}\rangle\rangle]$
  is translated to
  $[empty, Empty\text{-}context, \langle global2, 6, \langle\langle (a4_{init}\ \mathbf{eq}\ a5_{init}), empty, empty\rangle, a6_{init}\rangle\rangle]$

- **6:4** $[empty, \langle empty, (type(a1_{init}) \in C::update), empty\rangle, \langle global3, 2a, \langle Empty\text{-}context, object_2\rangle\rangle]$
  is translated to
  $[empty, \langle empty, (type(a4_{init}) \in C::update), empty\rangle, \langle global3, 2a, \langle Empty\text{-}context, object_2\rangle\rangle]$

Figure 4.4 shows all the resulting *DUA-dfelms* when the unknown initial values and unknown initial defs in the *DUA-dfelms* contained in the def-use summary transfer function of *method1* are instantiated by their values at call site **9**.

[*empty*,*Empty-context*,⟨*global1*,*8*,⟨*Empty-context*,$object_8$⟩⟩]
[*empty*,*Empty-context*,⟨$object_{10}.next$,*4*,⟨*Empty-context*,$object_{11}$⟩⟩]
[*empty*,*Empty-context*,⟨$object_{11}.next$,*11*,⟨*Empty-context*,*null*⟩⟩]
[*empty*,*Empty-context*,⟨*global3*,*1a*,⟨*Empty-context*,$object_1$⟩⟩]
[*empty*,*Empty-context*,⟨*global2*,*6*,⟨*Empty-context*,*null*⟩⟩]

Figure 4.5: Instantiated *DUA-dfelms* at call site **12**

At call site **12**, Def-Use-Algo instantiates the *DUA-dfelms* in the def-use summary transfer function of *method1* using $object_{10}$ as $a1_{init}$, $object_{11}$ as $a2_{init}$ and $a3_{init}$, and *11* as $a2_{init}.next_{initdef}$. For example:

- **7:1** [*empty*,*Empty-context*,⟨*global2*,*6*,⟨⟨($a1_{init}$ **eq** $a2_{init}$),*empty*,*empty*⟩,$a3_{init}$⟩⟩]
  is not applicable to this call site as ($object_{10}$ **eq** $object_{11}$) is false.

- **7:2** [*empty*,*Empty-context*,⟨*global2*,*6*,⟨⟨($a1_{init}$ **neq** $a2_{init}$),*empty*,*empty*⟩,$a2_{init}.next_{init}$⟩⟩]
  is translated to
  [*empty*,*Empty-context*,⟨*global2*,*6*,⟨*Empty-context*,*null*⟩⟩]
  because ($object_{10}$ **neq** $object_{11}$) is *true* and the value of $object_{11}.next$ is *null* at program point **12**. Recall that the value of $object_{11}.next$ is *null* at program point **12** because all pointer fields of a newly allocated object are initialized with *null* (as in Java). Thus, Phase I-pta computes a *null* value for $object_{11}.next$ at program point **12** (see Section 2.4.2.5).

- **6:2** [*empty*,⟨*empty*,(*type*($a1_{init}$)∈*A::update*),*empty*⟩,⟨*global3*,*1a*,⟨*Empty-context*,$object_1$⟩⟩]
  is translated to
  [*empty*,*Empty-context*,⟨*global3*,*1a*,⟨*Empty-context*,$object_1$⟩⟩]
  because B, the concrete type of $a1_{init}$, belongs to *A::update*, i.e., {A,B}.

Figure 4.5 shows all the resulting *DUA-dfelms* when the unknown initial values and unknown initial defs in the *DUA-dfelms* contained in the def-use summary transfer function of *method1* are instantiated by their values at call site **12**.

In addition to the actual-to-uiv bindings mentioned in Chapter 2, at each call site Def-Use-Algo stores the bindings between the actuals and the unknown initial defs at the entry of the methods invocable from the call site. Each def-to-uidef (or actual-to-uidef) binding has the form [*ecfi*,*rc*,⟨⟨*actual-loc,actual-def*⟩,*uidef-at-the-entry-of-a-target*⟩], where *ecfi* and *rc* are respectively an *ECFInfo* and relevant context under which the binding holds. *ecfi* **will be non-empty only at a call site directly contained in a finally**. *actual-loc* is a user-defined variable, a field of a heap-name or a field of an unknown initial value. *actual-def* is a program point (i.e., a definition point) or an unknown initial def. The reason why *actual-loc* is also stored in the binding is to increase precision in those cases where the reaching definition is that of a field of a heap-name or an unknown initial value. This ensures that the different objects and defs reaching the entry node of a method along different paths are not mixed. This is just a technique for maintaining more context and its usefulness will be become clear later in Sections 4.5.4 and 4.8. It can be ignored for the time being. For example, consider the call site **12** in Figure 1.5. The def-to-uidef binding [*empty*,*Empty-context*,⟨⟨$object_{11}.next$,*11*⟩,$a2_{init}.next_{initdef}$⟩] is computed at this call site. When an def-to-uidef binding is used for instantiating an unknown initial def with an actual, the relevant context of the binding becomes part of the relevant context of the *DUA-dfelm* resulting from the instantiation.

4.5.2.4 Examples of propagation of *DUA-dfelms* in the presence of exceptions

The Phase I-dua solution at a few program points in *method3* defined in Figure 1.5 is given in Figures 4.6 and 4.7. Again, the solution at each program point is shown in two parts: the first part shows the *DUA-dfelms* that represent reaching definitions and the second part shows the *DUA-dfelms* that represent def-use associations.

First consider the solution at program point **14**. The *DUA-dfelm*-**a 14:1** represents the initial definition of the parameter *e1* at the entry node of *method3* (i.e., *method3-entry*). The *DUA-dfelm*-**a 14:2** represents the unknown initial def of *global3* propagated to program point **14** from the entry node of *method3*.

Next consider the solution at program point **15**. The *DUA-dfelm*-**a 15:1** represents

Solution at program point **14**:
1. Defs reaching program point **14**:
{ 14:1 [*empty*,*Empty-context*,$\langle$*e1*,*method3-entry*,$\langle$*Empty-context*,$e1_{init}\rangle\rangle$],
14:2 [*empty*,*Empty-context*,$\langle$*global3*,$global3_{initdef}$,$\langle$*Empty-context*,$global3_{init}\rangle\rangle$] **}**
2. Def-use associations:
ϕ

Solution at program point **15**:
1. Defs reaching program point **15**:
{ 15:1 [*empty*,*Empty-context*,$\langle$*global3*,*14*,$\langle$*Empty-context*,$object_{14}\rangle\rangle$] **}**
$\cup$ **{ 14:1 }**
2. Def-use associations:
{ 15:2 [*Empty-context*,$\langle$*e1*,*method3-entry*,*15*$\rangle$] **}**

Solution at program point **16**:
1. Defs reaching program point **16**:
{ 16:1 [$e1_{init}$,*Empty-context*,$\langle$*global3*,*14*,$\langle$*Empty-context*,$object_{14}\rangle\rangle$],
16:2 [$e1_{init}$,*Empty-context*,$\langle$*e1*,*method3-entry*,$\langle$*Empty-context*,$e1_{init}\rangle\rangle$],
16:3 [**excp-def,***Empty-context*,$e1_{init}$,*15*] **}**
2. Def-use associations:
ϕ

Figure 4.6: Phase I-dua solution at a few program points in method3

Solution at program point **17**:
1. Defs reaching program point **17**:
{ 16:1 to **16:3 }**
2. Def-use associations:
{ 17:1 $[\langle \mathit{empty,empty,(type(}e1_{init}) \leq ET2)\rangle, \langle e1_{init}, 15, 17\rangle]$ **}**

Solution at program point **18**:
1. Defs reaching program point **18**:
{ 18:1 $[\mathit{empty}, \langle \mathit{empty,empty,(type(}e1_{init}) \leq ET2)\rangle, \langle \mathit{excp}, 17, \langle \mathit{Empty\text{-}context}, e1_{init}\rangle\rangle]$,
18:2 $[\mathit{empty}, \langle \mathit{empty,empty,(type(}e1_{init}) \leq ET2)\rangle, \langle \mathit{global3}, 14, \langle \mathit{Empty\text{-}context}, \mathit{object}_{14}\rangle\rangle]$,
18:3
$[\mathit{empty}, \langle \mathit{empty,empty,(type(}e1_{init}) \leq ET2)\rangle, \langle \mathit{e1}, \mathit{method3\text{-}entry}, \langle \mathit{Empty\text{-}context}, e1_{init}\rangle\rangle]$
}
2. Def-use associations:
{ 18:4 $[\langle \mathit{empty,empty,(type(}e1_{init}) \leq ET2)\rangle, \langle \mathit{global3}, 14, 18\rangle]$ **}**

Solution at program point **18a**:
1. Defs reaching program point **18a**:
{ 18a:1 $[\mathit{empty}, \mathit{Empty\text{-}context}, \langle \mathit{u1}, 18, \langle\langle \mathit{empty,empty,(type(}e1_{init}) \leq ET2)\rangle, \mathit{object}_{14}\rangle\rangle]$ **}**
$\cup$ **{ 18:1** to **18:3 }**
2. Def-use associations:
ϕ

Solution at program point **19**:
1. Defs reaching program point **19**:
{ 19:1 $[e1_{init}, \langle \mathit{empty,empty,(type(}e1_{init}) \not\leq ET2)\rangle, \langle \mathit{global3}, 14, \langle \mathit{Empty\text{-}context}, \mathit{object}_{14}\rangle\rangle]$,
19:2 $[e1_{init}, \langle \mathit{empty,empty,(type(}e1_{init}) \not\leq ET2)\rangle, \langle \mathit{e1}, \mathit{method\text{-}entry}, \langle \mathit{Empty\text{-}context}, e1_{init}\rangle\rangle]$,
19:3 [**excp-def,**$\langle \mathit{empty,empty,(type(}e1_{init}) \not\leq ET2)\rangle, e1_{init}, 15]$ **}**
$\cup$ **{ 18:2**, **18:3**, **18a:1 }**
2. Def-use associations:
ϕ

Figure 4.7: Phase I-dua solution at a few program points in method3

the definition of *global3* at program point **14**. Since the *DUA-dfelm*-**a** **14:2** is killed by program point **14**, **14:2** is not propagated to program point **15**. The statement **15** throws the object to which *e1* points, as a result, *e1* is read at program point **15**. Hence, we have **15:2** which represents the def-use relationship between the entry node of *method3* and program point **15** due to the definition and use of *e1*.

Now consider the solution at program point **16**, the exit node of the try statement. The statement **15** throws the unknown initial value $e1_{init}$, as a result, **16:1** and **16:2** are respectively generated from **15:1** and **14:1** at program point **15** and propagated to the exit node of *innermost-enclosing-exception-block(15)*, i.e., program point **16**. The *ECFInfo*'s of **16:1** and **16:2** store the signature[1] of the exception thrown at program point **15**. The *DUA-dfelm*-**b** **16:3** is also generated at program point **15** to represent the exception object thrown at program point **15**, and propagated to program point **16**.

Next consider the solution at program point **17**. If the concrete type of $e1_{init}$ is *ET2* or a subtype of *ET2*, then the exception thrown at program point **15** is caught by the catch statement at program point **17**. As a result, **16:1** to **16:3** are propagated to program point **17**. At program point **17**, **17:1** is generated from **16:3** to record the def-use association between the throw at program point **15** and the catch statement at program point **17** because the exception thrown at program point **15** can be caught by the catch statement at program point **17**. The *exception type context* of **17:1** shows the contexts under which the exception object $e1_{init}$ is caught by the catch statement.

Next consider the solution at program point **18**. **18:1** is generated from **16:3** to represent the definition of the parameter of the catch statement (i.e., *excp*) at program point **17** because the exception object caught by the catch statement is assigned to the parameter of the catch statement. Again, the *exception type context* of **18:1** shows the contexts under which the exception object $e1_{init}$ is caught by the catch statement. **18:2** and **18:3** are respectively generated from **16:1** and **16:2**. The *ECFInfo*'s of **18:2**

[1]Here they store the unknown initial value representing the exception object and not the type of the unknown initial value because the concrete type of the exception could be a subtype of the declared type of the unknown initial value (see Section 2.3.1).

and **18:3** are *empty* because the exception has been caught by the catch statement. Again, the *exception type contexts* of **18:2** and **18:3** show the contexts under which the exception object $e1_{init}$ is caught by the catch statement. *global3* is read at program point **18**. As a result, **18:4** is computed from **18:2** to record the def-use association between program points **14** and **18** due the definition and use of *global3*.

Now consider the solution at program point **18a**. The *DUA-dfelm*-**a** **18a:1** represents the definition of *u1* at program point **18**. The relevant context of the *value* field (i.e., *relevant context-2* in Figure 4.1) of **18a:1** shows the *exception type context* under which *global3* points to $object_{14}$ at program point **18**.

Finally consider the solution at program point **19**. The exception thrown at program point **15** (i.e., $e1_{init}$) is not caught by the catch statement at program point **17** if the concrete type of $e1_{init}$ is neither *ET2* nor a subtype of *ET2*. As a result, program point **16** generates **19:1** to **19:3** respectively from **16:1** to **16:3**, and propagates **19:1** to **19:3** to program point **19**. The *exception type contexts* of **19:1** to **19:3** show the contexts in which the exception object $e1_{init}$ is not caught by the catch statement at program point **17**. **18:2**, **18:3** and **18a:1** are propagated to program point **19** from the exit node of the catch statement at program point **17** in an obvious manner.

Now consider the Phase I-dua solution for *method4*. Figures 4.8 and 4.9 show the Phase I-dua solution at a few program points in *method4*.

First consider the solution at program point **21**. The *DUA-dfelms*-**a** **21:1** and **21:3** respectively represent the definitions of *global3* and *e2* reaching program point **21** from the entry node of *method4*. The *DUA-dfelm*-**a** **21:2** results from the assignment to *e3* at program point **20**. There is a def-use association between program points **20** and **21** due to the definition and use of *e3*. This def-use association is computed using **21:2** and it is represented by **21:4**.

Next consider the solution at program point **21a**. The value of $e1_{init}$ at call site **21** is $object_{20}$, whose type is *ET4*. As a result, **19:1** is instantiated into **21a:1**. The relevant context of **21a:1** is *Empty-context* because *(type(*$object_{20}$*)* $\not\leq$ *ET2)* is true and thus the relevant context of **19:1** evaluates to true after the instantiation. **21a:6** is generated

1. Defs reaching program point **21**:
{ 21:1 [*empty*,*Empty-context*,$\langle$*global3*,$global3_{initdef}$,$\langle$*Empty-context*,$global3_{init}\rangle\rangle$],
21:2 [*empty*,*Empty-context*,$\langle$*e3*,*20*,$\langle$*Empty-context*,$object_{20}\rangle\rangle$],
21:3 [*empty*,*Empty-context*,$\langle$*e2*,*method4-entry*,$\langle$*Empty-context*,$e2_{init}\rangle\rangle$] **}**
2. Def-use associations:
{ 21:4 [*Empty-context*,$\langle$*e3*,*20*,*21*$\rangle$] **}**

1. Defs reaching program point **21a**:
{ 21a:1 [*ET4*,*Empty-context*,$\langle$*global3*,*14*,$\langle$*Empty-context*,$object_{14}\rangle\rangle$],
21a:2 [*empty*,*Empty-context*,$\langle$*e3*,*20*,$\langle$*Empty-context*,$object_{20}\rangle\rangle$],
21a:3 [*ET4*,*Empty-context*,$\langle$*e3*,*20*,$\langle$*Empty-context*,$object_{20}\rangle\rangle$],
21a:4 [*empty*,*Empty-context*,$\langle$*e2*,*method4-entry*,$\langle$*Empty-context*,$e2_{init}\rangle\rangle$],
21a:5 [*ET4*,*Empty-context*,$\langle$*e2*,*method4-entry*,$\langle$*Empty-context*,$e2_{init}\rangle\rangle$],
21a:6 [excp-def,*Empty-context*,$object_{20}$,*15*] **}**
2. Def-use associations:
ϕ

1. Defs reaching program point **23**:
{ 23:1 [*empty*,*Empty-context*,$\langle$*global3*,*14*,$\langle$*Empty-context*,$object_{14}\rangle\rangle$],
23:2 [*empty*,*Empty-context*,$\langle$*e3*,*20*,$\langle$*Empty-context*,$object_{20}\rangle\rangle$],
23:3 [*empty*,*Empty-context*,$\langle$*e2*,*method4-entry*,$\langle$*Empty-context*,$e2_{init}\rangle\rangle$],
23:4 [*empty*,*Empty-context*,$\langle$*excp1*,*22*,$\langle$*Empty-context*,$object_{20}\rangle\rangle$] **}**
2. Def-use associations:
{ 23:5 [*Empty-context*,$\langle$*global3*,*14*,*23*$\rangle$] **}**

Figure 4.8: Phase I-dua solution for method4

1. Defs reaching program point **24**:
{ **24:1** [*empty*,*Empty-context*,$\langle$*global3*,$global3_{initdef}$,$\langle$*Empty-context*,$global3_{init}\rangle\rangle$],
24:2 [*empty*,*Empty-context*,$\langle$*e4*,*23b*,$\langle$*Empty-context*,$e2_{init}\rangle\rangle$],
24:3 [*empty*,*Empty-context*,$\langle$*e2*,*method4-entry*,$\langle$*Empty-context*,$e2_{init}\rangle\rangle$] }
2. Def-use associations:
{ **24:4** [*Empty-context*,$\langle$*e4*,*23b*,*24*$\rangle$] }

1. Defs reaching program point **24a**:
{ **24a:1** [$e2_{init}$,$\langle$*empty*,*empty*,(type($e2_{init}$) $\not\leq$ ET2)$\rangle$,$\langle$*global3*,*14*,$\langle$*Empty-context*,$object_{14}\rangle\rangle$],
24a:2 [*empty*,$\langle$*empty*,*empty*,(type($e2_{init}$) $\leq$ ET2)$\rangle$,$\langle$*global3*,*14*,$\langle$*Empty-context*,$object_{14}\rangle\rangle$],
24a:3 [*empty*,*Empty-context*,$\langle$*e4*,*23b*,$\langle$*Empty-context*,$e2_{init}\rangle\rangle$],
24a:4 [$e2_{init}$,$\langle$*empty*,*empty*,(type($e2_{init}$) $\not\leq$ ET2)$\rangle$,$\langle$*e4*,*23b*,$\langle$*Empty-context*,$e2_{init}\rangle\rangle$],
24a:5 [*empty*,*Empty-context*,$\langle$*e2*,*method4-entry*,$\langle$*Empty-context*,$e2_{init}\rangle\rangle$],
24a:6
[$e2_{init}$,$\langle$*empty*,*empty*,(type($e2_{init}$) $\not\leq$ ET2)$\rangle$,$\langle$*e2*,*method4-entry*,$\langle$*Empty-context*,$e2_{init}\rangle\rangle$],
24a:7 [**excp-def**,$\langle$*empty*,*empty*,(type($e2_{init}$) $\not\leq$ ET2)$\rangle$,$e2_{init}$,*15*] }
2. Def-use associations:
ϕ

1. Defs reaching program point **26**:
{ **26:1** [*empty*,$\langle$*empty*,*empty*,(type($e2_{init}$) $\leq$ ET4)$\rangle$,$\langle$*global3*,*14*,$\langle$*Empty-context*,$object_{14}\rangle\rangle$],
26:2 [*empty*,$\langle$*empty*,*empty*,(type($e2_{init}$) $\leq$ ET4)$\rangle$,$\langle$*e4*,*23b*,$\langle$*Empty-context*,$e2_{init}\rangle\rangle$],
26:3
[*empty*,$\langle$*empty*,*empty*,(type($e2_{init}$) $\leq$ ET4)$\rangle$,$\langle$*e2*,*method4-entry*,$\langle$*Empty-context*,$e2_{init}\rangle\rangle$],
26:4 [*empty*,$\langle$*empty*,*empty*,(type($e2_{init}$) $\leq$ ET4)$\rangle$,$\langle$*excp2*,*25*,$\langle$*Empty-context*,$e2_{init}\rangle\rangle$] }
2. Def-use associations:
{ **26:5** [$\langle$*empty*,*empty*,(type($e2_{init}$) $\leq$ ET4)$\rangle$,$\langle$*global3*,*14*,*26*$\rangle$] }

1. Defs reaching program point **28**:
{ **28:1** [*empty*, $\langle$*empty*,*empty*,(type($e2_{init}$) $\not\leq$ ET2)$\wedge$(type($e2_{init}$) $\not\leq$ ET4)$\rangle$,
$\langle$*global3*, *14*, $\langle$*Empty-context*, $object_{14}$ $\rangle\rangle$],
28:2 [*empty*, $\langle$*empty*,*empty*,(type($e2_{init}$) $\not\leq$ ET2)$\wedge$(type($e2_{init}$) $\not\leq$ ET4)$\rangle$,
$\langle$*e4*, *23b*, $\langle$*Empty-context*, $e2_{init}$ $\rangle\rangle$],
28:3 [*empty*,$\langle$*empty*,*empty*,(type($e2_{init}$) $\not\leq$ ET2)$\wedge$(type($e2_{init}$) $\not\leq$ ET4)$\rangle$,
$\langle$*e2*,*method4-entry*,$\langle$*Empty-context*,$e2_{init}\rangle\rangle$],
28:4 *OutLD empty*, $\langle$*empty*,*empty*,(type($e2_{init}$) $\not\leq$ ET2)$\wedge$(type($e2_{init}$) $\not\leq$ ET4)$\rangle$,
$\langle$*excp3*, *27*, $\langle$*Empty-context*, $e2_{init}$ $\rangle\rangle$] }
2. Def-use associations:
{ **28:5** [$\langle$*empty*,*empty*,(type($e2_{init}$) $\not\leq$ ET2)$\wedge$(type($e2_{init}$) $\not\leq$ ET4)$\rangle$,$\langle$*global3*,*14*,*28*$\rangle$] }

Figure 4.9: Phase I-dua solution for method4

from **19:3** in a similar manner. **21a:2** and **21a:4** are respectively generated from **21:2** and **21:3** because there are possible invocation contexts of *method3* in which the exit node of *method3* is reachable from the entry node of *method3* along a path without any uncaught exception. For example, if the concrete type of $e1_{init}$ is *ET2*, then the exception thrown at program point **15** will be caught at program point **17** and the exit node of *method3* will be reached without any uncaught exception. Although this invocation context is not applicable to call site **21**, the decision to retain the *ECFInfo* of a *DUA-dfelm*-**a** representing the definition of a local variable across a call site is made based on whether the exit node of the target is reachable from the entry node of the target along a path without any uncaught exception in any invocation context. This results in a safe solution in all cases and in a precise solution in those cases where the problem is polynomial-time solvable (see Chapter 3). **21a:3** and **21a:5** are respectively generated from **21:2** and **21:3** because *method3* throws $e1_{init}$ as exception object and the concrete type of $e1_{init}$ at call site **21** is *ET4*. The *ECFInfo*'s of **21a:3** and **21a:5** store the signature of the exception thrown by *method3*.

Now consider the solution at program point **23**. The exception thrown by the call at program point **21** is caught by the catch at program point **22**. As a result, **23:1**, **23:2** and **23:3** are respectively generated from **21a:1**, **21a:3** and **21a:5**. The *ECFInfo*'s of **23:1**, **23:2** and **23:3** are *empty* because the exception of concrete type *ET4* has been caught by the catch at program point **22**. **23:4** is generated from **21a:6** because the exception object caught by the catch at program point **22** is assigned to the parameter of the catch (i.e., *excp1*). There is a def-use association between program points **14** and **23** due to the definition and use of *global3*. This def-use association is computed using **23:1** and it is represented by **23:5**.

Next consider the solution at program point **24**. The *DUA-dfelms*-**a** **24:1** and **24:3** respectively represent the definitions of *global3* and *e2* reaching program point **24** from the entry node of *method4*. The *DUA-dfelm*-**a** **24:2** results from the assignment to *e4* at program point **23b**. There is a def-use association between program points **23b** and **24** due to the definition and use of *e4*. This def-use association is computed using **24:2** and it is represented by **24:4**.

Now consider the solution at program point **24a**. The value of $e1_{init}$ at call site **24** is $e2_{init}$. As a result, **19:1**, **18:2** and **19:3** are respectively instantiated into **24a:1**, **24a:2** and **24a:7**. **24a:3** and **24a:5** are respectively generated from **24:2** and **24:3** because (as stated above) there are possible invocation contexts of *method3* in which the exit node of *method3* is reachable from the entry node of *method3* along a path without any uncaught exception. **24a:4** and **24a:6** are respectively generated from **24:2** and **24:3** because *method3* throws $e1_{init}$ as exception object under the context $(type(e1_{init}) \not\leq ET2)$ and the value of $e1_{init}$ at call site **24** is $e2_{init}$. The *ECFInfo*'s of **24a:4** and **24a:6** store the signature of the exception thrown by *method3*.

Next consider the solution at program point **26**. The exception object $e2_{init}$ thrown by the call at program point **24** is caught by the catch at program point **25** under the type constraint $(type(e2_{init}) \not\leq ET2) \wedge (type(e2_{init}) \leq ET4)$. However, $(type(e2_{init}) \not\leq ET2) \wedge (type(e2_{init}) \leq ET4)$ is equivalent to $(type(e2_{init}) \leq ET4)$ because $(type(e2_{init}) \leq ET4)$ implies[2] $(type(e2_{init}) \not\leq ET2)$. As a result, **26:1**, **26:2** and **26:3** are respectively generated from **24a:1**, **24a:4** and **24a:6**. The *ECFInfo*'s of **26:1**, **26:2** and **26:3** are *empty* because the exception has been caught by the catch at program point **25**. **26:4** is generated from **24a:7** because the exception object caught by the catch at program point **25** is assigned to the parameter of the catch (i.e., *excp2*). There is a def-use association between program points **14** and **26** due to the definition and use of *global3*. This def-use association is computed using **26:1** and it is represented by **26:5**.

Finally consider the solution at program point **28**. The exception object $e2_{init}$ thrown by the call at program point **24** is caught by the catch at program point **27** under the type constraint $(type(e2_{init}) \not\leq ET2) \wedge (type(e2_{init}) \not\leq ET4)$ because the exception thrown at program point **15** has to escape the catches at program points **17** and **25** to be caught by the catch at program point **27**. As a result, **28:1**, **28:2** and **28:3** are respectively generated from **24a:1**, **24a:4** and **24a:6**. The *ECFInfo*'s of **28:1**, **28:2** and **28:3** are *empty* because the exception has been caught by the catch at program point

[2] Def-Use-Algo uses the following simple heuristic for simplifying *exception type contexts*. Whenever a new type constraint is added to an *exception type context*, Def-Use-Algo checks if the new type constraint implies any of the existing type constraints. The implied type constraints are dropped.

27. **28:4** is generated from **24a:7** because the exception object caught by the catch at program point **27** is assigned to the parameter of the catch (i.e., *excp3*). There is a def-use association between program points **14** and **28** due to the definition and use of *global3*. This def-use association is computed using **28:1** and it is represented by **28:5**.

4.5.2.5 Examples of propagation in the presence of finally statements

Consider *method5* and *method6* defined in Figure 1.5. First we will recapitulate the overview of control flow through methods *method5* and *method6* (discussed in Chapter 2), and then we will describe the Phase I-dua solutions for these methods.

method6 invokes *method5* from the call site **35**. *method5* allocates an exception object of type *ET1* at program point **31** and then throws this exception object at program point **32**. This exception is not caught in *method5* and it is propagated to *method6*. The catch statement at program point **36** cannot catch this exception as *ET1* is not a subtype of *ET5*. Thus, the control flow goes to the finally statement at program point **37**. The finally statement at program point **37** contains a try statement nested inside it at program point **38**. This nested try statement throws an exception of type *ET1* at program point **40**. Since the catch statement at program point **41** catches all exceptions of type *ET1* or a subtype of *ET1*, the exception thrown at program point **40** is propagated to the catch statement at program point **41**. Thus at program point **41** there are two active exceptions: one thrown at program point **32** and the other thrown at program point **40**. The exception thrown at program point **40** is caught by the catch statement at program point **41**. After this the control flow goes to the entry node of the finally statement at program point **42**. After the exit from the finally statement at program point **42**, the control flow goes to program point **44**, where only the exception thrown at program point **32** is active.

A part of the Phase I-dua solution for *method5* is shown below (we are ignoring a part of the solution for the ease of presentation):

---

private: static void method5(A *param) {

```
ET1 *unexp;
30:  global1 = param;
31:  unexp = new ET1();
/*** 32:1 [empty,Empty-context,⟨global1,30,⟨Empty-context,param_init⟩⟩] ***/
/*** 32:2 [empty,Empty-context,⟨unexp,31,⟨Empty-context,object_31⟩⟩] ***/
/*** 32:3 [Empty-context,⟨unexp,31,32⟩] ***/
32:  throw unexp;
/*** 33:1 [excp-def,Empty-context,object_31,32] ***/
/*** 33:2 [ET1,Empty-context,⟨global1,30,⟨Empty-context,param_init⟩⟩] ***/
/*** 33:3 [ET1,Empty-context,⟨unexp,31,⟨Empty-context,object_31⟩⟩] ***/
33: }
```

There is a def-use association between program points **31** and **32** due to the definition and use of *unexp*. This def-use association is computed using **32:2** and it is represented by **32:3**. The throw statement at program point **32** throws $object_{31}$, whose run-time type is *ET1*. As a result, the *DUA-dfelms*-**a** **33:2** and **33:3** are generated from the *DUA-dfelms*-**a** **32:1** and **32:2** respectively, and the *ECFInfo*'s of **33:2** and **33:3** are *ET1*, the signature the exception thrown at program point **32** (note the difference with the *DUA-dfelms*-**a** **16:1** and **16:2** defined in Figure 4.6). The *DUA-dfelm*-**b** **33:1** represents the exception thrown at program point **32**.

Next consider the part of the Phase I-dua solution for *method6* shown below:

```
public: static void method6( ) {
  A *local;
  ET1 *unexp1;
  34a:  unexp1 = 0;
  34:  local = new A();
```

`try {`

/*** **35:1** $[\mathit{empty},\mathit{Empty\text{-}context},\langle \mathit{local},34,\langle \mathit{Empty\text{-}context},\mathit{object}_{34}\rangle\rangle]$ ***/

/*** **35:2** $[\mathit{empty},\mathit{Empty\text{-}context},\langle \mathit{unexp1},34a,\langle \mathit{Empty\text{-}context},\mathit{null}\rangle\rangle]$ ***/

/*** **35:3** $[\mathit{Empty\text{-}context},\langle \mathit{local},34,35\rangle]$ ***/

35: `method5( local );`

/*** **35a:1** [**excp-def,**$\mathit{Empty\text{-}context},\mathit{object}_{31}$**,**$32$] ***/

/*** **35a:2** $[\mathit{ET1},\mathit{Empty\text{-}context},\langle \mathit{global1},30,\langle \mathit{Empty\text{-}context},\mathit{object}_{34}\rangle\rangle]$ ***/

/*** **35a:3** $[\mathit{ET1},\mathit{Empty\text{-}context},\langle \mathit{local},34,\langle \mathit{Empty\text{-}context},\mathit{object}_{34}\rangle\rangle]$ ***/

/*** **35a:4** $[\mathit{ET1},\mathit{Empty\text{-}context},\langle \mathit{unexp1},34a,\langle \mathit{Empty\text{-}context},\mathit{null}\rangle\rangle]$ ***/

35a: `}`

36: `catch( ET5 *param1 ) {`

`}`

37: `finally {`

/*** **38:1** [**excp-def,**$\mathit{Empty\text{-}context},\mathit{object}_{31}$**,**$32$] ***/

/*** **38:2** $[\mathit{ET1},\mathit{Empty\text{-}context},\langle \mathit{global1},30,\langle \mathit{Empty\text{-}context},\mathit{object}_{34}\rangle\rangle]$ ***/

/*** **38:3** $[\mathit{ET1},\mathit{Empty\text{-}context},\langle \mathit{local},34,\langle \mathit{Empty\text{-}context},\mathit{object}_{34}\rangle\rangle]$ ***/

/*** **38:4** $[\mathit{ET1},\mathit{Empty\text{-}context},\langle \mathit{unexp1},34a,\langle \mathit{Empty\text{-}context},\mathit{null}\rangle\rangle]$ ***/

/*** def-to-uidef-bindings:

38:5 $[\mathit{ET1},\mathit{Empty\text{-}context},\langle\langle \mathit{global1},30\rangle,38{::}\mathit{global}1_{\mathit{initdef}}\rangle]$

38:6 $[\mathit{ET1},\mathit{Empty\text{-}context},\langle\langle \mathit{local},34\rangle,38{::}\mathit{local}_{\mathit{initdef}}\rangle]$

38:7 $[\mathit{ET1},\mathit{Empty\text{-}context},\langle\langle \mathit{unexp1},34a\rangle,38{::}\mathit{unexp}1_{\mathit{initdef}}\rangle]$.

Here 38::$\mathit{global}1_{\mathit{initdef}}$, 38::$\mathit{local}_{\mathit{initdef}}$ and 38::$\mathit{unexp}1_{\mathit{initdef}}$ are respectively the unknown initial defs of global1, local and unexp1 associated with program point **38**. ***/

38: `try {`

/*** **39:1**

$[\mathit{empty},\mathit{Empty\text{-}context},\langle \mathit{global1},38{::}\mathit{global}1_{\mathit{initdef}},\langle \mathit{Empty\text{-}context},38{::}\mathit{global}1_{\mathit{init}}\rangle\rangle]$

***/

/*** **39:2**

$[\mathit{empty},\mathit{Empty\text{-}context},\langle \mathit{local},38{::}\mathit{local}_{\mathit{initdef}},\langle \mathit{Empty\text{-}context},38{::}\mathit{local}_{\mathit{init}}\rangle\rangle]$

***/

/*** **39:3**

$[empty, Empty\text{-}context, \langle unexp1, 38::unexp1_{initdef}, \langle Empty\text{-}context, 38::unexp1_{init} \rangle \rangle]$

***/

39: `unexp1 = new ET1();`

/*** **40:1**

$[empty, Empty\text{-}context, \langle global1, 38::global1_{initdef}, \langle Empty\text{-}context, 38::global1_{init} \rangle \rangle]$

***/

/*** **40:2**

$[empty, Empty\text{-}context, \langle local, 38::local_{initdef}, \langle Empty\text{-}context, 38::local_{init} \rangle \rangle]$

***/

/*** **40:3** $[empty, Empty\text{-}context, \langle unexp1, 39, \langle Empty\text{-}context, object_{39} \rangle \rangle]$ ***/

/*** **40:4** $[Empty\text{-}context, \langle unexp1, 39, 40 \rangle]$ ***/

40: `throw unexp1;`

/*** **40a:1**

$[ET1, Empty\text{-}context, \langle global1, 38::global1_{initdef}, \langle Empty\text{-}context, 38::global1_{init} \rangle \rangle]$

***/

/*** **40a:2**

$[ET1, Empty\text{-}context, \langle local, 38::local_{initdef}, \langle Empty\text{-}context, 38::local_{init} \rangle \rangle]$

***/

/*** **40a:3** $[ET1, Empty\text{-}context, \langle unexp1, 39, \langle Empty\text{-}context, object_{39} \rangle \rangle]$ ***/

/*** **40a:4** $[\textbf{excp-def}, Empty\text{-}context, object_{39}, 40]$ ***/

40a: }

/*** **41:1**

$[ET1, Empty\text{-}context, \langle global1, 38::global1_{initdef}, \langle Empty\text{-}context, 38::global1_{init} \rangle \rangle]$

***/

/*** **41:2**

$[ET1, Empty\text{-}context, \langle local, 38::local_{initdef}, \langle Empty\text{-}context, 38::local_{init} \rangle \rangle]$

***/

/*** **41:3** $[ET1, Empty\text{-}context, \langle unexp1, 39, \langle Empty\text{-}context, object_{39} \rangle \rangle]$ ***/

/*** **41:4** [**excp-def**,$\mathit{Empty\text{-}context}$,$\mathit{object}_{39}$,*40*] ***/

/*** **41:5** [$\mathit{Empty\text{-}context}$,$\langle \mathit{object}_{39}$,*40*,*41*$\rangle$] ***/

41: `catch( ET1 *param2 ) {`

/*** **41a:1**

[*empty*,*Empty-context*,$\langle$*global1*,*38*::$\mathit{global}1_{initdef}$,$\langle$*Empty-context*,*38*::$\mathit{global}1_{init}\rangle\rangle$]

***/

/*** **41a:2**

[*empty*,*Empty-context*,$\langle$*local*,*38*::$\mathit{local}_{initdef}$,$\langle$*Empty-context*,*38*::$\mathit{local}_{init}\rangle\rangle$] ***/

/*** **41a:3** [*empty*,*Empty-context*,$\langle$*unexp1*,*39*,$\langle$*Empty-context*,$\mathit{object}_{39}\rangle\rangle$] ***/

/*** **41a:4** [*empty*,*Empty-context*,$\langle$*param2*,*41*,$\langle$*Empty-context*,$\mathit{object}_{39}\rangle\rangle$] ***/

41a: }

42: `finally {`

/*** **43:1**

[*44*,*Empty-context*,$\langle$*global1*,*38*::$\mathit{global}1_{initdef}$,$\langle$*Empty-context*,*38*::$\mathit{global}1_{init}\rangle\rangle$]

***/

/*** **43:2**

[*44*,*Empty-context*,$\langle$*local*,*38*::$\mathit{local}_{initdef}$,$\langle$*Empty-context*,*38*::$\mathit{local}_{init}\rangle\rangle$]

***/

/*** **43:3** [*44*,*Empty-context*,$\langle$*unexp1*,*39*,$\langle$*Empty-context*,$\mathit{object}_{39}\rangle\rangle$] ***/

43:

}

/*** **44:1** [**excp-def**,*Empty-context*,object_{31},*32*] ***/

/*** **44:2** [*ET1*,*Empty-context*,$\langle$*global1*,*30*,$\langle$*Empty-context*,$\mathit{object}_{34}\rangle\rangle$] ***/

/*** **44:3** [*ET1*,*Empty-context*,$\langle$*local*,*34*,$\langle$*Empty-context*,$\mathit{object}_{34}\rangle\rangle$] ***/

/*** **44:4** [*ET1*,*Empty-context*,$\langle$*unexp1*,*39*,$\langle$*Empty-context*,$\mathit{object}_{39}\rangle\rangle$] ***/

44: `} // End of outer finally`

/*** **45:1** [**excp-def**,*Empty-context*,object_{31},*32*] ***/

/*** **45:2** [*ET1*,*Empty-context*,$\langle$*global1*,*30*,$\langle$*Empty-context*,$\mathit{object}_{34}\rangle\rangle$] ***/

/*** **45:3** [*ET1*,*Empty-context*,$\langle$*local*,*34*,$\langle$*Empty-context*,$\mathit{object}_{34}\rangle\rangle$] ***/

/* 45:4** [$ET1$,*Empty-context*,⟨*unexp1,39,*⟨*Empty-context,object*$_{39}$⟩⟩] *****/**

45: }

There is a def-use association between program points **34** and **35** due to the definition and use of *local*. This def-use association is computed using **35:1** and it is represented by **35:3**. The value of $param_{init}$ at call site **35** is $object_{34}$ because *local* points to $object_{34}$ at call site **35**. As a result, the *DUA-dfelm*-**a 33:2** is instantiated into the *DUA-dfelm*-**a 35a:2**. The *DUA-dfelm*-**a 35a:1** is generated from the *DUA-dfelm*-**a 33:1** is an obvious manner. Since the exit node of *method5* is reachable from the entry node of *method5* only along a path that throws an uncaught exception of run-time type *ET1*, **35:1** and **35:2** are respectively transformed into **35a:3** and **35a:4** as **35:1** and **35:2** are propagated across the call site **35**.

Since the catch statement at program point **36** cannot catch an exception of type *ET1*, **35a:1**, **35a:2**, **35a:3** and **35a:4** are propagated to the entry node of the finally statement at program point **37**, and **38:1**, **38:2**, **38:3** and **38:4** reach program point **38**. As stated earlier, a try statement directly contained in a finally statement is treated like a call to an anonymous procedure because it can cause exceptions (and other reasons for entering a finally statement) to stack up. Recall (see Section 2.4.2) that for doing analysis inside the anonymous procedure, the global variables in the *used-set* of the method containing the try statement and all the local variables of the method containing the try statement are assumed to have unknown initial values at the entry node of the nested try statement. The bindings between the values of the globals and locals and their unknown initial values associated with the entry node of the try statement are stored as actual-to-uiv bindings at the call to the anonymous procedure. Similarly, for doing analysis inside the anonymous procedure, the definitions of global variables in the *used-set* of the method containing the try statement and the definitions of all the local variables of the method containing the try statement, reaching the entry node of the try statement directly contained in the finally statement are represented by unknown initial defs at the entry node of the nested try statement. The bindings between the definitions of globals and locals reaching the entry node of the nested try statement and the unknown initial defs of the globals and locals associated with the entry node of the try statement are stored as def-to-uidef bindings at the call to the anonymous procedure. Thus, the try statement at program point **38** is treated like a call to an anonymous procedure. The def-to-uidef bindings **38:5**, **38:6** and **38:7** store the bindings between the reaching definitions and the unknown initial defs of *global1*, *local* and *unexp1* (i.e., $38::global1_{initdef}$, $38::local_{initdef}$ and $38::unexp1_{initdef}$) at this call site. Recall (see Section 2.4.2) that Phase I-pta computes the following actual-to-uiv bindings at program point **38**:

- [*ET1*,*Empty-context*,$\langle object_{34}$,$38::global1_{init}\rangle$]
- [*ET1*,*Empty-context*,$\langle object_{34}$,$38::local_{init}\rangle$]
- [*ET1*,*Empty-context*,$\langle$*null*,$38::unexp1_{init}\rangle$].

These def-to-uidef bindings (along with actual-to-uiv bindings shown above) are used for instantiating $38::global1_{initdef}$, $38::local_{initdef}$ and $38::unexp1_{initdef}$ in *DUA-dfelms* when these *DUA-dfelms* are propagated from the nested try statement (at program point **38**) into the outer finally statement (that starts at program point **37**). In this way the stack of exceptions (and other reasons for entering a finally statement) is maintained implicitly by remembering them in the corresponding actual-to-uiv and def-to-uidef bindings. For example, at program point **40a** there are two active exceptions: one thrown at program point **32** and the other thrown at program point **40**. However, the *ECFInfo*'s of the *DUA-dfelms* at program point **40a** need to store only the signature of the exception thrown at program point **40** as the signature of the other exception is stored in the actual-to-uiv and def-to-uidef bindings at program point **38**.

The exception thrown at program point **40** is caught by the catch statement at program point **41**. As a result, **41a:1** to **41a:4** are generated from **41:1** to **41:4** and **41:5** is generated to represent the def-use between the throw at program point **40** and the catch statement at program point **41**. **41a:1** to **41a:3** are propagated from the program point

41a to the entry node of the finally statement at program point **42**. As a result, **43:1** to **43:3** are propagated to program point **43**. Note that the *ECFInfo*'s of **43:1** to **43:3** are the label *44* because the finally statement has been entered due to falling through the catch statement at program point **41** and hence the control should go to program point **44** from the exit of the finally statement at program point **42**. At program point **43**, the def-to-uidef bindings **38:5** and **38:6** are used for instantiating $38{::}global1_{initdef}$ and $38{::}local_{initdef}$ in **43:1** and **43:2**. $38{::}global1_{init}$ and $38{::}local_{init}$ in **43:1** and **43:2** are instantiated using the actual-to-uiv bindings stored at program point **38**. As a result, **44:2** and **44:3** are generated from **43:1** and **43:2** respectively. Note that the *ECFInfo*'s of **44:2** and **44:3** are *ET1* because the *ECFInfo*'s of the actual-to-uiv bindings and def-to-uidef bindings used in generating **44:2** and **44:3** are *ET1*. **44:4** is generated from **43:3** because program point **38** is reachable only along a path that has an uncaught exception of run-time type *ET1* (thrown at program point **32**). **44:1** is generated from **38:1** because program point **43** is reachable from program point **38** along a path without any uncaught exception or incomplete, pending transition to a statement outside the outer finally statement.

4.5.3 Phase II

Def-Use-Algo traverses the SCC-DAG in a topological order (top-down) and performs the following two subphases on each SCC.

Phase II-pta: This is same as described in Chapter 2.

Phase II-dua: In this subphase Def-Use-Algo propagates *concrete values* of unknown initial defs to the *entry* nodes of methods. For example, at statement **11** in Figure 1.5, the *next* field of $object_{11}$ is initialized, and hence at call site **12**, Phase II-dua propagates program point **11** (i.e., the definition point) to the entry of *method1* as a concrete value of the unknown initial def of $a2_{init}.next$. Again, Phase II-dua treats an **interface initial def** like a concrete value and propagates an interface initial def to the entry nodes of other methods if, through a def-to-uidef binding at a call site, the interface initial def is the value of an unknown initial def at the entry of a target. For example, at call site **9** in Figure 1.5, Phase II-dua propagates the interface initial def $a5_{init}.next_{initdef}$ to the entry of *method1* as a concrete value of the unknown initial def of $a2_{init}.next$.

The propagation within a SCC is done iteratively until a fixed point is reached, while propagation across SCC's is done in a top-down manner without iteration.

Both Phase II-pta and Phase II-dua involve only the entry nodes and call nodes.

Figure 4.10 shows the Phase II-dua solution at the entry node of *method1* defined in Figure 1.5. Each cv-to-uidef binding computed at the entry node of a method has the form:

$[rc, \langle loc, def \rangle, uidef]$,

where (1) *uidef* is an unknown initial def at the entry node of the method, (2) *loc* is a user-defined variable or *loc* is a field of a heap-name or *loc* is a field of an interface initial value, (3) *def* is a program point (i.e., a definition point) or *def* is an interface initial def and (4)*rc* is either *Empty-context* or it is a relevant context that involves only the interface initial values of a single *public interface method* and that can evaluate to true by making conservative, worst-case assumptions about the interface initial values, meaning the binding holds at the entry node of the method if *rc* holds at the entry node of the *public interface method*. As stated earlier, the reason why *loc* is also stored in the binding is to increase precision in those cases where the reaching definition is that of a field of a heap-name or an unknown initial value. This ensures that the different objects and defs reaching the entry node of a method along different paths are not mixed. This is just a technique for maintaining more context and its usefulness will be become clear later in Sections 4.5.4 and 4.8. Note that in general to name an unknown initial def uniquely, we need to store in the name of the unknown initial def, the name of the method with whose entry node the unknown initial def is associated. In our earlier examples, for simplicity we did not do this as there was no ambiguity and the method name was clear from context; however, in Figure 4.10 we need to distinguish between the unknown initial defs of the same variable associated with the entry nodes of two different methods. Hence, in Figure 4.10 the unknown initial defs of globals

/*** The cv-to-uidef bindings **method1:1** to **method1:4** are propagated from call site **9**. ***/
/*** Recall that *method2* is a *public interface method* and $a5_{init}.next_{initdef}$ is an interface initial def at the entry of method2. ***/
method1:1 [*Empty-context*,⟨$a5_{init}.next$,$a5_{init}.next_{initdef}$⟩,$a2_{init}.next_{initdef}$]
/*** method2::$global1_{initdef}$ is the interface initial def of global1 at the entry of method2 and method1::$global1_{initdef}$ is the unknown initial def of global1 at the entry of method1. ***/
method1:2 [*Empty-context*,⟨*global1*,*method2::*$global1_{initdef}$⟩,*method1::*$global1_{initdef}$]
method1:3 [*Empty-context*,⟨*global2*,*method2::*$global2_{initdef}$⟩,*method1::*$global2_{initdef}$]
method1:4 [*Empty-context*,⟨*global3*,*method2::*$global3_{initdef}$⟩,*method1::*$global3_{initdef}$]

/*** The cv-to-uidef bindings **method1:5** to **method1:9** are propagated from call site **12**. ***/
method1:5 [*Empty-context*,⟨$object_{11}.next$,*11*⟩,$a2_{init}.next_{initdef}$]
method1:6 [*Empty-context*,⟨*global1*,*8*⟩,*method1::*$global1_{initdef}$]
method1:7 [*Empty-context*,⟨*global2*,*6*⟩,*method1::*$global2_{initdef}$]
method1:8 [⟨*empty*,(*type*($a4_{init}$)∈*A::update*),*empty*⟩,⟨*global3*,*1a*⟩,*method1::*$global3_{initdef}$]
method1:9 [⟨*empty*,(*type*($a4_{init}$)∈*C::update*),*empty*⟩,⟨*global3*,*2a*⟩,*method1::*$global3_{initdef}$]

Figure 4.10: Phase II-dua solution at the entry node of method1

have been named by prefixing the name of the method with whose entry node these unknown initial defs are associated.

4.5.4 Phase III-dua

This phase involves only non-entry nodes. In this phase the unknown initial values and the unknown initial defs in the parametrized def-use solution computed during Phase I-dua 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 def-use solution at each node is expressed entirely in terms of program variables, heap-names, interface initial values, fields of heap-names, fields of interface initial values, definition points, interface initial defs and use points.

In this phase unknown initial values and unknown initial defs in *DUA-dfelms-***c**, *DUA-dfelms-***d** and *DUA-dfelms-***e** are instantiated by their concrete values computed in Phase II. Recall that an interface initial value can be the concrete value of an unknown initial value and an interface initial def can be the concrete value of an unknown initial def. Unknown initial values are instantiated by their concrete values computed in Phase II-pta; unknown initial defs are instantiated by their concrete values computed in Phase II-dua. Whenever a cv-to-uidef binding [*rc1*,⟨*obj,def*⟩,*uidef1*] is used to instantiate an unknown initial def *uidef1* in a *DUA-dfelm-***c** [*rc,* ⟨*var1, uidef1, use-point*⟩] or a *DUA-dfelm-***e** [*rc,* ⟨*var1, uidef1, use-edge*⟩], *uidef1* is replaced by *def* and *var1* is replaced by *obj*. This pairing of *obj* and *def* increases precision by not mixing *def*'s and *obj*'s reaching the corresponding entry node from different paths. Those instantiations of *DUA-dfelms* for which relevant contexts either (1) evaluate to true or (2) are instantiated into relevant contexts involving only the interface initial values of a single *public interface method* and the instantiated relevant contexts can evaluate to true by making conservative, worst-case assumptions about these interface initial values, yield the final def-use solution. Each element of the final def-use solution at a node n has one of the following three forms:

1. [*rc,* ⟨*var, def-point, use-point*⟩], where *def-point* is a program point or an interface initial def, *use-point* is n and *var* is a global, a local, a field of a heap-name or a field of an interface initial value. This is used for def-uses of variables (including flow due to exceptions).

2. [*rc,* ⟨*excp-object, throw-point, catch-point*⟩], where *throw-point* is a throw statement number, *catch-point* is n and *excp-object* is a heap-name or an interface initial value. This is used for def-uses between throws and catches.

3. [*rc,* ⟨*var, def-point, use-edge*⟩], where either n is a condition node and *use-edge* is a decision edge out of n or n is a dynamically dispatched call site and *use-edge* is an interprocedural edge between n and one of its targets. *def-point* and *var* are same as in 1.

In all cases *rc* is either *Empty-context* (meaning the relevant context before the instantiation evaluated to true after the instantiation) or it is a relevant context involving only the interface initial values of a single *public interface method* that can evaluate to true by making conservative, worst-case assumptions about these interface initial values.

Figures 4.11 and 4.12 show the Phase III-dua solution at a few program points in *method1*, which is defined in Figure 1.5. The numbers of data-flow elements in Figures 4.11 and 4.12 respectively correspond to the numbers of data-flow elements in Figures 4.2 and 4.3. For example, **3:8a** is generated from **3:8**.

The set of def-uses computed by Phase III-dua for a library is a superset of the precise set of def-uses for the library; this is expected because computing the precise solution is undecidable [Lan92b, Ram94].

4.5.4.1 Implications for testing

The reason for retaining the relevant contexts in the final def-use solution is that for a library L, the set of relevant contexts of the def-use associations computed during Phase III-dua capture (up to the user specified bounds on the number of conjuncts of different kinds in any relevant context) all the conjunctions of potential aliases, potential non-aliases and type constraints involving interface initial values that are *relevant* for data-flow-based unit testing of L and need to be exercised by test cases. For example, the type constraint *(type(*$a4_{init}$*)*$\in$*A::update)* is relevant for data-flow-based unit testing of the *public interface method method2* of $L_{example}$ because it forms the relevant context of a def-use association (as shown in Figure 4.11) and hence needs to be exercised in a test case; however, the concrete type of $a6_{init}$ is not relevant for data-flow-based unit testing of *method2* because no type constraint involving $a6_{init}$ is part of the relevant context of any def-use association. If only the def-use solution is required then the relevant contexts can be dropped from the final solution.

Solution at program point **3**:
{ **3:8a** [*Empty-context*,⟨*a1,method1-entry,3*⟩] **}**

Solution at program point **4**:
{ **4:2a** [*Empty-context*,⟨*a1,method1-entry,4*⟩],
4:3a [*Empty-context*,⟨*a3,method1-entry,4*⟩] **}**

Solution at program point **5**:
{ **5:4a** [*Empty-context*,⟨*r,3,5*⟩],

/*** **5:5a** is generated from **5:5** by instantiating $a1_{init}$ with the interface initial value $a4_{init}$ as $a4_{init}$ is the concrete value of $a1_{init}$ at call site **9**.
***/
5:5a [⟨*empty,(type(*$a4_{init}$*)∈A::update),empty*⟩,⟨*r,3,*⟨*5,A::update*⟩⟩],

/*** **5:5b** is generated from **5:5** by instantiating $a1_{init}$ with $object_{10}$ as $object_{10}$ is the concrete value of $a1_{init}$ at call site **12**.
Note that *(type(*$object_{10}$*)∈A::update)* evaluates to true as type of $object_{10}$ (i.e., B) $\in \{ A, B\}$ and hence the relevant context of **5:5b** is *Empty-context*.
***/
5:5b [*Empty-context*,⟨*r,3,*⟨*5,A::update*⟩⟩],

/*** **5:6a** is generated from **5:6** by instantiating $a1_{init}$ with the interface initial value $a4_{init}$ as $a4_{init}$ is the concrete value of $a1_{init}$ at call site **9**.
***/

5:6a [⟨*empty,(type(*$a4_{init}$*)∈C::update),empty*⟩,⟨*r,3,*⟨*5,C::update*⟩⟩] **}**

Figure 4.11: Phase III-dua solution for method1

Solution at program point **6**:
{ 6:5a [*Empty-context*,⟨*a2,method1-entry,6*⟩],

/*** **6:6a** is generated from **6:6** because at call site **9**, the interface initial value $a4_{init}$ is the concrete value of $a1_{init}$ and the interface initial value $a5_{init}$ is the concrete value of $a2_{init}$. ***/
6:6a [⟨($a4_{init}$ **eq** $a5_{init}$), *empty, empty*⟩,⟨$a5_{init}.next$,*4,6*⟩],

/*** **6:7a** is generated from **6:7** because at call site **9**, the interface initial value $a4_{init}$ is the concrete value of $a1_{init}$, the interface initial value $a5_{init}$ is the concrete value of $a2_{init}$ and the interface initial def $a5_{init}.next_{initdef}$ is the concrete value of $a2_{init}.next_{initdef}$.
***/
6:7a [⟨($a4_{init}$ **neq** $a5_{init}$), *empty, empty*⟩,⟨$a5_{init}.next$,$a5_{init}.next_{initdef}$,*6*⟩],

/*** **6:7b** is generated from **6:7** because at call site **12**, $object_{10}$ is the concrete value of $a1_{init}$, $object_{11}$ is the concrete value of $a2_{init}$, ($object_{10}$ **neq** $object_{11}$) is true and *11* is the concrete value of $a2_{init}.next_{initdef}$.
***/
6:7b [*Empty-context*,⟨$object_{11}.next$,*11,6*⟩] **}**

Solution at program point **7**:
{ /*** **7:3a** is generated from **7:3** by instantiating $a1_{init}$ with the interface initial value $a4_{init}$ as $a4_{init}$ is the concrete value of $a1_{init}$ at call site **9**. ****/
7:3a [⟨*empty,(type(*$a4_{init}$*)∈A::update),empty*⟩,⟨*global1,1,7*⟩],

/*** **7:3b** is generated from **7:3** by instantiating $a1_{init}$ with $object_{10}$ as $object_{10}$ is the concrete value of $a1_{init}$ at call site **12**. Note that *(type(*$object_{10}$*)∈A::update)* evaluates to true as type of $object_{10}$ (i.e., *B*) ∈ { *A, B*} and hence the relevant context of **7:3b** is *Empty-context*.
***/
7:3b [*Empty-context*,⟨*global1,1,7*⟩],

/*** **7:4a** is generated from **7:4** by instantiating $a1_{init}$ with the interface initial value $a4_{init}$ as $a4_{init}$ is the concrete value of $a1_{init}$ at call site **9**.
***/
7:4a [⟨*empty,(type(*$a4_{init}$*)∈C::update),empty*⟩,⟨*global1,2,7*⟩] **}**

Figure 4.12: Phase III-dua solution for method1

4.5.5 Modularity

As explained in Section 1.6, *relevant context inference* is a *modular* data-flow analysis technique. Since Def-Use-Algo is an application of *relevant context inference*, Def-Use-Algo also has the same characteristic. 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-dua. The rest of the time, only a method's pointer summary transfer function (during Phase I-pta) or def-use summary transfer function (during Phase I-dua), or the Phase II solution at the entry node of a method needs to be in memory. Hence, this is a *modular* approach to finding def-uses.

4.6 Algorithm for finding reduced context cover

In this section we discuss some ways in which the relevant contexts of def-use associations, involving only interface initial values of a single *public interface method*, computed during Phase III-dua, can be used in designing relevant test cases for satisfying testing criteria based on def-use associations.

Let C be a set of conjunctions of potential aliases, potential non-aliases and type constraints. A set T is called a *reduced context cover* of C if and only if :

1. *size-of(T)* $\leq$ *size-of(C)*;
2. for each $c \in C$, $\exists t \in T \ni t \Rightarrow c$; and
3. for each $t \in T$, $\exists c \in C \ni t \Rightarrow c$.

Let $\mathbf{RelevantContextCover_L}$ be the set of relevant contexts of the def-use associations computed by Phase III-dua (see Section 4.5.4) for a library **L**. Let $\mathbf{RelevantContextCover_M}$ be the subset of $\mathbf{RelevantContextCover_L}$ such that each element of $RelevantContextCover_M$ involves only the interface initial values of a single *public interface method* **M** of L. Consider a call to M in a test case. A tester can completely specify the values of globals and parameters at the call to M. Alternatively, the tester may choose to specify the values of some of the globals or parameters of pointer type partially, and thus provide a test case template from which many test cases can be generated. In the latter situation, at the call to M, for each global or parameter v of pointer type that the tester wants to specify only partially, the tester may do one of the following: (1) indicate that v is completely unspecified or (2) specify the value of v using interface initial values. The value of each pointer field of an interface initial value used in the specification can be specified in one of the following ways: (1) indicate that it is completely unspecified, (2) specify that it is *null* or (3) specify its value using another interface initial value. For each interface initial value, the tester may optionally choose to specify its concrete type also. For example, one of the possible ways of specifying the value of the parameter *a4* of *method2* in Figure 1.5 is as follows: specify the value of *a4* as $a4_{init}$, put no restriction on the concrete type of $a4_{init}$ and specify the value of $a4_{init}.next$ as null.

Def-Use-Algo, like many other static analysis techniques (see Section 3.3.1), assumes that all realizable paths are executable (i.e., the result of a test is independent of previous tests and all the branches are possible). As a result, the relevant alias context and type context computed by Def-Use-Algo for a def-use association are only necessary conditions for the def-use association to happen. The actual execution of a def-use path may also depend on the values of non-pointer variables involved in a test expression. Thus, the value of each non-pointer field of an interface initial value used in the specification needs to be specified completely; Def-Use-Algo keeps track of only the definitions of non-pointer variables and not their values.

Given a template **t** for a call to M, let $\mathbf{ConsistentRelContCover_{M_t}}$ be the subset of $RelevantContextCover_M$ such that each element of $ConsistentRelContCover_{M_t}$ is consistent with the specification of t. One strategy for the tester is to manually[3] enhance t using each element of $ConsistentRelContCover_{M_t}$ and generate *size-of(*$ConsistentRelContCover_{M_t}$*)* test cases (one test case per element of $ConsistentRelContCover_{M_t}$) which can be executed separately. However, there is

[3]It is possible to automate this step to a great extent and we plan to investigate this in future.

a more efficient alternative: first the tester can compute a reduced context cover **ReducedCover** of $ConsistentRelContCover_{M_t}$ having as small a size as possible and then generate *size-of(ReducedCover)* test cases by enhancing t using the elements of $ReducedCover$. Intuitively, the goal is to cover as many def-use associations as possible starting with the partial specification given by the tester. Note that the execution of some def-use paths require that non-pointer variables have specific initial values and hence their coverage cannot be guaranteed unless these specific initial values are assigned by the tester.

For any pointer-type global, parameter or field of an interface initial value whose value is not mentioned in the template specification or in the relevant context used to enhance the template, the following can be done: If the variable or field has been read, any object of the declared type of the variable or field can be assigned to it. Otherwise, it can be assigned *null* as its initial value is not relevant for data-flow testing. For non-pointer-type fields of objects introduced during enhancement, one of the following alternatives can be used: (1) the tester may assign some values or (2) default values for the types of these fields may be assigned.

Given a template t of a call to a *public interface method* M and $ConsistentRelContCover_{M_t}$, the simple greedy algorithm shown in Figure 4.13 can be used for finding a reduced context cover $ReducedCover$ of $ConsistentRelContCover_{M_t}$ such that each element of $ReducedCover$ is consistent with t, although it may not find the smallest sized reduced context cover in some cases:

For example, $RelevantContextCover_{L_{example}}$ for $L_{example}$ given in Figure 1.5 is
$\{$ $\langle$*empty*$\rangle$, $\langle$*(a*4_{init} **eq** *a*5_{init}*)*$\rangle$, $\langle$*(a*4_{init} **neq** *a*5_{init}*)*$\rangle$,
$\langle$*(type(a*4_{init}*)*$\in$*A::update)*$\rangle$, $\langle$*(type(a*4_{init}*)*$\in$*C::update)*$\rangle$,
$\langle$*(type(e*2_{init}*)* $\leq$ *ET2)*$\rangle$, $\langle$*(type(e*2_{init}*)* $\leq$ *ET4)*$\rangle$,
$\langle$*(type(e*2_{init}*)* $\not\leq$ *ET2)* $\wedge$ *(type(e*2_{init}*)* $\not\leq$ *ET4)*$\rangle$ $\}$. The relevant def-uses whose relevant contexts are the non-empty elements of $RelevantContextCover_{L_{example}}$ are shown in Figure 4.15. Figure 4.14 shows the def-uses computed by Phase I-dua and Figure 4.15 shows the same def-uses after the unknown initial values and unknown initial defs have

```
ReducedCover = φ
for each s ∈ ConsistentRelContCover_{M_t}
  if s contains a representative unknown initial value
    ReducedCover = ReducedCover ∪ {s}
  else
    if (∃ r ∈ ReducedCover such that r ∧ s can be
        satisfied and r ∧ s is consistent with t)
      new-el = r ∧ s
      ReducedCover = ReducedCover - {r} ∪ {new-el}
    else
      ReducedCover = ReducedCover ∪ {s}
```

Figure 4.13: Greedy algorithm for finding reduced context cover

been instantiated by their concrete values in Phase III-dua.

Now consider a template *t1* of a call to the *public interface method method2* of $L_{example}$ where all pointer variables are left unspecified by the tester. $RelevantContextCover_{method2}$ is

$\{$ $\langle empty \rangle$, $\langle (a4_{init}$ **eq** $a5_{init}) \rangle$, $\langle (a4_{init}$ **neq** $a5_{init}) \rangle$,
$\langle (type(a4_{init}) \in A{::}update) \rangle$, $\langle (type(a4_{init}) \in C{::}update) \rangle \}$ and
$ConsistentRelContCover_{method2_{t1}}$ is same as
$RelevantContextCover_{method2}$. As a result, the reduced context cover *ReducedCover* computed by the algorithm given above is $\{$ $\langle (a4_{init}$ **eq** $a5_{init}) \wedge (type(a4_{init}) \in A{::}update) \rangle$, $\langle (a4_{init}$ **neq** $a5_{init}) \wedge (type(a4_{init}) \in C{::}update) \rangle \}$.

4.7 Propagation of *dfelms* in Phase I of Def-Use-Algo

This section presents an overview of information flow during Phase I of Def-Use-Algo. The overall structure of information flow during Phase I-dua of Def-Use-Algo is very similar to the information flow during Phase I-pta described in Section 2.8, except that Phase I-dua propagates *DUA-dfelms* instead of *PTA-dfelms* propagated by Phase I-pta. Thus, we skip the detailed description of the iterative worklist algorithm given in Section 2.8. We will illustrate the different steps of Phase I-dua using the example in Figure 1.5. The pseudocode for Phase I-dua transfer function for each kind of *ICFG* node is given in Appendix G.

5:5 $[\langle empty, (type(a1_{init}) \in A::update), empty\rangle, \langle r, 3, \langle 5, A::update\rangle\rangle]$,
5:6 $[\langle empty, (type(a1_{init}) \in C::update), empty\rangle, \langle r, 3, \langle 5, C::update\rangle\rangle]$

6:6 $[\langle (a1_{init}$ **eq** $a2_{init}), empty, empty\rangle, \langle a2_{init}.next, 4, 6\rangle]$
6:7 $[\langle (a1_{init}$ **neq** $a2_{init}), empty, empty\rangle, \langle a2_{init}.next, a2_{init}.next_{initdef}, 6\rangle]$

7:3 $[\langle empty, (type(a1_{init}) \in A::update), empty\rangle, \langle global1, 1, 7\rangle]$
7:4 $[\langle empty, (type(a1_{init}) \in C::update), empty\rangle, \langle global1, 2, 7\rangle]$

17:1 $[\langle empty, empty, (type(e1_{init}) \leq ET2)\rangle, \langle e1_{init}, 15, 17\rangle]$

18:4 $[\langle empty, empty, (type(e1_{init}) \leq ET2)\rangle, \langle global3, 14, 18\rangle]$

25:1 $[\langle empty, empty, (type(e2_{init}) \leq ET4)\rangle, \langle e2_{init}, 15, 25\rangle]$

26:5 $[\langle empty, empty, (type(e2_{init}) \leq ET4)\rangle, \langle global3, 14, 26\rangle]$

27:1 $[\langle empty, empty, (type(e2_{init}) \not\leq ET2) \wedge (type(e2_{init}) \not\leq ET4)\rangle, \langle e2_{init}, 15, 27\rangle]$

28:5 $[\langle empty, empty, (type(e2_{init}) \not\leq ET2) \wedge (type(e2_{init}) \not\leq ET4)\rangle, \langle global3, 14, 28\rangle]$

Figure 4.14: Part of Phase I-dua def-use solution

4.7.1 Overall structure of Phase I of Def-Use-Algo

A high-level pseudocode for Phase I of Def-Use-Algo is given in Figure 4.16. **phase-I-pta** performs Phase I-pta as described in Chapter 2. *mark-used-fields*, *initialize-worklist-I-dua* and *process-worklist-I-dua* perform Phase I-dua.

Figure 4.17 shows an overview of the call graph for Phase I-dua. Table 4.1 shows the section numbers and figure numbers containing the explanation/pseudocode of the functions invoked by Phase I-dua. Recall that if an entry is *terminal*, it means that for simplicity we will not be presenting the pseudocode of the corresponding function. We will only describe the properties of such a function.

As in the case of Phase I-pta, the reason why in Figure 4.16, the solution of each ICFG node that represents the return-site of a call node is augmented by the solution at the successor of the node is that during Phase I-dua, certain *DUA-dfelms* propagated across a call node are propagated to the successor of the return-site of the call node, bypassing

/***
Recall interface initial values $a4_{init}$ and $a5_{init}$ are respectively concrete values of $a1_{init}$ and $a2_{init}$ at call site **9**,
and interface initial def $a5_{init}.next_{initdef}$ is the concrete value of $a2_{init}.next_{initdef}$ at call site **9**.***/

5:5a [⟨*empty,(type(*$a4_{init}$*)*∈*A::update),empty*⟩,⟨*r,3,*⟨*5,A::update*⟩⟩]
5:6a [⟨*empty,(type(*$a4_{init}$*)*∈*C::update),empty*⟩,⟨*r,3,*⟨*5,C::update*⟩⟩]

6:6a [⟨*(*$a4_{init}$ **eq** $a5_{init}$*),empty,empty*⟩,⟨$a5_{init}$*.next,4,6*⟩]
6:7a [⟨*(*$a4_{init}$ **neq** $a5_{init}$*),empty,empty*⟩,⟨$a5_{init}$*.next,*$a5_{init}.next_{initdef}$*,6*⟩]

7:3a [⟨*empty,(type(*$a4_{init}$*)*∈*A::update),empty*⟩,⟨*global1,1,7*⟩]
7:4a [⟨*empty,(type(*$a4_{init}$*)*∈*C::update),empty*⟩,⟨*global1,2,7*⟩]

/*** The interface initial value $e2_{init}$ is the concrete value of $e1_{init}$ at call site **24**. ***/

17:1a [⟨*empty,empty,(type(*$e2_{init}$*)* $\leq$ *ET2)*⟩,⟨$e2_{init}$*,15,17*⟩]

18:4a [⟨*empty,empty,(type(*$e2_{init}$*)* $\leq$ *ET2)*⟩,⟨*global3,14,18*⟩]

25:1a [⟨*empty,empty,(type(*$e2_{init}$*)* $\leq$ *ET4)*⟩,⟨$e2_{init}$*,15,25*⟩]

26:5a [⟨*empty,empty,(type(*$e2_{init}$*)* $\leq$ *ET4)*⟩,⟨*global3,14,26*⟩]

27:1a [⟨*empty,empty,(type(*$e2_{init}$*)* $\not\leq$ *ET2)* $\wedge$ *(type(*$e2_{init}$*)* $\not\leq$ *ET4)*⟩,⟨$e2_{init}$*,15,27*⟩]

28:5a [⟨*empty,empty,(type(*$e2_{init}$*)* $\not\leq$ *ET2)* $\wedge$ *(type(*$e2_{init}$*)* $\not\leq$ *ET4)*⟩,⟨*global3,14,28*⟩]

Figure 4.15: Part of Phase III-dua def-use solution

```
for each SCC in reverse topological order
   Phase-I-pta()
   /*** following three functions perform Phase I-dua ***/
   mark-used-fields()
   initialize-worklist-I-dua()
   process-worklist-I-dua()
for each return-site n
   n.reaching-DUA-dfelms = n.reaching-DUA-dfelms ∪ n.successor.reaching-DUA-dfelms
```

Figure 4.16: Phase I of Def-Use-Algo

```
                        Phase I-dua
                             |
     ------------------------------------------------
     |                     |                          |
mark-used-fields    initialize-worklist-I-dua    process-worklist-I-dua
                            |                      /      / \
                            |                     /      /   \
                            |                    /      / ... \
                            |                   /     Phase I-dua
                            |                  /         node
                            |                 /     transfer functions
                            |                /         /
                            |               /         /
                            |              /         /
                            |             /         /
                 add-to-soln-and-worklist-if-needed-I-dua
```

Figure 4.17: Overview of the call graph for Phase I-dua

Function Name	Sections Containing Explanation/Pseudocode	Figures Containing Pseudocode
mark-used-fields	Sec 4.7.2	terminal
initialize-worklist-I-dua	Sec 4.7.3 and Appendix F	Fig F.2 in Appendix F
process-worklist-I-dua	Sec 4.7.4	Figs 4.18, 4.19
add-to-soln-and-worklist-if-needed-I-dua	Sec 4.7.4	Fig 4.19
Phase I-dua node transfer functions	Appendix G	Figs G.1 to G.14

Table 4.1: Guide to functions invoked by Phase I-dua

the return-site. This is done for the ease of analysis. This is a low-level technical detail which can be ignored for the time being.

4.7.2 mark-used-fields

mark-used-fields iteratively analyzes the current SCC to mark all the non-pointer fields of unknown initial values that are defined or read. This is needed because non-pointer fields are not considered in Phase I-pta. Iteration is needed because due to actual-to-uiv bindings, the use of a field in a callee may imply the use of another field in the caller. Note that Phase I-pta has already marked the pointer fields of unknown initial values that have been found to be used. For example, in *method1* in Figure 1.5, Phase I-pta marks $a1_{init}.next$ as written and $a2_{init}.next$ as read. Since $a3_{init}.next$ is not used in *method1* or in any method invoked during the lifetime of *method1*, $a3_{init}.next$ is not marked.

4.7.3 initialize-worklist-I-dua

initialize-worklist-I-dua (Figure F.2) initializes the worklist with the definitions generated at entry nodes, and reachable (marked by *mark-reachable*) assignment nodes, throw nodes, object creation sites and call nodes. This function is very similar to *initialize-worklist-I-pta* described in Section 2.8. The details of *initialize-worklist-I-dua* are given in Appendix F.

4.7.4 process-worklist-I-dua

process-worklist-I-dua(Figure 4.18) invokes the Phase I-dua node transfer functions discriminating by node type. *rdfe* is either (1) (*first case*) a *DUA-dfelm*-**a** [*ecfi*,*rc1*,⟨*var*,*def*,⟨*rc2*,*val*⟩⟩] or (2) (*second case*) a *DUA-dfelm*-**b** [**excp-def**,*rc1*,*excp-obj*,*throw-point*]. Intuitively, a Phase I-dua transfer function of a node computes the def-use-analysis effect of the code associated with the node on a new *DUA-dfelm* reaching the node (i.e., *rdfe*) and adds the resulting *DUA-dfelms* and the successors of the node to the worklist for the propagation of the resulting *DUA-dfelms* to the successors

```
process-worklist-I-dua( ) {
while worklist is not empty {
  wl-node = delete from work-list
  /*** wl-node = (node,dfe) ***/
  n = wl-node.node
  rdfe = wl-node.dfe
  if ( n is an assignment node )
    1: process-assignment-dua( n, rdfe )
  if ( n is a condition node )
    /*** i.e., n represents the test expression of if or while ***/
    2: process-condition-dua( n, rdfe )
  if ( n is an object creation node )
    /*** e.g., p = new A() ***/
    3: process-new-dua( n, rdfe )
  if ( n is a throw node )
    4: process-throw-dua( n, rdfe )
  if (n is the entry node of a try)
    5: process-try-entry-dua( n, rdfe )
  if ( n is the exit node of a try block )
    6: process-try-exit-dua( n, rdfe )
  if ( n is the entry of a catch statement)
    7: process-catch-dua( n, rdfe )
  if ( n is the exit node of a catch statement )
    8: process-catch-exit-dua( n, rdfe )
  if (n is the entry node of a finally)
    9: add-to-soln-and-work-list-if-needed-I-dua(n.successor,{rdfe})
  if (n is the exit node of a finally)
    10: process-finally-exit-dua( n, rdfe )
  if ( n is a call node )
    11: process-call-dua( n, rdfe )
  if ( n is a method exit node )
    12: process-method-exit-dua( n, rdfe )
  if (n represents a break statement)
    13: process-break-dua( n, rdfe )
  if (n represents a continue statement)
    14: process-continue-dua( n, rdfe )
  if ( n is a return site )
    /*** n is the successor of a call node ***/
    15: process-return-site-dua( n, rdfe )
  if (n represents a return statement)
    16: process-return-statement-dua( n, rdfe )
} }
```

Figure 4.18: process worklist for Phase I-dua

add-to-soln-and-worklist-if-needed-I-dua(node, dfelms) {
for each $dfe \in dfelms$
if $dfe \notin node.reaching\text{-}DUA\text{-}dfelms$
$node.reaching\text{-}DUA\text{-}dfelms = node.reaching\text{-}DUA\text{-}dfelms \cup \{dfe\}$
wl-node = new worklist node (*node, dfe*)
add *wl-node* to *worklist* **}**

Figure 4.19: add-to-soln-and-worklist-if-needed-I-dua

of the node. *add-to-soln-and-worklist-if-needed-I-dua* is defined in Figure 4.19 and it is very similar to *add-to-soln-and-worklist-if-needed-I-pta* used by Phase I-pta. Similar to Phase I-pta, for each ICFG node n that is not the entry node of a method, *n.reaching-DUA-dfelms* represents the set of *DUA-dfelms* that have been found (so far) to reach the top of n.

4.8 Phase II

A high-level description of Phase II is given in Figures 4.20, 4.21 and 4.22.

phase-II-pta performs Phase II-pta as described in Chapter 2.

Phase II-dua consists of *initialize-worklist-II-dua* and *process-worklist-II-dua*.

initialize-worklist-II-dua (Figure 4.21) initializes the worklist with bindings between unknown initial defs and their concrete values at reachable call sites. It considers a binding only if the value of the unknown initial def is a program point or an interface initial def. Propagation through a binding in which the value of the unknown initial def is itself a non-interface-initial-def unknown initial def is done by *process-worklist-II-dua*. *instantiate-def-uidef-binding* instantiates the unknown initial values in the *rc* and *var* of a def-uidef binding with their concrete values computed at the entry node of the corresponding method during Phase II-pta. It returns those instantiations for which *rc* can evaluate to true: either (1) all the unknown initial values in *rc* are instantiated by heap-names and *rc* evaluates to true (*rc1* is *Empty-context* in this case), or (2) *rc* is instantiated into a relevant context *rc1* that involves only the interface initial values of a single *public interface method* and *rc1* can evaluate to

for each SCC in topological order
  phaseII-pta()
  /*** following two functions perform Phase II-dua ***/
  1: initialize-worklist-II-dua()
  2: process-worklist-II-dua()

Figure 4.20: phase II

true by making conservative, worst-case assumptions about these interface initial values. For example, consider call site **9** in *public interface method method2* in Figure 1.5. $[empty, Empty\text{-}context, \langle\langle a5_{init}.next, a5_{init}.next_{initdef}\rangle, a2_{init}.next_{initdef}\rangle] \in \mathbf{9}.def\text{-}uidef\text{-}bindings$, where $a5_{init}$ is an interface initial value and $a5_{init}.next_{initdef}$ is an interface initial def. As a result, *initialize-worklist-II-dua* propagates this def-to-uidef binding as the cv-to-uidef binding $[Empty\text{-}context, \langle a5_{init}.next, a5_{init}.next_{initdef}\rangle, a2_{init}.next_{initdef}]$ to the entry node of *method1*.

process-worklist-II-dua (Figure 4.22) iteratively propagates the concrete values of unknown initial defs to the entry nodes of methods. *instantiate-rc-under-constraint(m, rc, var, s)* instantiates the unknown initial values in the *rc* with their concrete values computed at the entry node of *m* during Phase II-pta; if *var* is an unknown initial value occurring in *rc*, *instantiate-rc-under-constraint* uses only *s* as the concrete value of *var* during each instantiation (hence the suffix under-constraint). This is the reason why *var* is stored in a def-uidef-binding. *instantiate-rc-under-constraint* returns a set of instantiations of *rc* that can evaluate to true. Each instantiation in this set is either (1) *Empty-context*, meaning the *rc* evaluates to true for the instantiation, or (2) it is a relevant context that involves only the interface initial values of a single *public interface method* and it can evaluate to true by making conservative, worst-case assumptions about these interface initial values.

1. initialize-worklist-II-dua() {
worklist = *empty*

for each reachable method *m* in the current SCC {
  for each *binding* ∈ *m.reaching-defs*
    wl-node = new worklist node (*m, binding*)
    add *wl-node* to *worklist*
  for each reachable call node *c* in *m* {
    for each *def-uidef binding* [*ecfi*,*rc*,⟨⟨*var,def*⟩,*uidef1*⟩] ∈ *c.def-uidef-bindings* {
      if *def* is a program point or an interface initial def
        a: *result* = instantiate-def-uidef-binding(*m*, [*ecfi*,*rc*,⟨⟨*var,def*⟩,*uidef1*⟩])
        // *uidef1.method* is the target (of *c*) with
        // whose entry node *uidef1* is associated
        add-to-soln-and-worklist-if-needed-II-dua(*uidef1.method*, *result*)
    }
  }
} **}**

1.a instantiate-def-uidef-binding(method, binding) {
result = ϕ
let *binding* be [*ecfi*,*rc*,⟨⟨*var,def*⟩,*uidef1*⟩]
for each instantiation of unknown initial values in ⟨*rc,var*⟩
    with their concrete values computed at
    the entry node of *method* during Phase II-pta {
  if rc *can be true* after the instantiation
    let *var1* be the value of *var* after the instantiation
    let *rc1* be the value of *rc* after the instantiation
    result = *result* ∪ {[*rc1*,⟨*var1,def*⟩,*uidef1*]}
}
return *result*
}

Figure 4.21: Phase II-dua worklist initialization

2. process-worklist-II-dua() {
while *worklist* is not *empty* {

wl-node = delete from *work-list*
/*** *wl-node = (method,binding)* ***/
m = *wl-node.method*
binding = *wl-node.binding*
let *binding* be [*rc1*,⟨*s,def*⟩,*uidef1*]

for each reachable call node *c* in *m*
for each *def-uidef binding* [*ecfi*,*rc2*,⟨⟨*var,uidef1*⟩,*uidef2*⟩] ∈ *c.def-uidef-bindings*
rc3 = *rc1* ∧ *rc2*
new-rcs = instantiate-rc-under-constraint(*m*, *rc3*, *var*, *s*)

for each *rc* ∈ *new-rcs*
reaching-def = [*rc*,⟨*s,def*⟩,*uidef2*]
add-to-soln-and-worklist-if-needed-II-dua(*uidef2.method*, {*reaching-def*})
} **}**

---

add-to-soln-and-worklist-if-needed-II-dua(method, bindings) {
for each *binding* ∈ *bindings*
if *binding* ∉ *method.reaching-defs*
method.reaching-defs = *method.reaching-defs* ∪ {binding}
if (method is in the current SCC)
wl-node = new worklist node (*method, binding*)
add *wl-node* to *worklist*
}

Figure 4.22: Phase II-dua worklist processing

Chapter 5

Empirical Evaluation of Points-to-Algo

We have built a proof-of-concept prototype of *RCI* for points-to analysis of C++programs. In this chapter we discuss this prototype and present empirical results obtained using this prototype. Our empirical results are encouraging and argue the effectiveness of Points-to-Algo in practice. Our prototype has been built using the PROLANGS Analysis Framework (PAF) which incorporates the Edison Design Group front end for ANSI C++.[1]

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

5.1 Lazy Strong Update

This section explains how our prototype performs strong updates or kills for the fields of unknown initial values. 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. Our prototype is sometimes 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]. The example in Section 2.5 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 *dfelms*. *RCI* performs kills for the fields of unknown initial values lazily. During Phase I, when a *dfelm* d representing the value of a field of an unknown initial value reaches a pointer assignment node n that can kill d according to the current points-to solution (i.e., the current solution implies that n *must* update that field of the unknown initial value), the decision for killing d cannot be made immediately. Until a fixed point is reached, only a partial solution is available at n and the *must* update could change to a *may* update according to a potentially larger fixed-point solution. In this situation, *RCI* does not propagate d immediately to the successors of n; instead, it marks n and d. After the worklist for a SCC becomes empty, *RCI* revisits the marked nodes to check if the marked *dfelms* are killed according to the current solution. If not, it unmarks the *dfelms* that should not be killed, and restarts iteration to propagate these unmarked *dfelms*. The propagation stops for a SCC when none of the remaining marked *dfelms* require repropagation (i.e., the marked nodes kill the marked *dfelms* according to the fixed-point solution).

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 later in this chapter, the SCC's are usually small in practice.

program	lines	ICFG nodes	methods	pointer variables	virtual calls	SCC's	Max SCC
trees	217	280	25	111	3	21	3
deriv1	192	320	27	136	28	23	3
employ	947	463	60	128	4	60	1
richards	987	918	87	303	82	87	1
deltablue	1509	1471	112	321	185	112	1
sampleAdv	3130	2203	130	397	9	130	1
vmatrix	3607	2783	128	559	9	128	1
vvector	3681	3401	138	511	9	138	1
FeynLib	6222	7247	229	1271	47	219	4
opProd	6289	7340	230	1700	47	219	4
penguin	6313	7450	230	1613	47	219	4
Bdecay	6293	7465	230	1591	47	219	4
electron	6321	7502	230	1625	47	219	4

Table 5.1: Benchmarks

5.2 Benchmarks

Table 5.1 contains some characteristics of the thirteen C++ programs we have analyzed. These are some of the benchmarks used in [PR96, BS96, CGZ95].*trees* implements trees, *deriv1* implements arithmetic expression trees, *employ* implements a class hierarchy for different kinds of employees in a company, *richards* is an operating system scheduler, *deltablue* is a symbolic constraint solver, *sampleAdv, vmatrix* and *vvector* perform matrix computations, and *FeynLib* is a library for drawing Feynman Diagrams for which *Bdecay, electron, opProd and penguin* are drivers for different kinds of elementary particles.

The Columns *lines*, *ICFG nodes*, *methods*, pointer variables, *virtual calls*, *SCC's* and *Max SCC* respectively show the number of lines of code, the number of ICFG nodes, the number of methods, the number of pointer variables (the number of user-defined pointer variables + the number of pointer fields of heap-names), the number of dynamically dispatched call sites, the number of nodes in SCC-DAG and the maximum number of methods in a SCC. Note that SCC's are small in practice and *RCI* needs to analyze only a small number of methods simultaneously.

Table 5.2 contains the timings obtained on a Sparc-20 with 352 megabytes of memory.

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

Table 5.2: Timings in Seconds

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 5.2, 5.3, 5.4 and 5.5, but there was no difference in the information reported for the two applications in Tables 5.6 and 5.7. 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 Tables 5.6 and 5.7.

Although Phase III-pta is demand-driven, for these experiments Phase III-pta was performed at all non-entry nodes. The Phase I-pta solution of a library can be shared by different driver programs. This is illustrated by the Phase I-pta timings of *Bdecay*,

program	Potential Aliases	Type Constraints	Bounds
trees	4	12	(∞,∞)
deriv1	0	116	(∞,∞)
deriv1	0	116	(∞,1)
employ	0	36	(∞,∞)
richards	85	5	(∞,∞)
richards	84	5	(1,1)
richards	77	5	(1,1)
deltablue	1	12	(∞,∞)
sampleAdv	11	5	(∞,∞)
vmatrix	10	2	(∞,∞)
vvector	10	6	(∞,∞)
FeynLib	115	535	(1,1)
FeynLib	245	579	(2,2)
opProd	115	535	(1,1)
opProd	245	579	(2,2)
penguin	115	535	(1,1)
penguin	245	579	(2,2)
Bdecay	115	535	(1,1)
Bdecay	245	579	(2,2)
electron	115	535	(1,1)
electron	245	579	(2,2)

Table 5.3: Performance Data I

electron, *opProd* and *penguin*. For these programs, $t_1 + t_2$ means the time for the driver code is t_1 and the time for the library code is t_2, a shared cost which is only incurred once.

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

5.3 Performance Data

In Table 5.3, the Columns *Potential Aliases* and *Type Constraints* show the total number of potential aliases and type constraints generated. Note that these numbers are small compared to the sizes of the programs. This shows that Points-to-Algo is quite effective

program	Max Size Relevant Context	Average No of Relevant Contexts per Points-to	Bounds
trees	1	1	(∞,∞)
deriv1	3	2.68	(∞,∞)
deriv1	1	1.49	$(\infty,1)$
employ	1	1	(∞,∞)
richards	3	1.17	(∞,∞)
richards	2	1.20	(1,1)
richards	1	1.16	(1,1)
deltablue	1	1	(∞,∞)
sampleAdv	2	1.00053	(∞,∞)
vmatrix	2	1.00017	(∞,∞)
vvector	2	1.00011	(∞,∞)
FeynLib	1	1.30	(1,1)
FeynLib	2	2.28	(2,2)
opProd	1	1.29	(1,1)
opProd	2	2.23	(2,2)
penguin	1	1.27	(1,1)
penguin	2	2.16	(2,2)
Bdecay	1	1.27	(1,1)
Bdecay	2	2.15	(2,2)
electron	1	1.26	(1,1)
electron	2	2.11	(2,2)

Table 5.4: Performance Data II

program	Average Size Pointer Summary Function	Bounds
trees	8.7%	(∞,∞)
deriv1	6.4%	(∞,∞)
deriv1	6.3%	$(\infty,1)$
employ	5.8%	(∞,∞)
richards	9.0%	(∞,∞)
richards	9.0%	(1,1)
richards	9.0%	(1,1)
deltablue	3.9%	(∞,∞)
sampleAdv	7.4%	(∞,∞)
vmatrix	7.4%	(∞,∞)
vvector	7.2%	(∞,∞)
FeynLib	3.5%	(1,1)
FeynLib	3.7%	(2,2)
opProd	3.5%	(1,1)
opProd	3.7%	(2,2)
penguin	3.5%	(1,1)
penguin	3.7%	(2,2)
Bdecay	3.5%	(1,1)
Bdecay	3.7%	(2,2)
electron	3.5%	(1,1)
electron	3.7%	(2,2)

Table 5.5: Performance Data III

in inferring only relevant contexts.

In Table 5.4, the Column *Max Size Relevant Context* shows the maximum number of conjuncts in any non-empty relevant context in the Phase I-pta solution. The Column *Average No of Relevant Contexts per Points-to* shows the average number of relevant contexts associated with a points-to in the Phase I-pta solution. Again, note that these numbers are small and this shows that Points-to-Algo is quite effective in inferring only relevant contexts.

In order to get an estimate of the memory saving obtained by using pointer summary transfer functions, we normalized the size of the pointer summary transfer function of each method with the size of the complete Phase I-pta 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 pointer summary transfer function for a program; results are presented as a percentage in Column *Average Size Pointer Summary Function* of Table 5.5. Another possibility would have been to compare the sizes of the pointer summary transfer functions with the number of nodes in the corresponding methods. But this is not reasonable because the cost of reanalyzing a method depends upon the size of the points-to solution of the method and the number of nodes reachable from the method during an invocation (not just the number of nodes in the method). Our comparison gives a better indication of the reduction in memory requirement achieved, as *RCI* needs to keep in memory only the pointer summary transfer functions of the methods called from the current SCC, rather than the complete solution of those methods.

No counts are reported for potential non-aliases because at present, our prototype does not generate any potential non-alias. By Theorem 1, this does not affect safety of the computed solution. The impact of potential non-aliases on the precision of points-to analysis is likely to be small. This is because a potential non-alias $uiv_1 \mathbf{neq} uiv_2$ contained in the relevant context of a *PTA-dfelm* propagated from the exit node of a method to a call site of the method (during Phase I-pta) must evaluate to *false* at the

program	AvAbstract MOD	AvConcrete MOD	Bounds
trees	1	1.54	(∞,∞)
deriv1	1	3.67	(∞,∞)
employ	1	2.60	(∞,∞)
richards	1.100	2.48	(∞,∞)
deltablue	1.062	2.50	(∞,∞)
sampleAdv	1	5.14	(∞,∞)
vmatrix	1	15.14	(∞,∞)
vvector	1	9.40	(∞,∞)
opProd	1.007	2.54	(1,1)
penguin	1.007	1.88	(1,1)
Bdecay	1.007	1.98	(1,1)
electron	1.007	2.17	(1,1)

Table 5.6: Side-effect analysis

call site to increase precision. If the potential non-alias evaluates to *true* then there is no loss in precision due to dropping the potential non-alias. The above potential non-alias will evaluate to *false* only if both uiv_1 and uiv_2 map to the same unknown initial value at the call site. If they both map to the same heap-name, the potential non-alias evaluates to *true* because the heap-name can represent more than one run-time object. We have chosen to describe Points-to-Algo as including calculation of potential non-aliases, because we believe they can be useful for other applications like data-flow-based testing. Test cases can be generated to ensure that two interface initial values are not equal at a call site that invokes a *public interface method*.

5.4 Applications of points-to solution

In order to test the quality of the solution computed by Points-to-Algo, we used the solution for two different applications: *side-effect analysis* or **MOD** [LRZ93, SRLZ98] and *virtual function resolution* [PR96]. The results for *MOD* are shown in Table 5.6. The Column *AvConcrete MOD* shows the average number of heap-names whose fields are modified by a pointer-assignment statement according to the Phase III-pta solution. In object-oriented programs, since a method is usually called from many different contexts, the number of heap-names modified at a statement could be large even in the

program	Reachable Virtual Calls	Unique Hierarchy	Differences RCI-Hierarchy	Bounds
trees	1	0	1	(∞,∞)
deriv1	20	10	10	(∞,∞)
employ	4	0	4	(∞,∞)
richards	82	81	1	(∞,∞)
deltablue	133	132	1	(∞,∞)
sampleAdv	2	0	2	(∞,∞)
vmatrix	6	5	1	(∞,∞)
vvector	2	0	2	(∞,∞)
opProd	5	0	5	(1,1)
penguin	6	0	6	(1,1)
Bdecay	4	0	4	(1,1)
electron	7	0	7	(1,1)

Table 5.7: Static resolution of dynamic dispatch

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

Table 5.7 and Figure 5.1 show our results for virtual function resolution. *RCI* is probably more precise than necessary for solving virtual function resolution for C++, because it is really aimed at problems where there is more gain from flow and context sensitivity; nevertheless, our results show that there are calls for which *RCI* enables much better resolution, with concomitant opportunities for aggressive optimizations such as instruction scheduling and specialization. Column *Reachable Virtual Calls* shows how many of the virtual calls in each program, are actually reachable (with the given drivers for libraries). Column *Unique Hierarchy* shows the number of reachable calls which are uniquely resolved by hierarchy analysis. Column *Differences RCI-Hierarchy* shows the number of those reachable calls for which the number of targets found by *RCI* is less than the number of targets found by hierarchy analysis. In Figure 5.1 we show the

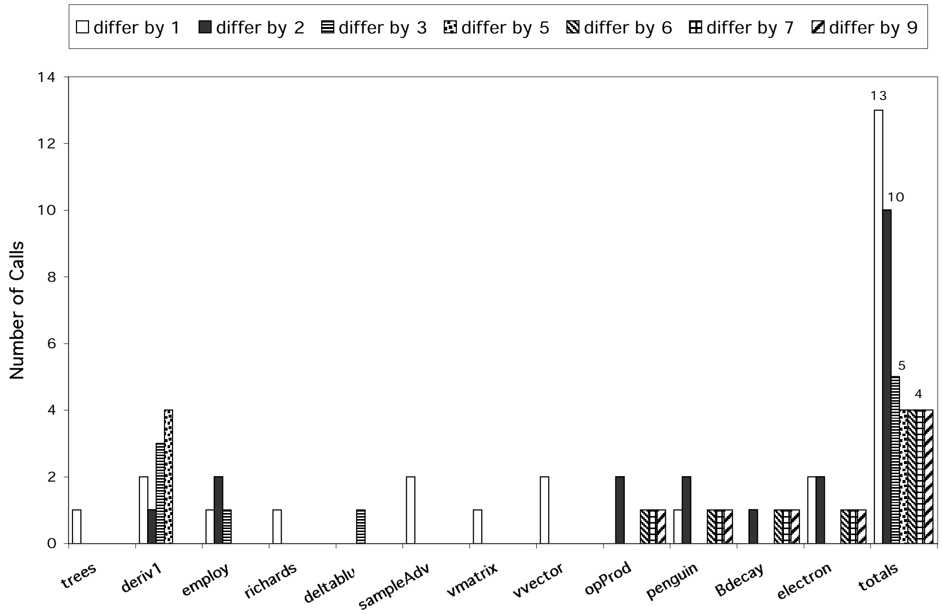

Figure 5.1: Differences in Precision between RCI and Hierarchy Analysis

number of these differing calls in each program and the sizes of the differences.

As noted above, changing bounds made no significant difference in the results of the two applications discussed above. In fact, there was no significant difference in the points-to solutions for the different bounds reported above. The possible loss of precision due to imposing the above reported bounds on the number of conjuncts in a relevant context was not observed for these benchmarks. For them, using lower bounds improved running times significantly without any significant loss in precision. However, for other applications and benchmarks the situation might be different.

Chapter 6

Related Work

6.1 Data-flow Analysis Techniques

The essential features of *RCI* as a data-flow analysis technique are the following:

- The decomposition of the call graph into SCC's for modular analysis.
- Bottom-up inference of calling contexts facilitating modular analysis and analysis of incomplete programs.
- The use of unknown initial values.
- The formulation of data-flow analysis as a multiple pass and multiple direction propagation on an SCC-DAG .

In this section, we will discuss other data-flow analysis techniques from the point of view of the above mentioned features of *RCI*. We will describe the use of unknown initial values by other researchers in Section 6.2, which deals with pointer analysis.

6.1.1 SCC Decomposition

The use of SCC decomposition for data-flow analysis was proposed in [HDT87]. Later, Marlowe and Ryder [MR90b] used the decomposition of the control flow graph of a procedure into strongly connected components in their *hybrid* algorithm for data-flow analysis. Intuitively, the main idea of the *hybrid* algorithm is to factor the data-flow problem on strongly connected components of the control flow graph into *local* and *external* parts, solving the local parts by iteration and propagating these effects on the SCC condensation of the control flow graph to obtain the entire data-flow solution. The *hybrid* algorithm uses unknown initial values to model representative external values in a local analysis. Marlowe and Ryder [MR90b] considered Fortran-like languages without general purpose pointers, dynamic dispatch or exceptions.

6.1.2 Calling Context

There are many methods proposed for distinguishing calling contexts (i.e., the state of the call stack) in data-flow algorithms. Sharir and Pnueli [SP81] advocate the use of a call-string list of open and not yet closed procedure activations to label data-flow information precisely with the calling context in which it was obtained. They also suggest use of an approximate call-string consisting of the last j calls on the call stack. The call-string list is close to the approach used in the points-to algorithm developed at McGill University, where every procedure activation is analyzed separately [HN90, EGH94]; optimizations to reduce computation by reusing the results for similar calling contexts were suggested by Emami [Ema93], and have been subsequently developed by Wilson and Lam [WL95] for points-to analysis and Ghiya and Hendren for connection analysis [GH96]. Empirical data seems to suggest that such optimizations can dramatically reduce the number of contexts actually analyzed. Landi and Ryder [LR92] used *reaching aliases* to capture contexts in their alias analysis algorithm for C, where reaching aliases are the aliases reaching the entry node of a procedure. The aliases within a procedure are computed conditioned on the aliases reaching the entry node of the procedure. [LR91] showed that using a single reaching alias in conditions yields the precise solution for programs with only single-level pointers, while [LR92] showed that using a single reaching alias in conditions yields a safe solution in the presence of general purpose pointers. Jones and Muchnick [JM82b] describe the use of an abstraction of the calling context at a dynamic creation site for a variable; the precision of this abstraction plus the approximation lattice for the data-flow problem in question determine the precision of the solution. Choi *et al.* use the immediate past call site as their encoding of the calling context in their flow- and context-sensitive aliasing algorithm [CBC93, MLR^{+}93, BCCH97]. They also describe an algorithm variant that uses alias sets of unrestricted size at the call site, called *source alias sets*, as additional call site encoding information. Their use of the previous call site name is the same as approximations suggested earlier by [JM82b, SP81]. [GDDC97] presents a more complex notion of context for object-oriented languages, which subsumes calling context and type context for some program variables. Shivers [Shi88] introduced control flow analysis of functional languages such as Scheme. [Shi88] uses the term *k-CFA* to describe the length of the call chain used in approximating the context of a function during the analysis. For example, *0-CFA* distinguishes no call site and *1-CFA* distinguishes by the last call site. The key difference between the techniques mentioned above and *RCI* is that contexts are propagated in a top-down manner in the above techniques, while *RCI* infers the contexts in a bottom-up manner. As a result, the above techniques are not suitable for modular analysis and analysis of incomplete programs. In addition, the above techniques do not consider flow due to exceptions, *RCI* can do context-sensitive analysis even in the presence of exceptions.

6.1.3 Modular Analysis

Cooper, Kennedy and Torczon [CKT86] discuss the $\mathbf{R^n}$ environment for optimizing Fortran programs. [CKT86] considers analysis such as side-effect analysis and constant propagation. In this approach procedures are first analyzed separately to compute summary information for each procedure (e.g., jump functions for constant propagation and set of modified variables for side-effect analysis). These procedure summaries are then used for interprocedural analysis. When the code for a procedure is changed, only the summaries of affected procedures need to be recomputed.

Heintze [Hei90, Hei92] introduced a *set-based* approach to program analysis. This approach is based on a single notion of approximation that treats program variables as sets of values. The approach splits the analysis into two phases: a *specification* phase and a *solution* phase. During the specification phase, the analysis derives *constraints* on the sets of values that program variables may assume. These constraints approximate the data-flow relationships of the analyzed program. During the solution phase, the analysis produces sets of values that satisfy these constraints. The result is an approximation of the value sets for the program variables.

[FF97] presents *componential set-based analysis* for Scheme. [FF97] simplifies the constraint systems of each of the modules separately and then combines these simplified constraint systems to compute the solution for the whole program. In the second step, the simplified constraint systems of all the modules need to be in the memory simultaneously. In contrast, *RCI* needs only one SCC and the summary transfer functions of the methods directly called from the SCC to be in memory simultaneously. *RCI* separates the iterative computation within a module from the hierarchical propagation across the modules. [FF97] performs such iterative and hierarchical propagation for the simplified systems simultaneously, which requires that all the simplified systems be in memory simultaneously. In [FF97] and [FF96], there is no discussion of indirect modifications through aliases; [FF96] only discusses direct assignments to variables. Although this may not be critical for a functional language such as Scheme, handling of such modifications is crucial for languages such as C++ and Java. [FF97] is *flow-insensitive* and hence does not perform *strong updates*. [FF97] keeps context by copying a function's simplified constraints, while *RCI* keeps context using summary transfer functions computed in terms of unknown initial values.

6.1.4 Multiple Direction Propagation

Horwitz, Reps and Binkley [HRB90] have used multiple direction propagation in two phases on the program summary graph in their algorithm for interprocedural slicing for Fortran programs. Intuitively, given a set S of vertices contained in a procedure P as a slicing criterion, Phase I identifies nodes that can reach S and that are either in P or in a procedure that calls P (either directly or transitively); while Phase II identifies nodes that can reach S from procedures (transitively) called by P or from procedures called by procedures that (transitively) call P.

6.1.5 Other Techniques

[RHS95] presents a solution technique for interprocedural data-flow analysis problems that are distributive finite subset problems. [RHS95] reduces a data-flow analysis problem to the reachability problem on a graph called the *exploded* graph. The *exploded* graph is obtained by replacing each node in the control flow graph by a bipartite graph that represents the node transfer function of the control flow graph node. This can be done without affecting precision because the problems being considered are distributive finite subset problems. points-to analysis and def-use analysis are not distributive in the presence of general purpose pointers, exceptions or dynamic dispatch.

Kildall [Kil73] introduced the technique of using an iterative worklist algorithm for data-flow analysis formulated as the solution of a set of equations on a semi-lattice. He also introduced the idea of associating a node transfer function with each node in the control flow graph. He considered constant propagation, common subexpression elimination, elimination of redundant register load operations and live variable analysis for languages without general purpose pointers (i.e., pointers in languages such as C and C++), dynamic dispatch or exceptions.

6.2 Pointer Analysis

[LR92] presents a non-modular, whole-program-analysis algorithm for alias analysis of C programs. The algorithm in [LR92] is both flow- and context-sensitive. The aliases at a program point are computed conditioned on *reaching aliases* that represent aliases reaching the entry node of the procedure containing the program point. Since the reaching aliases are known in a top-down manner, this algorithm needs to keep the entire program in memory and it cannot analyze incomplete programs such as libraries. Moreover, some reaching aliases not relevant for analyzing a procedure may also be passed to a procedure. Unknown initial values are similar to *non-visible variables* used in [LR92] for summarizing the values of pointers that point to out-of-scope variables.

[HR96] presents an alias analysis algorithm based on the algorithm in [LR92] for alias analysis of modules written in C. Intuitively, the idea is to analyze a module for all possible reaching aliases. In contrast, *RCI* analyzes a library by inferring only the relevant potential aliases and concrete types.

[EGH94] presents a non-modular, whole-program-analysis algorithm for points-to analysis of C programs. The algorithm in [EGH94] is both flow- and context-sensitive. [EGH94] separates the analysis of pointers to stack from the analysis of pointers to heap and it represents all the heap-allocated objects using a single name. Context-sensitivity is achieved by analyzing a procedure for each path from the *main* function to the procedure in the procedure invocation graph. Recursion in the procedure invocation graph is handled by collapsing repeated subpaths in the procedure invocation graph. Calls through function pointers are handled by repeatedly expanding the procedure invocation graph as new targets are discovered during analysis at a call site that invokes through a function pointer. Unknown initial values are similar to *invisible variables* used in [EGH94] for summarizing the values of pointers that point to out-of-scope variables. Since the procedure invocation graph is constructed in a top-down manner during analysis, this algorithm needs to keep the entire program in memory and it cannot analyze incomplete programs such as libraries.

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

[And94] presents a flow- and context-insensitive algorithm for points-to analysis of C programs. It formulates the points-to analysis problem as a set of constraints of a non-standard type inference problem. It iteratively solves the constraints until a fixed point is reached. Since the constraints need to be solved together, it needs to keep all the constraints in memory simultaneously and it cannot analyze incomplete programs such as libraries.

[Ste96] presents a flow- and context-insensitive algorithm for points-to analysis of C programs. It also formulates the points-to analysis problem as a non-standard type inference problem but it uses a union-find algorithm to solve the type inference problem in almost linear time. This algorithm cannot be used for analyzing libraries due to unknown aliasing between interface initial values and unknown concrete type of interface initial values.

6.3 Concrete Type Inference and Call Graph Construction

Concrete type inference and call graph construction for object-oriented languages are widely studied problems. The goal of concrete type inference is to find the classes of objects to which a pointer can point at a program point. Both these problems are solved by Points-to-Algo. The concrete types of a pointer are the classes of the objects to which it can point, and the final call graph is constructed during Phase II-pta. There are many non-modular, whole-program-analysis approaches for these two problems. Most use constraint-based analysis [Suz81, PS91, PC94, Age95, GDDC97, DGC98], but a few [CHS95, PR96, DMM96] are data-flow-based. *RCI* differs from all these approaches because it is modular, it is able to analyze incomplete programs and it can analyze programs that have exceptions. Here we will discuss only some representative techniques from the set of techniques mentioned above.

[PS91] formulates the concrete type inference problem for programs written in a language like Smalltalk as a set of conditional constraints, simplifies the constraints and then solves the simplified set of constraints iteratively until a fixed point is reached. A program *trace graph* is built to to represent type information about all program executions. Intuitively, for each call site that can invoke a method, there is a node for the method in the trace graph. Each node of the trace graph contains a collection of *local constraints* that the types of expressions must satisfy. The edges of the trace graph represent connections between a message send and a method that may implement it. *Connecting constraints* are associated with trace graph edges to reflect the relationship between formal and actual parameters and results. The *local* and *connecting constraints* are combined to obtain *global constraints* for the entire program. No implementation is reported in [PS91].

[PC94] presents a concrete type inference algorithm for the *concurrent aggregates* language. It provides a way to increase precision in *interesting* regions of a program by splitting the contexts (thus increasing context-sensitivity in a sense) in which a procedure is analyzed. The contexts can be repeatedly split and the resulting constraints can be resolved until desired precision is reached or no more increase in precision can be obtained.

[Age95] presents a cartesian product algorithm for concrete type inference of *Self* programs. A procedure is analyzed for each combination of concrete types of the parameters of the procedure and the solutions are tagged by the corresponding combinations of the concrete types of the parameters. The set of contexts for which a procedure is analyzed is a cartesian product of the concrete types of the parameters of the procedure. Whenever a combination of concrete types of the actuals is encountered at a call site the computed solutions for the targets are searched to see if the solution has been already computed for this combination of the concrete types of the parameters.

As stated earlier, [GDDC97] presents a more complex notion of context which subsumes calling context and type context for some program variables. It presents a general framework for call graph construction of object-oriented programs. [GDDC97] shows that [PS91], [PC94], [Age95] and *k-CFA* are special cases of the general framework presented in it. The key difference between the general framework presented in [GDDC97] and

RCI is that contexts are propagated in a top-down manner in [GDDC97], while *RCI* infers the contexts in a bottom-up manner. As a result, the framework in [GDDC97] is not suitable for modular analysis and analysis of incomplete programs. As stated earlier, another difference between [GDDC97] and *RCI* is that [GDDC97] does not consider flow due to exceptions.

Among constraint-based approaches, [Suz81, DGC98] are flow- and context-insensitive, while [PS91, PC94, Age95] are flow-insensitive, but context-sensitive.

[PR96] extends the alias analysis algorithm presented in [LR92] for concrete type inference of C++ programs. As a result, the algorithm in [PR96] is both flow- and context-sensitive. It computes the aliases at a program point conditioned on both reaching aliases and reaching types. It uses the technique used in [EGH94] for calls through function pointers to deal with virtual function calls. It expands the the set of possible targets of a virtual function call site as new targets are discovered during the analysis. We have compared our empirical results with those in [PR96] which were non-scalable.

[CHS95] is another data-flow-based flow- and context-sensitive algorithm for concrete type inference of C++ programs. It infers the concrete types of pointers from the classes of objects to which the pointer may point. The algorithm in [CHS95] is based on the alias analysis algorithm in [CBC93, MLR$^+$93]. No implementation is reported in [CHS95].

[DMM96] presents different type analysis techniques for Modula-3: hierarchy analysis, flow-sensitive intraprocedural type propagation and context-insensitive interprocedural type propagation. It presents experimental results comparing the effectiveness of different combinations of these techniques. It formulates intraprocedural type propagation as a data-flow problem similar to reaching definitions. Types are propagated from *type events* (such as object allocation, assignment or type discrimination operators) to method invocations within a procedure. The context-insensitive interprocedural type propagation propagates types only to scalars and it assumes the most conservative type (the declared type) for all data accessed through pointer traversal. The interprocedural type propagation does not propagate side effects from calls and assigns the most conservative type (the declared type) for any variable changed by the call.

[DMM98] presents different techniques for detecting aliases in Modula-3 programs using types. Intuitively, two names are aliases if and only if (a) the names have the same declared types, (b) their declared types are related by subtype-supertype relationship or (c) their is a common subtype of the declared types of the two names. They propose different techniques such as using field names and merging types due to assignments (similar to [Ste96]) to improve the precision of the basic scheme mentioned above. This technique can be applied to libraries.

[BS96] presents an extension of class hierarchy analysis for C++. It computes the set of types that are instantiated in the program. The set of possible types of a pointer consists of those subtypes of the declared type of the pointer that are in the set of instantiated types. The approach in [BS96] is flow- and context-insensitive.

Class hierarchy analysis has also been used in [Fer95] and [DGC95] for optimizing Modula-3 and Cecil programs respectively.

6.4 Complexity Characterizations

[Lan92b, Ram94] show that alias analysis for programs written in a C-like language is undecidable. [Ram94] presents simpler proofs of the results presented in [Lan92b]. It is easy to modify the proof in [Lan92b] to show that points-to analysis for programs written in RL is undecidable.

[GI98] characterizes the complexity of points-to analysis[1] in the presence of dynamic dispatch, recursion and multi-level pointers. [GI98] does not consider flow due to exceptions. [GI98] shows that for non-recursive programs with single-level types and dynamic dispatch, points-to analysis is **PSPACE**-complete, and for recursive programs with single-level types and dynamic dispatch, points-to analysis is **EXP-TIME**-complete.

[1]Strictly speaking [GI98] considers the problem of concrete type inference.

Our common result is that for programs with single-level types and dynamic-dispatch, points-to analysis is **PSPACE**-hard. The **PSPACE**-completeness proof we presented for intraprocedural points-to analysis for programs with single-level types and exceptions with subtyping can be easily adapted to show that for non-recursive programs with single-level types and dynamic dispatch, points-to analysis is in **PSPACE** and hence it is **PSPACE**-complete. This is because for non-recursive programs the call stack of return addresses is of length at most the total number of procedures in the program and hence can be explicitly maintained in polynomial amount of space.

6.5 Def-Use Analysis and Data-flow-based Testing

Data-flow-based testing has a long history [FO76, Ost77, LK83, RW85, FW88, LCS89, HS89, OW91]. Different data-flow-based testing criteria were introduced in [RW85] and later refined in [FW88] and [OW91]. [OW91] presents new definitions of def-uses for a C-like, using the alias solution at different program points. These definitions depend upon whether a location may/must be defined/used at a program point and whether there is or may be a definition clear path between two program points. These definitions are slightly unsatisfactory because they depend on the static alias analysis solution rather than program execution. This is why we have used a more traditional definition of def-uses that is based on execution paths.

[PLR94] presents a flow- and context-sensitive def-use analysis algorithm for C programs. The algorithm is based on the alias analysis algorithm presented in [LR92]. As a result, it is a whole-program-analysis algorithm and it cannot compute potential def-uses. It needs to know the aliases reaching the entry node of a procedure before it can compute the definitions and uses within the procedure. It first uses the algorithm in [LR92] to compute the alias solution at each program point. Then the alias solution is used to compute the definitions and uses that result from accesses through pointer dereferences.

[HR94] applied the algorithm in [PLR94] to C++ classes. As stated earlier, the approach presented in [HR94] cannot compute the potential def-uses due to possible aliasing at the entry node of a *public interface method*. [HR94] essentially puts calls to the different methods of a class within a big while loop, thus making the def-uses between program points contained in different methods explicit. Another advantage of our work is that it can be easily incorporated with the approach presented in [HR94]. [PLR94, HR94] did not consider flow due to exceptions.

[HS94] presents a flow- and context-sensitive algorithm for computing def-use and use-def chains in Fortran programs. The algorithm first abstracts intraprocedural def-use information for each procedure and then it uses this information to construct an interprocedural summary flow graph. The intraprocedural data-flow information is propagated throughout the program via the interprocedural summary flow graph to obtain sets of reaching definitions and reachable uses for each interprocedural control point. The algorithm handles aliasing between formals and aliasing between globals and formals. This is also a whole-program-analysis algorithm and it cannot be applied to libraries as it cannot compute potential def-uses.

[HFGO94] presents an empirical investigation of finding errors using data-flow-based testing criteria. Their conclusion is that although a set of test cases that achieves 100% data-flow-based coverage does not imply absence of errors, a program that has not been tested for 100% data-flow-based coverage has not been tested enough.

[RBS97] presents an algorithm for refining def-use solution using information about infeasible paths.

Although there are many tools [PFW85, HFGO94, Ost90] for checking def-use coverage, to the best of our knowledge, there is no tool for helping a test case writer to write test cases for satisfying testing criteria based on def-uses, except for pointing out the def-uses that need to be covered. One advantage of our approach is that the relevant contexts inferred by Def-Use-Algo can guide the the test case writer in writing test cases.

Chapter 7

Conclusions and Future Work

7.1 Future Work

7.1.1 Implementation

Our prototype of Points-to-Algo is a proof-of-concept implementation. We would like to further optimize the prototype, this will improve the experimental results further. We would like to pass bigger benchmarks (20000 lines) through our prototype. This will require collection of appropriate benchmarks.

At present our data-flow analyzer has only a C++ front-end. Since constructs for exception handling are rarely used in C++ programs (none of our benchmarks has exception handling constructs), we have not yet implemented those aspects of Points-to-Algo that deal with exceptions, and *ECFInfo* and *exception type context* of each data-flow-element computed by our prototype are *empty*. However, exceptions are frequently used in Java programs. Thus, in future we plan to add a Java front-end to our data-flow analyzer and implement those aspects of Points-to-Algo that deal with exceptions.

We also plan to implement Def-Use-Algo and incorporate it in a tool for program understanding and test case generation.

7.1.2 Extensions of *RCI*

7.1.2.1 Other data-flow analysis problems

We plan to investigate which other data-flow analysis problems can be solved using *RCI*. Those data-flow analysis problems that deal with flow of values of variables are promising candidates. In particular, we plan to solve constant propagation using *RCI*.

7.1.2.2 Threads

In this thesis we have not considered threads. We plan to extend *RCI* to work in the presence of Java-like threads. The information computed by *RCI* for programs with threads can be useful for many new applications like removal of redundant synchronizations, detection of local objects of threads, thread scheduling etc.

7.1.2.3 Parallelization of data-flow analysis

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. This approach to the parallelization of data-flow analysis is similar to the approach discussed in [Lee92, LR94]. The approach in [Lee92, LR94] is based on the *hybrid* algorithm discussed in Section 6.1.1 and thus, it uses the SCC decomposition of the control flow graph of a procedure for the parallelization of data-flow analysis. In contrast, *RCI* is based on the SCC decomposition of the call graph.

7.1.2.4 Flow-insensitive variants of *RCI*

Context-sensitivity is inherent to *RCI*, but flow-sensitivity is not. Flow-insensitive but context-sensitive variants of *RCI* are possible. We want to investigate the cost-precision tradeoff of these variants of *RCI*.

7.1.2.5 Unknown methods

RCI as presented in this thesis computes a *safe* solution for a library with respect to a driver program if at each dynamically dispatched call 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 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 solution computed by *RCI* is *safe* for a driver.

When an interface initial value is a possible receiver at a dynamically dispatched call 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[1], *then a method that is not known while*

[1] or an analogous situation exists for a call site that calls through a function pointer.

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* is a *typed unknown object* of the declared type of the variable or field. There is one typed unknown object for each type. *RCI* needs to make conservative, worst-case assumptions about typed unknown objects while analyzing the library. At a call site C that invokes a *public interface method* from a driver, a typed unknown object 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 unknown overriding method, 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 unknown overriding method. 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 unknown overriding methods. 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 the fields of unknown initial values and heap-names that can be modified by an unknown overriding method, only those globals mentioned in the annotations and those fields whose type signatures are mentioned in the annotations need to point to typed unknown objects after a call site that can invoke the unknown overriding method. The implementation of typed unknown objects, the investigation of their impact on precision and the use of annotations to increase precision are part of future work.

7.2 Conclusions

In this thesis we have presented a new data-flow analysis technique called relevant context inference (or *RCI*) for modular, flow- and context-sensitive data-flow analysis of statically typed object-oriented programming languages such as C++ and Java. This technique has several long sought-after characteristics:

1. *RCI* can analyze programs by keeping only a part of the program in memory at a time (with a constant bound on the number of times a procedure needs to be

moved into and out of memory).

2. *RCI* can analyze incomplete programs such as libraries.

3. *RCI* can analyze programs that have exceptions.

We have presented instantiations of *RCI* for two important data-flow analysis problems: points-to analysis and def-use analysis.

We have built a proof-of-concept prototype of *RCI* for points-to analysis of C++ programs. The empirical results obtained using this prototype show that *RCI* is effective in practice:

- We have obtained two orders of magnitude speedup over the Pande-Ryder [PR96] algorithm on certain benchmarks, without loss in precision.

- The average numbers of of locations modified by a through-deref statement[2] computed using Phase I-pta solution are close to one for most benchmarks we analyzed, indicating the precision of the solution computed by *RCI*.

- The average number of relevant contexts per points-to is around 2 for most benchmarks, indicating that *RCI* is quite effective in inferring only relevant contexts.

We have presented several new complexity characterizations of points-to analysis in the presence of object-oriented language constructs such as dynamic dispatch and exceptions. These complexity results identify the difficult constructs and indicate approximations that any efficient algorithm has to make. Some of our new complexity characterizations are:

- Points-to analysis for programs with only single-level types and without dynamic dispatch or exceptions with subtyping is polynomial time solvable.

- Points-to analysis for programs with single-level types and exceptions with subtyping is **PSPACE**-hard.

[2] A statement that modifies a location through pointer dereference.

- Points-to analysis of programs with only single-level types and without dynamic dispatch or exceptions can be solved (using *RCI*) in $O(n^4)$ worst-case time, improving the previously known bound of $O(n^7)$.

We have presented a new approach to data-flow-based testing of object-oriented libraries. We have shown how *RCI* can be used for computing def-use associations in object-oriented libraries and how the information computed by *RCI* can be used for generating relevant test cases. There are several new challenges in computing def-use associations in object-oriented libraries:

- First, the unknown aliasing between unknown initial values at the entry nodes of *public interface methods* needs to be handled;
- second, the unknown concrete types of the unknown initial values at the entry nodes of *public interface methods* need to be handled;
- and finally, flow due to exceptions needs to be handled.

We have shown how *RCI* can be used for overcoming the above challenges. *RCI* can be used for computing the def-use associations that arise in object-oriented libraries due to

- the potential aliasing between unknown initial values at the entry nodes of *public interface methods*,
- the potential concrete types of the unknown initial values at the entry nodes of *public interface methods*,
- exception objects that cause def-use associations between throw statements throwing the exception objects and the entry nodes of the corresponding catch statements which catch the exception objects (because the parameter of a catch statement is assigned the caught exception object), and
- value flows along paths caused by exceptions.

References

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

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

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison–Wesley, 1986.

[Bar78] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, 1978.

[BCCH97] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Interprocedural pointer alias analysis. Research Report RC 21055, IBM T. J. Watson Research Center, December 1997.

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

[Cal88] D. Callahan. The prgram summary graph and flow-sensitive interprocedural dat a flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 47–56, June 1988.

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

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

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

[CKT86] K. Cooper, K. Kennedy, and L. Torczon. Optimization of compiled code in the $\mathbf{r}^n$ programming env ironment. In *Proceedings of the 19th Annual Hawaii International Conference on Systems Sciences*, January 1986.

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

[CR99] Ramkrishna Chatterjee and Barbara Ryder. Data-flow-based testing of object-oriented libraries. Technical Report DCS-TR-382, Dept of CS, Rutgers University, March 1999.

[CRL97] Ramkrishna Chatterjee, Barbara Ryder, and William Landi. Complexity of concrete type-inference in the presence of exceptions. Technical Report DCS-TR-341, Dept of CS, Rutgers University, September 1997.

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

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

[CRL99] Ramkrishna Chatterjee, Barbara Ryder, and William Landi. Relevant context inference. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 133–147, January 1999.

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

[DGC95] Jeffery Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of Europ ean Conference on Object-Oriented Programming*, 1995.

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

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

[DMM98] Amer Diwan, Kathryn S. McKinley, and J.Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.

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

[Ema93] Maryam Emami. A practical interprocedural alias analysis for an optimizing/parallelizing C compiler. Master's thesis, McGill University, Montreal, Canada, July 1993.

[Fer95] Mary F. Fernandez. Simple and effective link-time optimization of modula-3 programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1995.

[FF96] C. Flanagan and M. Felleisen. Set-based analysis for full scheme and its use in soft-typing. Technical Report TR-96-266, Rice University, 1996.

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

[FI98] P. Frankel and O. Iakounenko. Further empirical studies of test effectiveness. In *ACM SIGSOFT '98 Sixth International Symposium on the Foundations of Software Engineering*, pages 153–162, November 1998.

[FO76] L. D. Fosdick and L. J. Osterweil. Data flow analysis in software reliability. *Computing Surveys*, 8:305–330, September 1976.

[FW88] P. Frankl and E. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[FW93] P. Frankel and S. Weiss. An experimental comparison of the effectiveness of branch testing with data-flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.

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

[GH96] R. Ghiya and L. Hendren. Connection analysis: A practical interprocedural heap analysis for c. *International Journal of Parallel Programming*, 1996.

[GI98] Joseph Gil and Alon Itai. The complexity of type analysis of object oriented programs. In *LNCS 1445, Proceedings of European Conference on Object-Oriented Programming*, July 1998.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[HDT87] S. Horwitz, A. Demers, and T. Teitelbaum. An efficient general iterative algorithm for data flow analysis˙ *Acta Informatica*, 24(6):679–694, 1987.

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, 1977.

[Hei90] N. Heintze. Set-based analysis of ml programs. In *In Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 306–317, 1990.

[Hei92] N. Heintze. Set-based program analysis. Technical report, Carnegie Mellon University, 1992.

[HFGO94] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 191–200, May 1994.

[HN90] L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transaction on Parallel and Distributed Systems*, 1990.

[HR94] Mary Jean Harrold and Gregg Rothermel. Performing dataflow testing on classes. In *Proceedings of the second Symposium on the Foundations of Software Engineering*, December 1994.

[HR96] Mary Jean Harrold and Greg Rothermel. Separate computation of alias information for reuse. *IEEE Transactions on Software Engineering*, 22(7), July 1996.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), January 1990.

[HS89] M. J. Harrold and M. L. Soffa. Interprocedural data flow testing. In *Proceedings of the Third Testing, Analysis, and Verification Symposium*, pages 158–167, December 1989.

[HS94] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

[HU79] J. E. Hofcroft and J. D. Ullman. *Introduction to automata theory, languages and computation.* Addison–Wesley, 1979.

[JM82a] N. D. Jones and S. Muchnick. *Flow Analysis and Optimization of LISP-like Structures.* Prentice Hall, 1982.

[JM82b] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 66–74, January 1982.

[Jon75] N. Jones. Space-bounded reducibility among combinatorial problems. *Journal of Computer and System Sciences*, 11:68–85, 1975.

[Kil73] G. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, January 1973.

[KU76] J.B. Kam and J.D. Ullman. Global data flow analysis and iterative algorithms. *Journal of ACM*, 23(1):158–171, 1976.

[Lan92a] W. A. Landi. Interprocedural aliasing in the presence of pointers, phd thesis. Technical Report LCSR-TR-174, Dept of CS, Rutgers University, 1992.

[Lan92b] W. A. Landi. Undecidability of static analysis. *acm Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[LCS89] D. Richardson L.A. Clarke, A. Podgurski and S. Seil. A comparison of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, November 1989.

[Lee92] Yong-fong Lee. *Performing Data Flow Analysis in Parallel*. PhD thesis, Department of Computer Science, Rutgers University, May 1992.

[LK83] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9:347–354, May 1983.

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

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

[LR94] Yong-fong Lee and Barbara G. Ryder. Effectively exploiting parallelism in data flow analysis. *The Journal of Supercomputing*, pages 233–262, 1994.

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

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

[MR90a] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: A unified model. *Acta Informatica*, 28:121–163, 1990.

[MR90b] Thomas. J. Marlowe and Barbara G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 184–196, January 1990.

[Ost77] L. J. Osterweil. The detection of unexecutable program paths through static data-flow analysis. In *In Proceedings of IEEE COMPSAC 77*, pages 406–413, December 1977.

[Ost90] Thomas J. Ostrand. Data-flow testing with pointers and function calls. In *Proceedings of the Pacific Northwest Software Quality Conference*, October 1990.

[OW88] T. J. Ostrand and E. Weyuker. Using data flow analysis for regression testing. In *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference*, September 1988.

[OW91] T. J. Ostrand and E. Weyuker. Data flow based test adequecy analysis for languages with pointers. In *Proceedings of the 1991 Symposium on Software Testing, Analysis and Verification (TAV4)*, October 1991. Victoria, B.C., Canada.

[Pap94] C. H. Papadimitriou. *Computational Complexity*. Addison–Wesley, 1994.

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

[PFW85] S. Weiss P.G. Frankl and E.J. Weyuker. Asset: A system to select and evaluate tests. In *In Proceedings of the IEEE Conference on Software Tools*, pages 72–79, April 1985.

[PLR94] H. D. Pande, W. Landi, and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

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

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

[Ram94] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, September 1994.

[RBS97] Rajiv Gupta Rastislav Bodik and Mary Lou Soffa. Refining data flow information using infeasible paths. In *Proceedings of the Fifth Symposium on the Foundations of Software Engineering*, September 1997.

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

[RM88] C. Ruggieri and T. Murtagh. Lifetime analysis of dynamically allocated objects. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 285–293, January 1988.

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

[RW85] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

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

[Shi88] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, Jun 1988.

[SP81] M. Sharir and A. Pnueli. *Two Approaches to Interprocedural Data Flow Analysis*. Prentice Hall, 1981.

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

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

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

[TCFR97] F. Tip, J-D Choi, J. Field, and G. Ramalingam. Slicing class hierarchies in c^{++}. In *Proceedings of OOPSLA'96: Conference on Object-oriented Programming Systems, Languages and Applications*, pages 179–197, October 1997.

[Tip96] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1996.

[TS97] F. Tip and P. F. Sweeney. Class hierarchy specialization. In *Proceedings of OOPSLA'97: Conference on Object-oriented Programming Systems, Languages and Applications*, pages 271–285, October 1997.

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

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

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

Appendix A

Mark-reachable

Figures A.1, A.2, A.3, A.4, A.5, A.6, A.7, A.8 and A.9 define *mark-reachable*. The data-flow elements computed by *mark-reachable* have one of the following three forms:

1. $\langle$*empty,reachable*$\rangle$

2. $\langle$*excp-type,reachable*$\rangle$

3. $\langle$*label,reachable*$\rangle$

```
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 ∪ n.successor.reaching-reachability-dfes
}
```

Figure A.1: 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 entry node of a finally)
    add-to-soln-and-worklist-if-needed-mr(node.successor,{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 represents a continue statement)
    process-continue-mr( node, dfe )
  if (node is a call node)
    process-call-mr( node, dfe )
  if (node is the return-site of a call node)
    // i.e., node is the successor of a call node
    process-return-site-mr( node, dfe )

  if (node represents a return statement)
    process-return-statement-mr( node, dfe )
} }
```

Figure A.2: mark-reachable 1

```
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 A.3: 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 A.4: mark-reachable 3

propagate-to-finally-if-needed-mr(node, dfe) {
  if (*dfe.ecfi* is *empty*)
    succ = successor of the try-catch-finally construct associated with *node*
    dfe = $\langle$*succ,reachable*$\rangle$
  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*,{$\langle$*empty,reachable*$\rangle$})
      else
        x = exit of *innermost-enclosing-exception-block(node.try)*
        add-to-soln-and-worklist-if-needed-mr(x,{*dfe*})
}

Figure A.5: mark-reachable 3a

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

Figure A.6: 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*,{$\langle$*empty,reachable*$\rangle$})
    else
      x = exit of *innermost-enclosing-exception-block(node.try)*
      add-to-soln-and-worklist-if-needed-mr(x,{*dfe*})
}

Figure A.7: mark-reachable 5

process-call-mr(node, dfe) {
/*** *node.ecfis* contains the *ECFInfo*'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 excp-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 A.8: 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,{$\langle$*y,reachable*$\rangle$})
}

---

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,{$\langle$*y,reachable*$\rangle$})
}

---

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, {$\langle$*method-exit,reachable*$\rangle$})
}

---

add-to-soln-and-worklist-if-needed-mr(node, dfes) {
for each *dfe* $\in$ *dfes*
  if *dfe* $\notin$ *node.reaching-reachability-dfes*
    node.reaching-reachability-dfes = *node.reaching-reachability-dfes* $\cup$ {*dfe*}
    wl-node = new worklist node (*node, dfe*)
    add *wl-node* to *worklist*
}

Figure A.9: mark-reachable 7

Appendix B

Pseudocode for initialize-worklist-I-pta

This appendix presents the pseudocode for worklist initialization done during Phase I-pta. Recall that the function *initialize-worklist-pta* performs worklist initialization during Phase I-pta. As far as possible, we will follow the following format for discussing the pseudocode of various functions. First, we will explain the purpose of the function and present the call graph of the function (if any). Next, we will discuss the data structures used in the pseudocode of the function. Finally, we will explain the pseudocode using an example. For simplicity, if the contents of any of the above three sections is obvious from context, we will skip the description of such a section.

B.1 initialize-worklist-I-pta (Figure B.2)

Purpose of ***initialize-worklist-I-pta*****:** *initialize-worklist-I-pta* initializes the worklist with *PTA-dfelms*-**a** representing (1) the unknown initial values of globals and parameters of pointer types, at the entry node of each method in the current SCC, and (2) context-independent points-tos generated at reachable (marked by *mark-reachable*) assignment nodes, object creation sites and call nodes. Recall that a node generates a context-independent points-to if the node generates the points-to in all calling contexts. For example, the statement *p = 0;* generates the context-independent points-to ⟨*p, null*⟩ in all calling contexts. *initialize-worklist-I-pta* puts the above *PTA-dfelms*-**a** on the worklist for the propagation of these *PTA-dfelms*-**a** to the successors of the ICFG nodes where these *PTA-dfelms*-**a** are generated.

Call graph of ***initialize-worklist-I-pta*****:** The call graph of *initialize-worklist-I-pta* is shown in Figure B.1. Table B.1 shows the section numbers and figure numbers

```
                        initialize-worklist-I-pta
                              |
      ----------------------------------------------
      |              |             |               |
    func1          func2         func3           func4
       \            /            /                / | \
        \          /            /                /  |  \
         \        /            /                /   |   \
          \      /            /                /    |   |
           \    /            /                /     |   |
         add-to-soln-and-worklist-if-needed-I-pta   |   |----
                                                    |        |
                                                  back-bind  |
                                                             |
                                            propagate-to-call-site

Here func1 is propagate-unknown-initial-values-from-entry-node,
     func2 is propagate-initial-points-tos-from-assignment-node,
     func3 is propagate-values-of-pointer-fields-of-newly-created-object
and  func4 is back-bind-using-pointer-summary-transfer-function.
```

Figure B.1: Overview of the call graph of initialize-worklist-I-pta

Function Name	Sections Containing Explanation/Pseudocode	Figures Containing Pseudocode
func1	Sec B.2	Fig B.3
func2	Sec B.3	Fig B.3
func3	Sec B.4	Fig B.3
func4	Sec B.5	Fig B.4
back-bind	Sec B.5	terminal
add-to-soln-and-worklist-if-needed-I-pta	Sec 2.8.2.2	Fig 2.27
propagate-to-call-site	Sec C.9.2 in Appendix C	Fig C.21

Table B.1: Guide to functions invoked by initialize-worklist-I-pta

```
initialize-worklist-I-pta( ) {
  worklist = empty

  for each method entry node n in the current SCC
    1: propagate-unknown-initial-values-from-entry-node( n )

  for each entry node n of each reachable try t in the current SCC
    if (innermost-enclosing-exception-block(t)  is a finally)
      2: propagate-unknown-initial-values-from-entry-node( n )

  for each reachable assignment node n in the current SCC
    3: propagate-initial-points-tos-from-assignment-node( n )

  for each reachable object creation site n in the current SCC {
    4: propagate-values-of-pointer-fields-of-newly-created-object(n)
  }

  for each reachable statically dispatched call site n in the current SCC
    // let m be the method invoked from n
    if (m is not in the current SCC)
      5:back-bind-using-pointer-summary-transfer-function(m,n)
}
```

Figure B.2: initialize worklist for Phase I-pta

1. propagate-unknown-initial-values-from-entry-node(n) {
/*** it propagates PTA-*dfelms*-**a** ***/
/*** n is the entry node of *n.method* ***/
/*** a global $g \in$ the *used-set* of *n.method*
if and only if either (1) g is used directly in a reachable node in the SCC containing *n.method* or (2) $g \in$ the *used-set* of one of the methods invocable from a reachable call node in the SCC containing *n.method* ***/
for each global $g \in$ the *used-set* of *n.method*
y = [*empty*,*Empty-context*,$\langle g, g_{init} \rangle$]
add-to-soln-and-worklist-if-needed-I-pta(*n.successor*, {*y*})

for each parameter p of *n.method* read in *n.method*
y = [*empty*,*Empty-context*,$\langle p, p_{init} \rangle$]
add-to-soln-and-worklist-if-needed-I-pta(*n.successor*, {*y*})
}

---

3. propagate-initial-points-tos-from-assignment-node(n) {
if (*n.lhs* is a pointer variable name and *n.rhs* is *0*)
for each ⟨*tag*, *reachable* ⟩ computed by *mark-reachable* at *n*,
such that *tag* is *empty* or a *label*
y = [*tag*,*Empty-context*,⟨*n.lhs*,*null*⟩]
add-to-soln-and-worklist-if-needed-I-pta(*n.successor*, {*y*})
}

---

4. propagate-values-of-pointer-fields-of-newly-created-object(n) {
for each ⟨*tag*, *reachable* ⟩ computed by *mark-reachable* at *n*,
such that *tag* is *empty* or a *label*
/*** propagate newly created object ***/
y = [*tag*,*Empty-context*,$\langle n.lhs, object_n \rangle$]
add-to-soln-and-worklist-if-needed-I-pta(*n.successor*, {*y*})

/*** propagate the values of pointer fields of newly created object ***/
for each pointer field *f* of $object_n$
y = [*tag*,*Empty-context*,$\langle object_n.f, null \rangle$]
add-to-soln-and-worklist-if-needed-I-pta(*n.successor*, {*y*})
}

Figure B.3: Auxiliary functions of initialize-worklist-I-pta

5. back-bind-using-pointer-summary-transfer-function(*m,n*) {
for each *dfe* $\in$ *m.exit-node.reaching-PTA-dfelms*
/*** *dfe* is *PTA-dfelm*-**a** [*ecfi*,*rc1*,$\langle$*var*,*val*$\rangle$] or *dfe* is *PTA-dfelm*-**b** [*rc1*,*excp-obj*] ***/
if (*dfe* does not contain any unknown initial value and
(*dfe* is a *PTA-dfelm*-**b** or (*dfe* is a *PTA-dfelm*-**a** and
dfe.var is not a local variable of *m*)))
x = back-bind(*dfe*, *n*, *m.exit-node*, *all*)
/*** Each method has a *return variable* associated with it, which
is assigned the value returned by the method.
If *var* is the *return variable*, back-bind
replaces it by the variable that is assigned the result of the call at *n*. ***/
/*** update the list of exceptions thrown by the call ***/
for each [*rc3*,*excp-obj1*] $\in x$
n.rc-excp-object-pairs = *n.rc-excp-object-pairs* $\cup$ { $\langle$*rc3*,*excp-obj1*$\rangle$ }
propagate-to-call-site(*n, x, dfe*)
}

Figure B.4: Auxiliary functions of initialize-worklist-I-pta

containing the explanation/pseudocode of the functions invoked by *initialize-worklist-I-pta*. If an entry is *terminal*, it means that for simplicity we will not be presenting the pseudocode of the corresponding function. We will only describe the properties of such a function.

B.2 propagate-unknown-initial-values-from-entry-node(n) (Figure B.3)

Purpose of *propagate-unknown-initial-values-from-entry-node***:** *propagate-unknown-initial-values-from-entry-node* initializes the worklist with *PTA-dfelms*-**a** representing the values of parameters and of globals.

Data structures used by *propagate-unknown-initial-values-from-entry-node***:** n is the entry node of *n.method*. Intuitively, the *used-set* of a method contains the global variables that could be used during the life-time of the method. Formally, a global g $\in$ the *used-set* of a method m if and only if either (1) g is used directly in a reachable node in the SCC containing m or (2) $g \in$ the *used-set* of one of the methods invocable from a reachable call node in the SCC containing m.

Explanation of the pseudocode using an example: *propagate-unknown-initial-values-from-entry-node* considers only those globals that are used and only those parameters that are read. For example, in Figure 1.5 at the entry node of *method1*, *propagate-unknown-initial-values-from-entry-node* adds the following *PTA-dfelms*-**a** to the worklist:

[*empty*,*Empty-context*,$\langle$*global1*,$global1_{init}\rangle$],

[*empty*,*Empty-context*,$\langle$*global2*,$global2_{init}\rangle$],

[*empty*,*Empty-context*,$\langle$*global3*,$global3_{init}\rangle$],

[*empty*,*Empty-context*,$\langle$*a1*,$a1_{init}\rangle$],

[*empty*,*Empty-context*,$\langle$*a2*,$a2_{init}\rangle$],

[*empty*,*Empty-context*,$\langle$*a3*,$a3_{init}\rangle$].

Recall (see Section 2.4.2.7) that a try statement directly nested inside a finally statement (i.e., *innermost-enclosing-exception-block(try)* is a finally statement) is treated like a call to an anonymous procedure. As a result, *propagate-unknown-initial-values-from-entry-node* is called to initialize the worklist with the unknown initial values of all the pointer variables visible at the entry of a try statement nested inside a finally statement. *propagate-unknown-initial-values-from-entry-node* treats all variables visible at the entry of the try statement as global variables for the purpose of propagating their unknown initial values from the entry node of the try statement.

B.3 propagate-initial-points-tos-from-assignment-node(n) (Figure B.3)

Purpose of *propagate-initial-points-tos-from-assignment-node***:** *propagate-initial-points-tos-from-assignment-node* initializes the worklist with *initial points-tos* generated at an assignment node. An assignment node n generates an *initial points-to* if and only if *n.lhs* is a pointer variable and *n.rhs* is *0*.

Data structures used by *propagate-initial-points-tos-from-assignment-node***:** n is an ICFG node that represents an assignment statement and *n.lhs* and *n.rhs* respectively represent the left-hand-side expression and the right-hand-side expression of the assignment statement represented by n.

Explanation of the pseudocode using an example: Suppose n is directly contained in a finally statement. Then, for every reason for entering the finally statement, a *PTA-dfelm*-**a** y needs to be propagated to the successor of n, such that y contains the initial points-to generated at n and the *ECFInfo* of y encodes the reason for entering the finally statement. If the finally statement is entered due to an uncaught exception, then this exception will be propagated to n by a *PTA-dfelm*-**b** representing the exception. As a result, Points-to-Algo uses *PTA-dfelms*-**b** reaching n for generating *PTA-dfelms*-**a** that contain the initial points-to and whose *ECFInfo* encode those reasons for entering the finally that are uncaught exceptions. This will become clear when the processing of a reaching *PTA-dfelm*-**b** by the Phase I-pta transfer function of an assignment node is described in Section C.1. For other reasons for entering the finally (i.e., pending transitions), *propagate-initial-points-tos-from-assignment-node* uses the information computed by *mark-reachable* (see Section 2.8.2.3) to generate the *ECFInfo*'s of the generated *PTA-dfelms*-**a**. Next, suppose n is is not directly contained in a finally statement. Then, since n is reachable, only $\langle$*empty,reachable*$\rangle$ will be computed by *mark-reachable* at n. Thus, *ECFInfo* of the *PTA-dfelm*-**a** y computed by *propagate-initial-points-tos-from-assignment-node* will be *empty*. *ECFInfo*'s of the generated *PTA-dfelms*-**a** will be non-empty only at nodes directly contained in a finally statement. Unfortunately, no assignment node in the example in Figure 1.5 generates an initial points-to.

B.4 propagate-values-of-pointer-fields-of-newly-created-object(n) (Figure B.3)

Purpose of *propagate-values-of-pointer-fields-of-newly-created-object***:** *propagate-values-of-pointer-fields-of-newly-created-object* initializes the worklist with two sets of *PTA-dfelms*-**a**. First, *propagate-values-of-pointer-fields-of-newly-created-object* initializes the worklist with *PTA-dfelms*-**a** which say that *n.lhs* points to the heap-name representing objects allocated at the site. Second, *propagate-values-of-pointer-fields-of-newly-created-object* initializes with *null* the pointer fields of the heap-name representing the objects allocated at an object creation site, and adds *PTA-dfelms*-**a** representing these points-tos to the worklist.

Data structures used by *propagate-values-of-pointer-fields-of-newly-created-object***:** n is an ICFG node representing an object creation site and $n.lhs$ is the left-hand-side variable of the object creation site (recall that in RL, the left-hand-side of each object creation statement is a variable, e.g., p = *new T(a,b))*.

Explanation of the pseudocode using an example: For the same reasons as explained for *propagate-initial-points-tos-from-assignment-node*, *propagate-values-of-pointer-fields-of-newly-created-object* uses the information computed by *mark-reachable* (see Section 2.8.2.3) to generate the *ECFInfo*'s of the generated PTA-*dfelms*-**a**. Recall that *ECFInfo*'s will be non-empty only at nodes directly contained in a finally statement. For example, at program point **10** in Figure 1.5, this function adds the following PTA-*dfelms*-**a** to the worklist:

[*empty*,*Empty-context*,⟨*p*,$object_{10}$⟩],

[*empty*,*Empty-context*,⟨$object_{10}$*.next*,*null*⟩]. The *ECFInfo* of these PTA-*dfelms*-**a** is *empty* because only ⟨*empty,reachable*⟩ is computed by *mark-reachable* at program point **10**.

B.5 back-bind-using-pointer-summary-transfer-function(m,n) (Figure B.4)

Purpose of *back-bind-using-pointer-summary-transfer-function***:** *back-bind-using-pointer-summary-transfer-function* If a method m that is not contained in the current SCC is invoked from a reachable statically dispatched call site n in the current SCC, *back-bind-using-pointer-summary-transfer-function* propagates from the exit node of the non-same-SCC method m, those PTA-*dfelms* that do not contain any unknown initial value and that do not represent the values of local variables of m to the return-site of the call site represented by n.

Data structures used by *back-bind-using-pointer-summary-transfer-function***:** n is an ICFG node representing a statically dispatched call site and m is a method invoked from n such that m is not contained in the SCC containing n. *dfe* is a PTA-*dfelm*-**a** [*ecfi*,*rc1*,⟨*var*,*val*⟩] or *dfe* is a PTA-*dfelm*-**b** [*rc1*,*excp-obj*]. x is a set of PTA-*dfelms*. Each method has a *return variable* associated with it, which is assigned the value returned by the method. For each ICFG node n that represents a call node, *n.rc-excp-object-pairs* stores pairs of exception objects thrown by the call node and the relevant contexts under which these exception objects are thrown.

Explanation of the pseudocode using an example: *back-bind* instantiates a *PTA-dfelm* at the exit node of a method with the actuals at a call site of the method. It returns the set of *PTA-dfelms* resulting from the instantiations. If *var* of *dfe* represents the *return variable* of m, in the instantiated *PTA-dfelm*-**a**, *back-bind* replaces *var* by the variable that is assigned the result of the call at n. Intuitively, the argument *all* of *back-bind* means that *back-bind* is being asked to look at all the actual-to-uiv bindings at the call site. The significance of this parameter of *back-bind* will become clear later in Section C.9, for the time being it can be ignored. For example, in Figure 1.5, [*empty*,*Empty-context*,⟨*global1*,$object_8$⟩] reaches the exit of *method1* and hence this *PTA-dfelm*-**a** is part of the pointer summary transfer function of *method1*. Thus, when Phase I-pta is done on *method2*, at call site **9**, the above *PTA-dfelm*-**a** is added to the worklist by *back-bind-using-pointer-summary-transfer-function*. If *dfe* represents an exception object (i.e., a *PTA-dfelm*-**b**), the list of exceptions thrown by the call node n may need to be updated. For example, [*Empty-context*,$object_{31}$] reaches program point **33** and hence this *PTA-dfelm*-**b** is part of the pointer summary transfer function of *method5*. Thus, when Phase I-pta is done on *method6*, ⟨*Empty-context*,$object_{31}$⟩ is added to *35.rc-excp-object-pairs*.

Intuitively, *propagate-to-call-site(n,x,dfe)* propagates the *PTA-dfelms* contained in x to the return-site of n. *propagate-to-call-site* is defined later in Figure C.21.

Appendix C

Points-to analysis node transfer functions

This appendix presents the pseudocode for Phase I-pta transfer functions for *ICFG* nodes. Recall that a Phase I-pta transfer function of an ICFG node computes the points-to-analysis effect of the code associated with the node on a new *PTA-dfelm* reaching the node and adds the resulting *PTA-dfelms* and the successors of the node to the worklist for the propagation of the resulting *PTA-dfelms* to the successors of the node. As far as possible, we will follow the following format for discussing the pseudocode of various functions. First, we will explain the purpose of the function and present the call graph of the function (if any). Next, we will discuss the data structures used in the pseudocode of the function. Finally, we will explain the pseudocode using an example. For simplicity, if the contents of any of the above three sections is obvious from context, we will skip the description of such a section.

Throughout this appendix *rdfe* is either (1) (*first case*) a *PTA-dfelm*-**a** [*ecfi*,*rc1*,⟨*var,val*⟩] or (2) (*second case*) a *PTA-dfelm*-**b** [*rc1*,*excp-obj*], unless stated otherwise.

C.1 process-assignment-pta (Figure C.2)

Purpose of *process-assignment-pta***:** *process-assignment-pta* computes the points-to-analysis effect of an assignment statement on a *PTA-dfelm* reaching the top of the assignment statement. If the assignment statement does not assign to any pointer variable, then the assignment statement is pass-through for points-to analysis. Otherwise, *process-assignment-pta* incrementally computes a set of locations for the left-hand-side of the pointer assignment statement and a set of locations for the right-hand-side of the pointer assignment statement. Roughly speaking, a points-to is generated for each

```
                        process-assignment-pta
                              |
        ----------------------------------------------------
        |       |     |       |      |      |      |       |
      func1   func2  func3  func4  func5  func6  kills   can_update
       / \      |     |
      /   \     |     |
     /     \    |     |
  get-ecfi  \   |     |
             \  |     |
          add-to-soln-and-worklist-if-needed-I-pta
```

```
Here func1 is process-new-exception,
     func2 is lazily-mark-as-written-field,
     func3 is lazily-mark-as-read-field,
     func4 is get_new_generated_dfes,
     func5 is generate-points-tos-due-to-pot-aliases
and  func6 is generate-points-tos-due-to-pot-non-aliases.
```

Figure C.1: Overview of the call graph of process-assignment-pta

Function Name	Sections Containing Explanation/Pseudocode	Figures Containing Pseudocode
func1	Sec C.1.1	Fig C.3
func2	Sec C.1.6	terminal
func3	Sec C.1.6	terminal
func4	Sec C.1.3	Fig C.3
func5	Sec C.1.4	Fig C.4
func6	Sec C.1.5	Fig C.5
kills	trivial	Fig C.4
can_update	trivial	Fig C.4
get-efci	trivial	Fig C.3
add-to-soln-and-worklist-if-needed-I-pta	Sec 2.8.2.2	Fig 2.27

Table C.1: Guide to functions invoked by process-assignment-pta

```
1. process-assignment-pta( n, rdfe ) {
if ( n does not modify any pointer or rdfe is a PTA-dfelm-b )
  add-to-soln-and-worklist-if-needed-I-pta(n.successor, {rdfe})
  if ( rdfe is a PTA-dfelm-b )
    process-new-exception( rdfe, n )
  return
// Here we know that n is a pointer-assignment node and rdfe is a
// PTA-dfelm-a [ecfi,rc1,⟨var,val⟩]

old_lhs_rc_loc_pairs = n.lhs_rc_loc_pairs;
old_rhs_rc_loc_pairs = n.rhs_rc_loc_pairs;

new_lhs_rc_loc_pairs = new rc_loc_pairs for left hand side implied by rdfe;
for each ⟨ecfi,rc,u ⟩ ∈ new_lhs_rc_loc_pairs
  Let n.lhs be p→f1
  if (u is v.f1 such that v is an unknown initial value and v.f1
    has not been marked as written)
    /*** lazy generation of fields of unknown initial values ***/
    lazily-mark-as-written-field( v.f1 )

new_rhs_rc_loc_pairs = new rc_loc_pairs for right hand side implied by rdfe;
if (n.rhs is of the form q→f2 and rdfe.var is q)
  if ( val is an unknown initial value and val.f2 has not been marked as read )
    /*** lazy generation of fields of unknown initial values ***/
    lazily-mark-as-read-field( val.f2 )

n.lhs_rc_loc_pairs = old_lhs_rc_loc_pairs ∪ new_lhs_rc_loc_pairs;
n.rhs_rc_loc_pairs = old_rhs_rc_loc_pairs ∪ new_rhs_rc_loc_pairs;

new_generated_dfes = get_new_generated_dfes( n, old_lhs_rc_loc_pairs,
                new_lhs_rc_loc_pairs, old_rhs_rc_loc_pairs, new_rhs_rc_loc_pairs )

new_generated_dfes = new_generated_dfes ∪
        generate-points-tos-due-to-pot-aliases(n.method, new_generated_dfes);
new_generated_dfes = new_generated_dfes ∪
    generate-points-tos-due-to-pot-non-aliases(new_lhs_rc_loc_pairs,
      n.reaching-PTA-dfelms, n);

if ( !kills(n, rdfe) ) {
  if (can_update(var,n)) {
    new_generated_dfes = new_generated_dfes ∪
          generate-points-tos-due-to-pot-non-aliases(n.lhs_rc_loc_pairs, {rdfe}, n);
  }
  else {
    new_generated_dfes ∪ {rdfe};
  }
}

add-to-soln-and-worklist-if-needed-I-pta( n.successor, new_generated_dfes ) }
```

Figure C.2: Phase I-pta transfer function for assignment node

```
get_new_generated_dfes( n, old_lhs_rc_loc_pairs, new_lhs_rc_loc_pairs,
                        old_rhs_rc_loc_pairs, new_rhs_rc_loc_pairs ) {
new_generated_dfes = φ;
// The following loop is executed only if n.lhs has dereference.
// Otherwise new_lhs_rc_loc_pairs is φ.
1: for (each ⟨ecfi,rc2,u ⟩ in new_lhs_rc_loc_pairs) {
  if (u != null) {
    if (n.rhs is 0)
      new_dfe = [ecfi,rc2,⟨u,null⟩];
      new_generated_dfes = new_generated_dfes ∪ {new_dfe};
    else
      for (each ⟨ecfi,rc3,v ⟩ in n.rhs_rc_loc_pairs) {
        rc4 = rc2 ∧ rc3;
        new_dfe = [ecfi,rc4,⟨u,v⟩];
        new_generated_dfes = new_generated_dfes ∪ {new_dfe};
    }
  }
}

2: for (each ⟨ecfi,rc3,v ⟩ in new_rhs_rc_loc_pairs) {
  if (n.lhs does not have dereference)
    new_dfe = [ecfi,rc3,⟨n.lhs,v⟩];
    new_generated_dfes ∪ {new_dfe};
  else
    for (each ⟨ecfi,rc2,u ⟩ in old_lhs_rc_loc_pairs) {
      if (u != null) {
        rc4 = rc2 ∧ rc3;
        new_dfe = [ecfi,rc4,⟨u,v⟩];
        new_generated_dfes = new_generated_dfes ∪ {new_dfe};
      }
    }
}
return new_generated_dfes }
```

```
process-new-exception( rdfe, n ) {
/*** rdfe carries a new exception context to n.  So if n generates an
    initial points-to, this points-to needs to be generated in this
    new exception context.  ***/
  if ( n.lhs is a pointer variable name and n.rhs is 0 )
    let rdfe be [rc1,excp-obj]
    y = [get-ecfi(excp-obj),rc1,⟨n.lhs,null⟩]
    add-to-soln-and-worklist-if-needed-I-pta(n.successor, {y})
}
```

```
get-ecfi( object ) {
  if ( object is a heap-name )
    return typeof(object)
  else // object is an unknown initial value
    return object
}
```

Figure C.3: auxiliary function for process-assignment-pta

```
kills( n, rdfe ) {
// rdfe is PTA-dfelm-a [ecfi,rc1,⟨var,val⟩]
  if (n.lhs does not have dereference) {
    if (var and n.lhs are the same variable)
      return true;
  }
return false; }

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

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

generate-points-tos-due-to-pot-aliases(method, dfes) {
  generated-dfes = φ
  for each dfe ∈ dfes
    let dfe be [ecfi1,rc2,⟨var1,val1⟩]
    let rc2 be ⟨ac2, tc2, etc2⟩
    if ( var is u.f and u is an unknown initial value ) {
      for each unknown initial value v at method.entry compatible with u
        if ( v.f has been marked as read ) {
          rc3 = ⟨ac2 ∧ (u eq v), tc2, etc2⟩
          new-dfe = [ecfi1,rc3,⟨v.f,val1⟩]
          generated-dfes = generated-dfes ∪ {new-dfe}
        }
    }
  return generated-dfes }
```

Figure C.4: auxiliary functions for process-assignment-pta

```
generate-points-tos-due-to-pot-non-aliases( rc_loc_pairs, dfes, n ) {
if (n.lhs does not have dereference)
  return φ;

// else say n.lhs is p→f1 and p is of type A *
generated_dfes_pna = φ;
for (each dfe [ecfi1,rc2,⟨u.f1,y⟩] in dfes such that u is an
      unknown initial value whose type is compatible with A) {
  for (each ⟨ecfi1,rc3,loc ⟩ in rc_loc_pairs) {
    if ( loc is v.f1 and v is an unknown initial value compatible with u) {
      if (v != u) {
        rc4 = rc2 and rc3 and ⟨(u neq v), empty, empty⟩;
        new_dfe = [ecfi1,rc4,⟨u.f1,y⟩];
        generated_dfes_pna ∪ {new_dfe};
      }
    }
    else {
      generated_dfes_pna ∪ {dfe};
    }
  }
}
return generated_dfes_pna;
}
```

Figure C.5: auxiliary functions for process-assignment-pta

combination of a left-hand-side location and a right-hand-side location.

Call graph of *process-assignment-pta***:** The call graph of *process-assignment-pta* is shown in Figure C.1. Table C.1 shows the section numbers and figure numbers containing the explanation/pseudocode of the functions invoked by *process-assignment-pta*. If an entry is *terminal*, it means that for simplicity we will not be presenting the pseudocode of the corresponding function. We will only describe the properties of such a function. If an entry is *trivial*, it means the corresponding function is trivial and needs no explanation.

Data structures used by *process-assignment-pta***:** *n* represents an assignment statement, *n.lhs* is the left-hand-side expression of the assignment statement and *n.rhs* is the right-hand-side expression of the assignment statement. *n.lhs-rc-loc-pairs* contains pairs of the form ⟨*ecfi, rc, loc*⟩, where *loc* is a location represented by *n.lhs* (*n.left-hand-side*), and *rc* and *ecfi* are respectively the relevant context and *ECFInfo* under which *n.lhs* represents this location. If *n.lhs* is a variable name, say *p*, then *n.lhs-rc-loc-pairs* is

$\{\langle$*empty,Empty-context,p* $\rangle\}$. The meaning of *n.rhs-rc-loc-pairs* is similar except that it represents values of *n.rhs* (*n.right-hand-side*). If *n.rhs* (*right-hand-side*) is of an arithmetic type, i.e., it does not represent any address, then *n.rhs-rc-loc-pairs* is $\{\langle$*empty, Empty-context, don't-care*$\rangle\}$. *n.lhs-rc-loc-pairs* and *n.rhs-rc-loc-pairs* are incrementally computed by *process-assignment-pta*.

Explanation of the pseudocode using an example: If n is not a pointer assignment statement, then n is a pass-through node for points-to analysis and *process-assignment-pta* propagates *rdfe* to the successor of n without any further processing.

C.1.1 process-new-exception (Figure C.3)

If *rdfe* is a *PTA-dfelm*-**b** and n generates an *initial points-to* (recall n generates an *initial points-to* if and only if *n.lhs* is a user-defined pointer variable and *n.rhs* is *0*), then this *initial points-to* needs to be generated in the new exception context represented by *rdfe*. This is done by the function *process-new-exception* defined in Figure C.3. Recall that such a situation (i.e., a *PTA-dfelm*-**b** reaching a pointer assignment statement) can only happen at a pointer assignment statement directly contained in a finally statement. Also note that only an *initial points-to* needs to be considered for this purpose. If n does not generate an *initial points-to*, then the *ECFInfo* of a generated *PTA-dfelm*-**a** is determined by the *ECFInfo*'s of the *PTA-dfelms*-**a** that give the values of *n.lhs* and *n.rhs*. Only when *n.lhs* is a user-defined variable and *n.rhs* is *0*, the *ECFInfo*'s of the generated *PTA-dfelms*-**a** need to be determined using the reachability information computed by *mark-reachable* and the *PTA-dfelms*-**b** reaching n. As shown earlier, the *PTA-dfelms*-**a** whose *ECFInfo*'s are computed using the reachability information computed by *mark-reachable* are computed by the function *propagate-initial-points-tos-from-assignment-node* defined in Figure B.3. Although the reachability information computed by *mark-reachable* does contain information about exception contexts (i.e., data-flow elements of the form $\langle$*excp-type,reachable*$\rangle$), this information is not used for generating *ECFInfo*'s of the *PTA-dfelms* generated from an *initial points-to* because this information is approximate, as *mark-reachable* represents the signature of an uncaught exception object that is an unknown initial value using the declared type of the unknown initial value rather than the unknown initial value itself. An *ECFInfo* that is the *ECFInfo* of a *PTA-dfelm*-**a** generated from an *initial points-to* at n and that represents an uncaught exception object reaching n, is computed when a *PTA-dfelm*-**b** representing the uncaught exception object reaches n.

C.1.2 Incremental computation of *n.lhs-rc-loc-pairs* and *n.rhs-rc-loc-pairs* (Figure C.2)

As stated above, *n.lhs-rc-loc-pairs* and *n.rhs-rc-loc-pairs* are incrementally computed by *process-assignment-pta*. For example, at statement **4** in Figure 1.5, *lhs-rc-loc-pairs* is initially ϕ. When the *rdfe* [*empty*,*Empty-context*,⟨*a1*,$a1_{init}$⟩] reaches statement **4**, *4.lhs-rc-loc-pairs* is updated to **{**⟨*empty,Empty-context,*$a1_{init}$*.next* ⟩**}**. Similarly, *rhs-rc-loc-pairs* at statement **4** is initially ϕ and when *rdfe* [*empty*,*Empty-context*,⟨*a3*,$a3_{init}$⟩] reaches statement **4**, *4.rhs-rc-loc-pairs* is updated to **{**⟨*empty,Empty-context,*$a3_{init}$ ⟩**}**. Similarly, at statement **6** in Figure 1.5, the final value of *6.lhs-rc-loc-pairs* is **{**⟨*empty,Empty-context,global2* ⟩**}** and the final value of *6.rhs-rc-loc-pairs* is **{**⟨*empty,*⟨*(*$a1_{init}$ **eq** $a2_{init}$*),empty,empty*⟩*,*$a3_{init}$ ⟩,
⟨*empty,*⟨*(*$a1_{init}$ **neq** $a2_{init}$*),empty,empty*⟩*,*$a2_{init}$*.next*$_{init}$ ⟩**}**. This is because statement **4** modifies $a2_{init}$*.next* if and only if $a1_{init}$ and $a2_{init}$ are the same object at the entry of *method1*, and it does not modify $a2_{init}$*.next* if and only if $a1_{init}$ and $a2_{init}$ are not the same object at the entry of *method1*. As a result, the *PTA-dfelms*-**a** [*empty,*⟨*(*$a1_{init}$ **eq** $a2_{init}$*),empty,empty*⟩*,*⟨$a2_{init}$*.next,*$a3_{init}$⟩] and
[*empty,*⟨*(*$a1_{init}$ **neq** $a2_{init}$*),empty,empty*⟩*,*⟨$a2_{init}$*.next,*$a2_{init}$*.next*$_{init}$⟩] reach the top of statement **6**.

C.1.3 get_new_generated_dfes (Figure C.3)

Intuitively, *get_new_generated_dfes* combines the compatible elements from *n.lhs-rc-loc-pairs* and *n.rhs-rc-loc-pairs* to produce the *PTA-dfelms*-**a** generated by the assignment statement. An element *e1* of *n.lhs-rc-loc-pairs* is compatible to an element *e2* of *n.rhs-rc-loc-pairs* if and only if the *ECFInfo*'s of the two elements are the same. The relevant context of a generated *PTA-dfelm*-**a** is the conjunction of the relevant contexts of the elements of *n.lhs-rc-loc-pairs* and *n.rhs-rc-loc-pairs* used for generating the *PTA-dfelm*-**a**. Boundary cases that arise when *n.lhs* is a variable, *n.rhs* is *0* or one of the locations contained in *n.rhs-rc-loc-pairs* is *null* are carefully processed.

C.1.4 generate-points-tos-due-to-pot-aliases

generate-points-tos-due-to-pot-aliases generates *PTA-dfelms*-**a** representing definitions of fields of unknown initial values due to potential aliasing between *compatible* unknown initial values at the entry node of a method. For example, in Figure 1.5, *generate-points-tos-due-to-pot-aliases* generates the *PTA-dfelms*-**a** [*empty*, $\langle$($a1_{init}$ *eq* $a2_{init}$),*empty*,*empty*$\rangle$, $\langle a2_{init}$.*next*,$a3_{init}\rangle$] when *method* is *method2* and *dfe* is [*empty*, *Empty-context*, $\langle a1_{init}$.*next*,$a3_{init}\rangle$].

C.1.5 generate-points-tos-due-to-pot-non-aliases

conditions the propagation of *dfes* across *n* on potential non-aliases between the unknown initial values at the entry node of a method. For example, consider statement **4** in Figure 1.5. Suppose, the *rdfe* [*empty*,*Empty-context*,$\langle a2_{init}$.*next*,$a2.next_{init}\rangle$] reaches program point **4**. Further suppose, for the sake of illustration, $\langle$*empty*,*Empty-context*,$a1_{init}$.*next* $\rangle \in$ *4.lhs-rc-loc-pairs*. As a result, *generate-points-tos-due-to-pot-non-aliases* generates [*empty*, $\langle$($a1_{init}$ *neq* $a2_{init}$),*empty*,*empty*$\rangle$, $\langle a2_{init}$.*next*,$a2.next_{init}\rangle$]

C.1.6 lazily-mark-as-read-field and lazily-mark-as-written-field

lazily-mark-as-read-field (*lazily-mark-as-written-field*) marks as read (written) a field of an unknown initial value that is found to read (written) for the first time. If due to actual-to-uiv bindings, the use of a field in a callee implies the use of another field in a caller in the same SCC as the callee, then the field in the caller is also marked lazily. This is done iteratively until a fixed point is reached. For each field *uiv.f* of an unknown initial value *uiv* marked by *lazily-mark-as-read-field* (*lazily-mark-as-written-field*), *lazily-mark-as-read-field* (*lazily-mark-as-written-field*) propagates [*empty*,*Empty-context*,$\langle$*uiv.f*,$uiv.f_{init}\rangle$] from the entry node of *uiv.method*. Here *uiv.method* is the method with which *uiv* is associated. For each field *uiv.f* of an unknown initial value *uiv*

```
3. process-throw-pta( n, rdfe ) {
  if ( rdfe is a PTA-dfelm-b )
    /*** The exception thrown by n overrides the previous exception represented
         by rdfe.  ***/
    return

  /*** else rdfe is a PTA-dfelm-a [ecfi1,rc1,⟨var1,object1⟩] ***/
  if ( n represents throw var1 )
    /*** i.e., n throws whatever var1 points to ***/
    if ( ⟨ecfi1,rc1,object1 ⟩ ∉ n.rc-excp-object-pairs )
      n.rc-excp-object-pairs = n.rc-excp-object-pairs ∪ ⟨ecfi1,rc1,object1 ⟩
      x = exit node of innermost-enclosing-exception-block(n)
      new-dfe = [rc1,object1]
      add-to-soln-and-worklist-if-needed-I-pta(x,{new-dfe})
      for each [ecfi1,rc2,⟨var2,object2⟩] ∈ n.reaching-PTA-dfelms
        rc3 = rc1 ∧ rc2
        new-dfe = [get-ecfi(object1),rc3,⟨var2,object2⟩]
        add-to-soln-and-worklist-if-needed-I-pta(x,{new-dfe})

  for each ⟨ecfi1,rc2,object2 ⟩ ∈ n.rc-excp-object-pairs
    rc3 = rc1 ∧ rc2
    new-dfe = [get-ecfi(object2),rc3,⟨var1,object1⟩]
    x = exit node of innermost-enclosing-exception-block(n)
    add-to-soln-and-worklist-if-needed-I-pta(x,{new-dfe})
}
```

Figure C.6: Phase I-pta transfer function for throw statement

marked by *lazily-mark-as-read-field*, *lazily-mark-as-read-field* also propagates (if needed) *PTA-dfelms*-**a** representing potential modifications to *uiv.f* at assignment nodes and call nodes contained in *uiv.method*.

C.2 process-throw-pta (Figure C.6)

Purpose of ***process-throw-pta*****:** *process-throw-pta* is the Phase I-pta transfer function of a throw statement. It propagates *PTA-dfelms*-**b** representing the exception objects thrown by the throw statement. Recall that in *RL*, each throw statement is of the form *throw variable;*.

Data structures used by ***process-throw-pta*****:** *n* is an ICFG node that represents a throw statement. Each element of *n.rc-excp-object-pairs* stores an exception object thrown by *n* and the relevant context and *ECFInfo* under which this exception object is thrown by *n*. *rdfe* is a *PTA-dfelm*-**a** [*ecfi1*,*rc1*,⟨*var1*,*object1*⟩] or *rdfe* is a *PTA-dfelm*-**b**.

Explanation of the pseudocode using an example: If *rdfe* is a *PTA-dfelm*-**b** and it represents an uncaught exception, *process-throw-pta* does not propagate *rdfe* any further because the exception thrown by n overrides the exception represented by *rdfe*. Recall that such a situation can only occur at a throw statement directly contained in a finally statement. As stated above, each element of *n.rc-excp-object-pairs* stores an exception object thrown by n and the relevant context and *ECFInfo* under which this exception object is thrown by n. *process-throw-pta* computes the elements of *n.rc-excp-object-pairs* incrementally. Intuitively, for each *PTA-dfelm*-**a** s reaching n, *process-throw-pta* uses the elements of *n.rc-excp-object-pairs* to generate *PTA-dfelms*-**a** whose *ECFInfo*'s store the signatures of the exceptions represented by the elements of *n.rc-excp-object-pairs* and which contain the points-to carried by s. The two *for* loops in Figure C.6 essentially accomplish this.

For example, consider statement **15** in Figure 1.5. Initially *15.rc-excp-object-pairs* is ϕ. Now suppose the *rdfe* [*empty*,*Empty-context*,$\langle$*e1*,e$1_{init}$$\rangle$] reaches statement **15**. As a result, *15.rc-excp-object-pairs* is updated to **{** $\langle$*empty*,*Empty-context*,e1_{init} $\rangle$ **}**. Now, *process-throw-pta* generates the *PTA-dfelm*-**b** [*Empty-context*,e1_{init}] to represent the exception thrown at program point **15**, which is propagated to the exit of *innermost-enclosing-exception-block(15)* , i.e., program point **16**, the exit node of the body of try statement in *method3*. Further, for the sake of illustration, suppose the *PTA-dfelm*-**a** [*empty*,*Empty-context*,$\langle$*global3*,*object*$_{14}$$\rangle$] already $\in$ *15.reaching-PTA-dfelms* . As a result, the first *for* loop in *process-throw-pta* generates [e1_{init},*Empty-context*,$\langle$*global3*,*object*$_{14}$$\rangle$] which is propagated to the exit of *innermost-enclosing-exception-block(15)* . Further, the last *for* loop in *process-throw-pta* generates [e1_{init},*Empty-context*,$\langle$*e1*,e$1_{init}$$\rangle$] which is also propagated to the exit of *innermost-enclosing-exception-block(15)*.

C.3 process-try-entry-pta (Figure C.7)

As stated earlier, a try statement directly contained in a finally statement is treated like a call to an anonymous procedure because it can cause *ECFInfo*'s and exceptions to stack up. If n is the entry node of a try statement directly contained in a finally

4. process-try-entry-pta(n, rdfe) {
```
  if ( innermost-enclosing-exception-block(n.try)  is a finally )
    // try nested inside a finally
    // treat like a call to an anonymous procedure
    process-call-pta( n.call-node, rdfe )
  else
    add-to-soln-and-worklist-if-needed-I-pta(n.successor,{rdfe})
}
```

Figure C.7: Phase I-pta transfer function for entry node of try

```
                       process-try-exit-pta
                              |
     ---------------------------------------------------
     |                       |                        |
     |            propagate-escaped-exception         |
     |                      /\                        |
     |                     /  \                       |
     |                    /  get-escape-tcs           |
     |                   /                            |
   propagate-to-finally-if-needed                     |
             /          \                             |
            /            \                            |
           /              \                           |
  process-method-exit-pta  \                          |
                \           \                         |
                 add-to-soln-and-worklist-if-needed-I-pta
```

Figure C.8: Overview of the call graph of process-try-exit-pta

statement, *n.call-node* is the call node that represents the call to the anonymous procedure.

C.4 process-try-exit-pta (Figure C.9)

Purpose of ***process-try-exit-pta*****:** *process-try-exit-pta* is the Phase I-pta transfer function of the exit node of a try statement. It uses the *ECFInfo* of a reaching *PTA-dfelm*-**a** or the *excp-obj* of a reaching *PTA-dfelm*-**b** to decide the further propagation of the reaching *PTA-dfelm* from the exit node of the try statement.

Call graph of ***process-try-exit-pta*****:** The call graph of *process-try-exit-pta* is shown in Figure C.8. Table C.2 shows the section numbers and figure numbers containing the

5. process-try-exit-pta(n, rdfe) {

```
/*** first case:rdfe is a PTA-dfelm-a [ecfi,rc1,⟨var,val⟩]
      or second case:  rdfe is a PTA-dfelm-b [rc1,excp-obj] ****/

if (first case and ecfi is empty or a label)
  /*** termination without exception ***/
  propagate-to-finally-if-needed( n, rdfe )
else {
  /*** termination due to exception ***/
  /*** either first case with ecfi representing an exception or second case ***/

  /*** find appropriate catches ****/
  catch-entry-nodes = φ
  for each catch c associated with n that can catch
      the exception represented by ecfi or excp-obj
    catch-entry-nodes = catch-entry-nodes ∪ { c.entry }
  for each c-entry ∈ catch-entry-nodes
    add-to-soln-and-worklist-if-needed-I-pta(c-entry,{rdfe})

  /*** Can the exception ESCAPE all the catches?  ***/
  if (the exception represented by ecfi or excp-obj
      can escape all the catches associated with n)
    a:  propagate-escaped-exception(n, rdfe)
}
}
```

5.a. propagate-escaped-exception(n, rdfe) {

```
  let rc1 be ⟨ac1, tc1, etc1⟩
  if (first case:  rdfe represents a pointer's value)
    new-rc = ⟨ac1, tc1, etc1 ∧ get-escape-tcs(n, ecfi)⟩
    new-PTA-dfelm = [ecfi,new-rc,⟨var,val⟩]
  else
    /*** second case:  rdfe represents an exception object ***/
    new-rc = ⟨ac1, tc1, etc1 ∧ get-escape-tcs(n, excp-obj)⟩
    new-PTA-dfelm = [new-rc,excp-obj]

  propagate-to-finally-if-needed( n, new-PTA-dfelm  )
}
```

Figure C.9: Phase I-pta transfer function for exit node of try

Function Name	Sections Containing Explanation/Pseudocode	Figures Containing Pseudocode
propagate-escaped-exception	Sec C.4.1	Fig C.9
get-escape-tcs	Sec C.4.1	terminal
propagate-to-finally-if-needed	Sec C.4.2	Fig C.10
process-method-exit-pta	Sec C.9	Fig C.21
add-to-soln-and-worklist-if-needed-I-pta	Sec 2.8.2.2	Fig 2.27

Table C.2: Guide to functions invoked by process-try-exit-pta

```
propagate-to-finally-if-needed( n, dfe ) {
if ( dfe is a PTA-dfelm-a and dfe.ecfi is empty )
  let dfe be [ecfi,rc1,⟨var,val⟩]
  succ = successor of the try-catch-finally construct associated with n
  dfe = [succ,rc1,⟨var,val⟩]

if ( n has a finally associated with it )
  add-to-soln-and-worklist-if-needed-I-pta( n.finally.entry, {dfe} )
else
  if ( innermost-enclosing-exception-block(n.try) is a finally )
    /*** try nested inside a finally ***/
    /*** treat like a return from an anonymous procedure ***/
    process-method-exit-pta( n.proc-exit, dfe )
    return
  if ((dfe is a PTA-dfelm-a and dfe.ecfi represents an exception) or
      dfe is a PTA-dfelm-b)
    /*** termination due to exception ***/
    x = exit node of innermost-enclosing-exception-block(n.try)
    add-to-soln-and-worklist-if-needed-I-pta( x, {dfe} )
    return

  /*** dfe is a PTA-dfelm-a and dfe.ecfi is a label ***/
  if ( dfe.ecfi is contained in innermost-enclosing-exception-block(n.try) )
    new-dfe = [empty,rc1,⟨var,val⟩]
    add-to-soln-and-worklist-if-needed-I-pta( dfe.ecfi, {new-dfe} )
  else
    x = exit node of innermost-enclosing-exception-block(n.try)
    add-to-soln-and-worklist-if-needed-I-pta( x, {dfe} )
}
```

Figure C.10: auxiliary function of process-try-exit-pta explanation/pseudocode of the functions invoked by *process-try-exit-pta*. If an entry is *terminal*, it means that for simplicity we will not be presenting the pseudocode of the corresponding function. We will only describe the properties of such a function.

Data structures used by ***process-try-exit-pta*****:** n is an ICFG node representing the exit node of a try statement. *n.try* is the try statement whose exit node is n. If the try statement whose exit node is n is directly contained in a finally statement, *n.proc-exit* represents the exit node of the anonymous procedure that corresponds to the try statement. If the try statement whose exit node is n has a finally statement associated with it, *n.finally* represents this finally statement and *n.finally.entry* represents the entry node of this finally statement. For each catch statement c, *c.entry* is the ICFG node representing the entry node of c.

Explanation of the pseudocode using an example: If the try statement terminates without an exception, *process-try-exit-pta* calls *propagate-to-finally-if-needed* to propagate the reaching PTA-*dfelm*-**a** to the entry node of the finally statement (if any) associated with the try statement.

If the try statement terminates due to an exception, *process-try-exit-pta* sequentially searches the list of catch statements associated with the try statement and it uses the *ECFInfo* of *rdfe* or the *excp-obj* contained in *rdfe* (as the case may be) to compute the set of catch statements that can catch the exception encoded in the *rdfe*. If *ecfi* or *excp-obj* is an unknown initial value, there could be more than one catch statement that can potentially catch the exception as the concrete type of the exception could be any subtype of the declared type of the unknown initial value.

A catch statement whose parameter is of type *catch-type* * can catch an exception represented by *ecfi* (*excp-obj*) if and only if either (1) *ecfi* (*excp-obj*) is an *excp-type* (heap-name) and *excp-type* (typeof(excp-obj)) is same as *catch-type* or *excp-type* (typeof(excp-obj)) is a subtype of *catch-type*, or (2) *ecfi* (*excp-obj*) is an unknown initial value and *typeof(ecfi)* (*typeof(excp-obj)*) and *catch-type* are *compatible.*

Consider program point **16** in Figure 1.5. Suppose, at program point **16**, *rdfe* is the PTA-*dfelm*-**a** [$e1_{init}$,*Empty-context*,$\langle$*global3*,$object_{14}\rangle$]. Since *typeof*($e1_{init}$) and *ET2* are compatible, the catch at statement **17** can catch the exception represented by the *ECFInfo* $e1_{init}$. As a result, *catch-entry-nodes* is **{ 17 }** and *rdfe* is propagated to program point **17**.

C.4.1 propagate-escaped-exception (Figure C.9)

An exception represented by *ecfi* (*excp-obj*) can escape all the catches associated with a try block if and only if there does not exist a catch associated with the try block such that type of the parameter of the catch is *catch-type* * and either (1) *ecfi* (*excp-obj*) is an *excp-type* (heap-name) and *excp-type* (typeof(excp-obj)) is same as *catch-type* or *excp-type* (typeof(excp-obj)) is a subtype of *catch-type*, or (2) *ecfi* (*excp-obj*) is an unknown initial value and *typeof(ecfi)* (*typeof(excp-obj)*) is same as *catch-type* or *typeof(ecfi)* (typeof(excp-obj)) is a subtype of *catch-type*. If the exception represented by *ecfi* (*excp-obj*) can escape all the catches associated with the try block, *process-try-exit-pta* calls *propagate-escaped-exception* to propagate the *rdfe* (under appropriate type constraints) to the finally statement (if any) associated with the try statement. If the escaped exception is represented by an unknown initial value (i.e., *ecfi* or *excp-obj* is an unknown initial value), the escaped exception may escape the catch statements only under certain type constraints on the unknown initial value. *get-escape-tcs* returns a conjunction of type constraints that say under what context the exception can escape all the catches associated with a try block.

Continuing the above example, the exception thrown at program point **16** and represented by the *ECFInfo* $e1_{init}$ can escape the catch at program point **17**. *get-escape-tcs(16,* $e1_{init}$ *)* returns *(type(*$e1_{init}$*)* $\not\leq$ *ET2)*. As a result, *new-PTA-dfelm* in *propagate-escaped-exception* is $[e1_{init}, \langle empty, empty, (type(e1_{init}) \not\leq ET2)\rangle, \langle global3, object_{14}\rangle]$.

C.4.2 propagate-to-finally-if-needed (Figure C.10)

If the try block has an associated finally statement, *propagate-to-finally-if-needed* propagates to the entry node of the finally statement. Otherwise, *propagate-to-finally-if-needed (n,dfe)* propagates to an appropriate node in *innermost-enclosing-exception-block(n.try)*. Here *dfe* is a *PTA-dfelm*-**a** or a *PTA-dfelm*-**b**. If *dfe* is a *PTA-dfelm*-**a**,

```
                     process-catch-entry-pta
                              |
    ---------------------------------------------
    |                       |                    |
catch-points-to    catch-exception-object        |
    \                   /                        |
     \                 /                         |
      \               /                          |
       \             /                           |
      get-catch-tcs                              |
                                                 |
                                                 |
              add-to-soln-and-worklist-if-needed-I-pta
```

Figure C.11: Overview of the call graph of process-catch-entry-pta

Function Name	Sections Containing Explanation/Pseudocode	Figures Containing Pseudocode
catch-points-to	Sec C.5.1	Fig C.12
catch-exception-object	Sec C.5.2	Fig C.12
get-catch-tcs	Sec C.5.1, C.5.2	terminal
add-to-soln-and-worklist-if-needed-I-pta	Sec 2.8.2.2	Fig 2.27

Table C.3: Guide to functions invoked by process-catch-entry-pta

dfe.ecfi represents the *ECFInfo* of *dfe*. Exit from a try statement directly contained in a finally statement to the finally statement is treated like a return from an anonymous procedure and *n.proc-exit* represents the exit node of this anonymous procedure. In the above example, since there is no finally statement associated with program point **16**, when *propagate-to-finally-if-needed* is called from *process-escaped-exception* with $n = \mathbf{16}$ and $dfe = [e1_{init}, \langle empty, empty, (type(e1_{init}) \not\leq ET2) \rangle, \langle global3, object_{14} \rangle]$, *propagate-to-finally-if-needed* propagates *dfe* to program point **19**.

C.5 process-catch-entry-pta (Figure C.12)

Purpose of *process-catch-entry-pta***:** *process-catch-entry-pta* is the Phase I-pta transfer function of the entry node of a catch statement. If *rdfe* represents a value of a pointer (i.e., *rdfe* is a *PTA-dfelm*-**a**), *process-catch-entry-pta* catches the exception represented

```
/*** first case:   rdfe is a PTA-dfelm-a [ecfi,rc1,⟨var,val⟩]
          or
second case:  rdfe is a PTA-dfelm-b [rc1,excp-obj] ***/
/*** let rc1 be ⟨ac1, tc1, etc1⟩***/
```

6. process-catch-entry-pta(n, rdfe) {

```
/*** n is the entry node of a catch statement that
  catches any exception whose type is catch-type or a subtype of catch-type ***/

if (first case)
  /*** rdfe represents a pointer's value ***/
  a:  new-dfe = catch-points-to(n, rdfe)
else
  /*** second case:  rdfe represents an exception object ***/
  b:  new-dfe = catch-exception-object(n, rdfe)

add-to-soln-and-worklist-if-needed-I-pta(n.successor, {new-dfe})
}
```

---

a. catch-points-to(n, rdfe) {

```
if (rdfe is [excp-type,rc1,⟨var,val⟩])
  /*** process-try-exit-pta has ensured that exp-type is either same
      as the catch-type or it is a subtype of the catch-type ***/
  new-rc = rc1
else
  /*** rdfe.ecfi is an unknown initial value uiv.  process-try-exit-pta has ensured
      that typeof(uiv) and the catch-type are compatible ****/
  new-rc = ⟨ac1, tc1, etc1 ∧ get-catch-tcs(n, uiv)⟩

new-dfe = [empty,new-rc,⟨var,val⟩]
return new-dfe }
```

---

b. catch-exception-object(n, rdfe) {

```
if ( rdfe.excp-obj is a heap-name )
  new-rc = rc1
else
  /*** rdfe.excp-obj is an unknown initial value ***/
  new-rc = ⟨ac1, tc1, etc1 ∧ get-catch-tcs(n, rdfe.excp-obj)⟩

  /*** param is the parameter of the catch ***/
  new-dfe = [empty,new-rc,⟨param,rdfe.excp-obj⟩]
return new-dfe }
```

Figure C.12: Phase I-pta transfer function for catch entry by the *ECFInfo* of *rdfe* and propagates a new *PTA-dfelm*-**a** whose *ECFInfo* is *empty* (meaning the exception has been caught) and which contains the points-to of the *rdfe*. If *rdfe* represents an exception object, *process-catch-entry-pta* generates a *PTA-dfelm*-**a** representing a value of the parameter of the catch statement because the parameter is assigned the caught exception object.

Call graph of *process-catch-entry-pta*: The call graph of *process-catch-entry-pta* is shown in Figure C.11. Table C.3 shows the section numbers and figure numbers containing the explanation/pseudocode of the functions invoked by *process-catch-entry-pta*. If an entry is *terminal*, it means that for simplicity we will not be presenting the pseudocode of the corresponding function. We will only describe the properties of such a function.

Data structures used by *process-catch-entry-pta*: n is an ICFG node representing the entry node of a catch statement that catches any exception whose type is *catch-type* or a subtype of *catch-type*. *rc1*, the relevant context of *rdfe*, is $\langle$ac1, tc1, etc1$\rangle$. If *rdfe* is a *PTA-dfelm*-**a**, *rdfe.ecfi* is the *ECFInfo* of the *PTA-dfelm*-**a**. If *rdfe* is a *PTA-dfelm*-**b**, *rdfe.excp-obj* is the exception object represented by the *PTA-dfelm*-**b**.

Explanation of the pseudocode using an example:

C.5.1 catch-points-to (Figure C.12)

If *rdfe* is a *PTA-dfelm*-**a**, *process-catch-entry-pta* calls *catch-points-to* to process the reaching *PTA-dfelm*-**a**. *rdfe.ecfi* is either an *excp-type* or an unknown initial value. If *rdfe.ecfi* is an *excp-type*, *process-try-exit-pta* has ensured that *exp-type* is either same as the *catch-type* or it is a subtype of the *catch-type*. If *rdfe.ecfi* is an unknown initial value *uiv*, *process-try-exit-pta* has ensured that *typeof(uiv)* and the *catch-type* are *compatible.* If *ECFInfo* is an unknown initial value, *catch-points-to* calls *get-catch-tcs* to generate the appropriate type constraints on the concrete types of the unknown initial value under which the exception represented by the *ECFInfo* is caught by the catch statement.

7. process-catch-exit-pta(n, rdfe) {
```
  if ( rdfe is PTA-dfelm-a and rdfe.var is parameter of the catch )
    /*** no need to propagate further as var is local to the catch ***/
    return
  propagate-to-finally-if-needed( n, rdfe )
}
```

Figure C.13: Phase I-pta transfer function for exit node of a catch

For example, consider program point **17** in Figure 1.5. Suppose the *rdfe* [$e1_{init}$,*Empty-context*,⟨*global3*,$object_{14}$⟩] reaches this program point. The *ECFInfo* of this *PTA-dfelm*-**a** is $e1_{init}$ and hence *get-catch-tcs* returns *(type(*$e1_{init}$*)* $\leq$ *ET2)*. As a result, *new-dfe* (in *catch-points-to*) is the *PTA-dfelm*-**a** [*empty*,⟨*empty,empty,(type(*$e1_{init}$*)* $\leq$ *ET2)*⟩,⟨*global3*,$object_{14}$⟩].

C.5.2 catch-exception-object (Figure C.12)

If *rdfe* is a *PTA-dfelm*-**b**, *process-catch-entry-pta* calls *catch-exception-object* to process the reaching *PTA-dfelm*-**b**. *rdfe.excp-obj* is either a heap-name or an unknown initial value. If *rdfe.excp-obj* is a heap-name, *process-try-exit-pta* has ensured that the type of the heap-name is either same as the *catch-type* or it is a subtype of the *catch-type*. If *rdfe.excp-obj* is an unknown initial value, *process-try-exit-pta* has ensured that *typeof(rdfe.excp-obj)* and the *catch-type* are *compatible*. Like *catch-points-to*, if *rdfe.excp-obj* is an unknown initial value, *catch-points-to* calls *get-catch-tcs* to generate the appropriate type constraints on the concrete types of the unknown initial value under which the exception is caught by the catch statement. *catch-exception-object* generates a *PTA-dfelm*-**a** representing the assignment of the caught exception object to the parameter of the catch statement.

Again consider program point **17** in Figure 1.5. Suppose the *rdfe* [*Empty-context*,$e1_{init}$] reaches this program point. *rdfe.excp-obj* is $e1_{init}$ and again *get-catch-tcs* returns *(type(*$e1_{init}$*)* $\leq$ *ET2)*. As a result, *new-dfe* (in *catch-exception-object*) = [*empty*,⟨*empty,empty,(type(*$e1_{init}$*)* $\leq$ *ET2)*⟩,⟨*excp*,$e1_{init}$⟩] is generated to represent a value of the parameter of the catch statement.

```
                  process-finally-exit-pta
                           /       |
                          /        |
                         /         |
                        /          |
      process-method-exit-pta      |
                                   |
                     add-to-soln-and-worklist-if-needed-I-pta
```

Figure C.14: Overview of the call graph of process-finally-exit-pta

Function Name	Sections Containing Explanation/Pseudocode	Figures Containing Pseudocode
process-method-exit-pta	Sec C.9	Fig C.21
add-to-soln-and-worklist-if-needed-I-pta	Sec 2.8.2.2	Fig 2.27

Table C.4: Guide to functions invoked by process-finally-exit-pta

C.6 process-catch-exit-pta (Figure C.13)

process-catch-exit-pta is the Phase I-pta transfer function of the exit node of a catch statement. If *rdfe* represents a value of the parameter of the catch statement, *process-catch-exit-pta* does not propagate *rdfe* any further because the parameter is not visible outside the catch statement. Otherwise, *process-catch-exit-pta* calls *propagate-to-finally-if-needed* to propagate *rdfe* to the entry node of the finally statement (if any) associated with the catch statement. The pseudocode for *propagate-to-finally-if-needed* is given in Figure C.10 and it is explained in Section C.4.2.

C.7 process-finally-exit-pta (Figure C.15)

Purpose of *process-finally-exit-pta***:** *process-finally-exit-pta* is the Phase I-pta transfer function of the exit node of a finally statement.

Call graph of *process-finally-exit-pta***:** The call graph of *process-finally-exit-pta* is shown in Figure C.14. Table C.4 shows the section numbers and figure numbers containing the explanation/pseudocode of the functions invoked by *process-finally-exit-pta*.

Data structures used by *process-finally-exit-pta***:** n is an ICFG node that represents

8. process-finally-exit-pta(n, rdfe) {

```
/*** n.try is the try statement associated with n ***/
if ( innermost-enclosing-exception-block(n.try) is a finally )
  /*** try nested inside a finally ***/
  /*** treat like a return from an anonymous procedure ***/
  process-method-exit-pta( n.proc-exit, rdfe )
  return
if ((rdfe is a PTA-dfelm-a and rdfe.ecfi represents an exception) or
    rdfe is a PTA-dfelm-b)
  /*** termination due to exception ***/
  x = exit node of innermost-enclosing-exception-block(n.try)
  add-to-soln-and-worklist-if-needed-I-pta( x, {rdfe} )
  return

/*** rdfe is a PTA-dfelm-a [ecfi,rc1,⟨var,val⟩] and ecfi is a label ***/
if ( ecfi is contained in innermost-enclosing-exception-block(n.try) )
  new-dfe = [empty,rc1,⟨var,val⟩]
  add-to-soln-and-worklist-if-needed-I-pta( ecfi, {new-dfe} )
else
  x = exit node of innermost-enclosing-exception-block(n.try)
  add-to-soln-and-worklist-if-needed-I-pta( x, {rdfe} )
```

}

Figure C.15: Phase I-pta transfer function for exit node of a finally

the exit node of a finally statement. *n.try* is the try statement associated with the finally statement whose exit node is *n*. If *n.try* is directly contained in a finally statement, *n.proc-exit* represents the exit node of the anonymous procedure representing *n.try*.

Explanation of the pseudocode using an example: As stated earlier, if the try statement with which the finally statement is associated is directly contained in a finally statement, then the exit from the finally statement is treated like a return from the anonymous procedure representing the nested try statement. Thus, in this situation, *process-finally-exit-pta* calls *process-method-exit-pta*, the Phase I-pta transfer function of the exit node of a method, to process the *rdfe*. If *rdfe* is a *PTA-dfelm-***b** or *rdfe* is a *PTA-dfelm-***a** and the *ECFInfo* of the *PTA-dfelm-***a** represents an exception, *process-finally-exit-pta* propagates *rdfe* to the exit node of the innermost exception block containing *n.try*. If *rdfe* is a *PTA-dfelm-***a** and the *ECFInfo* of the *PTA-dfelm-***a** represents a pending transition, then *process-finally-exit-pta* either propagates *rdfe* to the target of the pending transition or to the exit node of the innermost exception block containing *n.try*, depending upon whether the target of the pending transition lies within the

Function Name	Sections Containing Explanation/Pseudocode	Figures Containing Pseudocode
func1	Sec C.8.1	terminal
lazily-mark-used-fields	Sec C.8.2	Fig C.18
lazily-mark-as-read-field	Sec C.1.6	terminal
lazily-mark-as-written-field	Sec C.1.6	terminal
back-propagate	Sec C.8.3	Fig C.18
back-instantiate	Sec C.9	Fig C.21
func5	Sec C.8.4	Fig C.19
func6	Sec C.8.4	Fig C.19
add-to-soln-and-worklist-if-needed-I-pta	Sec 2.8.2.2	Fig 2.27

Table C.5: Guide to functions invoked by process-call-pta

innermost exception block containing *n.try* or not. If the target of the pending transition lies within the innermost exception block containing *n.try*, then *process-finally-exit-pta* generates a new *PTA-dfelm*-**a** with *empty ECFInfo*, signifying that the pending transition has been completed.

Consider program point **43**, the exit node of the nested finally statement in *method6* defined in Figure 1.5. Suppose *rdfe* is the *PTA-dfelm*-**a** [*44*,*Empty-context*,⟨*local*,$local_{init}$⟩]. Recall that entry to a try statement directly nested inside a finally statement is treated like a call to an anonymous procedure. Here $local_{init}$ is the unknown initial value of *local* at program point **38**, which is treated like the entry node of an anonymous procedure. Since *innermost-enclosing-exception-block(43.try)* is a finally statement, *process-method-exit-pta* is called to propagate *rdfe* from *43.proc-exit*, which represents the exit node of the anonymous procedure.

Next consider program point **44**, the exit node of the outer finally statement in *method6*. Suppose *rdfe* is the *PTA-dfelm*-**a** [*ET1*,*Empty-context*,⟨*local*,$object_{34}$⟩]. Now *innermost-enclosing-exception-block(44.try)* is the body of *method6*. Thus *rdfe* is propagated to program point **45**, the exit node of *innermost-enclosing-exception-block(44.try)* .

C.8 process-call-pta (Figure C.17)

Purpose of *process-call-pta***:** *process-call-pta* is the Phase I-pta transfer function

```
                    process-call-pta
                           |
    --------------------------------------------------
    |              |                 |        |       |
  func1   lazily-mark-used-fields  func4     func5   func6
                 /    \              |         |       |
                /      \             |         |       |
             func2   func3           |         |       |
                                     |         |       |
                          back-instantiate     |       |
                                               |       |
                      add-to-soln-and-worklist-if-needed-I-pta
```

```
Here func1 is compute-new-actual-to-uiv-bindings,
     func2 is lazily-mark-as-read-field,
     func3 is lazily-mark-as-written-field,
     func4 is back-propagate,
     func5 is propagate-across-call-PTA-dfelm-a
and  func6 is propagate-across-call-PTA-dfelm-b.
```

Figure C.16: Overview of the call graph of process-call-pta

10. process-call-pta(n, rdfe) {

```
  if ( rdfe is a PTA-dfelm-a )
    // each actual-to-uiv binding has the form [ecfi,rc,⟨actual,uiv-at-the-entry-of-a-target⟩]
    new-actual-to-uiv-bindings = compute-new-actual-to-uiv-bindings( n, rdfe )
    lazily-mark-used-fields( new-actual-to-uiv-bindings )
    back-propagate( n, new-actual-to-uiv-bindings )
    if (rdfe.var is n.lhs)
      /*** rdfe is killed as the result of the call is stored in var ***/
      return
    if (rdfe.var is a local, a field of a heap-name or a field of an
        unknown initial value )
      propagate-across-call-PTA-dfelm-a( n, rdfe )
    else // rdfe.var is a global
      if ( rdfe.var ∉ the used-set of at least one method invocable from n in
              the initial call graph )
        propagate-across-call-PTA-dfelm-a( n, rdfe )
  else // rdfe is a PTA-dfelm-b and it represents an exception object
    propagate-across-call-PTA-dfelm-b( n, rdfe )
```

}

Figure C.17: Phase I-pta transfer function of call node

```
lazily-mark-used-fields(new-actual-to-uiv-bindings) {
  for each [ecfi,rc,⟨actual,uiv⟩] ∈ new-actual-to-uiv-bindings
    if ( actual is an unknown initial value )
      for each used field f of uiv
        if ( uiv.f is marked as read and actual.f has not been marked as read )
          lazily-mark-as-read-field( actual.f )
        if (uiv.f is marked as written and actual.f has not been marked as written)
          lazily-mark-as-written-field( actual.f )
}
```

```
back-propagate(n, new-actual-to-uiv-bindings) {
  for each [ecfi,rc,⟨actual,uiv⟩] ∈ new-actual-to-uiv-bindings
    e = exit node of uiv.method
    for each dfe ∈ e.reaching-PTA-dfelms  such that dfe contains uiv
      if ( dfe is a PTA-dfelm-b or (dfe is a PTA-dfelm-a and dfe.var is not a
            local variable of uiv.method))
        back-instantiate( dfe, n, e, [ecfi,rc,⟨actual,uiv⟩] )
    a:  if ( uiv is the receiver and n is a dynamically dispatched call site )
      for each dfe ∈ e.reaching-PTA-dfelms  not containing uiv
        if ( dfe is a PTA-dfelm-b or (dfe is a PTA-dfelm-a and dfe.var is not a
                local variable of uiv.method))
          back-instantiate( dfe, n, e, [ecfi,rc,⟨actual,uiv⟩] )
}
```

Figure C.18: Auxiliary functions of process-call-pta

```
propagate-across-call-PTA-dfelm-a( n, dfe ) {
/*** dfe is PTA-dfelm-a [ecfi,rc1,⟨var,val⟩] ***/
  for each target t of n in the initial call graph
    for each ⟨tag,reachable⟩ computed by mark-reachable at exit of t
      if (tag is empty)
        add-to-soln-and-worklist-if-needed-I-pta( n.successor.successor, { dfe} )
      if (tag is a label contained in innermost-enclosing-exception-block(n))
        add-to-soln-and-worklist-if-needed-I-pta( tag, { dfe} )

      if (tag is a label not contained in innermost-enclosing-exception-block(n) )
        new-dfe = [tag,rc1,⟨var,val⟩]
        add-to-soln-and-worklist-if-needed-I-pta( n.successor, { new-dfe} )

  for each ⟨rc2,excp-obj1⟩ ∈ n.rc-excp-object-pairs
    rc3 = rc1 ∧ rc2
    new-dfe = [get-ecfi(excp-obj1),rc3,⟨var,val⟩]
    add-to-soln-and-worklist-if-needed-I-pta( n.successor, { new-dfe} )
}
```

```
propagate-across-call-PTA-dfelm-b( n, dfe ) {
/*** dfe is PTA-dfelm-b [rc1,excp-object] ***/
  for each target t of n
    for each ⟨tag,reachable⟩ computed by mark-reachable at exit of t
      if (tag is empty)
        add-to-soln-and-worklist-if-needed-I-pta( n.successor.successor, { dfe} )
      if (tag is a label contained in innermost-enclosing-exception-block(n) )
        add-to-soln-and-worklist-if-needed-I-pta( tag, { dfe} )
    if ( n is a statically dispatched call site )
      for each y ∈ t.exit-node.reaching-PTA-dfelms such that y
            does not contain any unknown initial value
        if ( y is a PTA-dfelm-a and (y.ecfi is empty or a label) and y.var is not a
              local variable of t )
          let y be [ecfi,Empty-context,⟨var,val⟩]
          z = [get-ecfi(excp-object),Empty-context,⟨var,val⟩]
          if (y.ecfi is empty)
            add-to-soln-and-worklist-if-needed-I-pta( n.successor.successor, { z} )
          if (y.ecfi is a label contained in innermost-enclosing-exception-block(n) )
            add-to-soln-and-worklist-if-needed-I-pta( y.ecfi, { z} )
}
```

Figure C.19: Auxiliary functions of process-call-pta of a call node. The corresponding call site could be either statically dispatched or dynamically dispatched. *process-call-pta* incrementally computes the bindings between actuals at the call site and the unknown initial values at the entry nodes of the targets of the call site. For each new binding between an actual at the call site and an unknown initial value at the entry node of a target of the call site, *process-call-pta* instantiates those *PTA-dfelms* that contain the unknown initial value and that are in the solution at the the exit node of the target (i.e., a *PTA-dfelm* in the pointer summary transfer function of the target) with the actuals at the call site, and propagates the instantiated *PTA-dfelms*. *process-call-pta* may or may not propagate the information in a reaching *rdfe* across the call site. In case the information in a *rdfe* is propagated across the call site, a part of the information (such as *ECFInfo*) may change reflecting the effect of the call site. If *rdfe* is a *PTA-dfelm*-**a**, the decision as to whether to propagate the information in *rdfe* across the call site depends on such factors as whether *rdfe.var* is a local, a field of an object or a global. If *rdfe* is a *PTA-dfelm*-**b**, the decision as to whether to propagate the information in *rdfe* across the call site depends on such factors as whether the call can terminate without an exception.

Call graph of ***process-call-pta*****:** The call graph of *process-call-pta* is shown in Figure C.16. Table C.5 shows the section numbers and figure numbers containing the explanation/pseudocode of the functions invoked by *process-call-pta*. If an entry is *terminal*, it means that for simplicity we will not be presenting the pseudocode of the corresponding function. We will only describe the properties of such a function.

Data structures used by ***process-call-pta*****:** n is an ICFG node representing a call site. Recall that a call site is represented by a pair of nodes: a call node and a return-site which is the successor of the call node. *n.lhs* is the variable (if any) in which the result of the call is stored. The *used-set* of a method contains the set of globals used during the life-time of the method and it is formally defined in Section B.2. For each unknown initial value *uiv*, *uiv.method* is the method with respect to whose entry node *uiv* is defined. For every *PTA-dfelm*-**a** y, *y.ecfi* is the *ECFInfo* of y and *y.var* is the location whose value y represents. For each target t of n, *t.exit-node* is the exit node of the target.

Explanation of the pseudocode using an example:

C.8.1 compute-new-actual-to-uiv-bindings

compute-new-actual-to-uiv-bindings computes new bindings between actuals at n and unknown initial values at the entry nodes of targets of n, implied by *rdfe*. Recall each binding has the form [*ecfi*,*rc*,⟨*actual,uiv-at-the-entry-of-a-target*⟩], where *ecfi* and *rc* are respectively an *ECFInfo* and relevant context under which the binding holds. *ecfi* will be non-empty only at a call site directly contained in a finally. For example, consider call site **12** in Figure 1.5. Suppose *rdfe* is [*empty*,*Empty-context*,⟨q,$object_{11}$⟩]. As a result, the following two new actual-to-uiv bindings are computed:
[*empty*,*Empty-context*,⟨$object_{11}$,$a2_{init}$⟩]
[*empty*,*Empty-context*,⟨$object_{11}$,$a3_{init}$⟩]. Next, consider call site **9** in Figure 1.5. Suppose *rdfe* is [*empty*,*Empty-context*,⟨$a5$,$a5_{init}$⟩]. As a result, the following new actual-to-uiv binding is computed:
[*empty*,*Empty-context*,⟨$a5_{init}$,$a2_{init}$⟩]. Whenever *compute-new-actual-to-uiv-bindings* computes a new binding between an actual a and an unknown initial value u at the entry node of a target, *compute-new-actual-to-uiv-bindings* recursively computes bindings between the values of the fields of a (computed so far at the call node) and the unknown initial values of the corresponding fields of u (found to be used so far).

C.8.2 lazily-mark-used-fields (Figure C.18)

lazily-mark-used-fields (Figure C.18) marks fields of actuals that are found be read/written for the first time due to an actual-to-uiv binding. For example, consider call site **9** in Figure 1.5. Suppose *rdfe* is [*empty*,*Empty-context*,⟨$a5$,$a5_{init}$⟩]. As a result, $a5_{init}.next$ is marked as read because $a2_{init}.next$ has been marked as read and there is a new new actual-to-uiv binding [*empty*,*Empty-context*,⟨$a5_{init}$,$a2_{init}$⟩].

C.8.3 back-propagate (Figure C.18)

back-propagate (Figure C.18) propagates those new instantiations of *PTA-dfelms* at the exit node of a target of n that are implied by new actual-to-uiv bindings at n. For example, consider call site **9** in Figure 1.5. Suppose *rdfe* is [*empty*,*Empty-context*,⟨*a6*,$a6_{init}$⟩]. As a result, there is a new actual-to-uiv binding [*empty*,*Empty-context*,⟨$a6_{init}$,$a3_{init}$⟩]. Further suppose for the sake of illustration, [*empty*,*Empty-context*,⟨$a4_{init}$,$a1_{init}$⟩] already ∈ *9.actual-to-uiv-bindings*. Now [*empty*,*Empty-context*,⟨$a1_{init}$*.next*,$a3_{init}$⟩] ∈ *method1.exit-node.reaching-PTA-dfelms* . As a result, *back-propagate* propagates [*empty*,*Empty-context*,⟨$a4_{init}$*.next*,$a6_{init}$⟩] to program point **10**. The details of *back-instantiate (Figure C.21)* are explained later. If n is a dynamically dispatched call site and *uiv* of a new actual-to-uiv-binding is the receiver of the called method, then the *PTA-dfelms* at the exit node of the target need to be propagated with new type constraints. This is accomplished by the last if statement at program point **a**.

C.8.4 propagate-across-call-*PTA-dfelm*-a (Figure C.19) and propagate-across-call-*PTA-dfelm*-b (Figure C.19)

If *rdfe* needs to be propagated across *n*, *propagate-across-call-PTA-dfelm*-**a** *(Figure C.19)* and *propagate-across-call-PTA-dfelm*-**b** *(Figure C.19)* are used to propagate *rdfe* across the call site with appropriate *ECFInfo*'s. There is efficiency-vs-precision tradeoff in deciding when to propagate a *rdfe* across the call site. Clearly the simplest solution is to propagate every *rdfe* across the call site. This will result in a safe solution, but the precision may not be good. We have made our choices based on the following philosophy: our choices enable us to compute the precise solution in those cases for which points-to analysis is polynomial-time solvable and for those cases for which points-to analysis is not polynomial-time solvable, we have made the simplest choices that enable us to compute a safe solution with reasonable precision.

The *rdfe* is not propagated across n if one of the following situations hold:

1. *rdfe* is a *PTA-dfelm*-**a** and *rdfe.var* is *n.lhs*, i.e., the result of the call is stored in *rdfe.var*.

2. *rdfe* is a *PTA-dfelm*-**a** and *rdfe.var* is a global variable that is contained in the *used-set* of all the methods invocable from n.

3. *rdfe* is a *PTA-dfelm*-**b** and the call always results in an exit from the *innermost-enclosing-exception-block(n)* (due to an uncaught exception thrown by the call or an incomplete, pending transition generated by the call).

The *rdfe* needs to be propagated across n if one of the following situations hold:

1. *rdfe* is a *PTA-dfelm*-**a** and *rdfe.var* is a local variable (provided the result of the call is not stored in this variable) of the method containing the node n. This is because a local variable cannot be modified by the call (recall that in *RL* the address of a local variable cannot be stored in a pointer).

2. *rdfe* is a *PTA-dfelm*-**a** and *rdfe.var* is a field of an unknown initial value or a heap-name. This is because Points-to-Algo treats the fields of unknown initial values and heap-names in a flow-insensitive manner at a call site. We have made this choice for simplicity, it is easy to extend Points-to-Algo to treat the fields of unknown initial values and heap-names in a flow-sensitive manner across a call site.

3. *rdfe* is a *PTA-dfelm*-**a** and *rdfe.var* is a global variable that is not contained in the *used-set* of at least one method potentially invocable (according to hierarchy analysis) from the call site. If the global variable is not contained in the *used-set* of any of the methods invocable from the call site, then the global variable cannot be modified by the call (recall that in *RL* the address of a global variable cannot be stored in a pointer). In this situation there is no loss in precision by propagating *rdfe* across n. However, if the global variable is not contained in the *used-sets* of some of the methods invocable from n but the global variable is contained in the *used-sets* of other methods invocable from n, then the global variable may be modified by the call. In this situation, there may be a loss in precision by propagating *rdfe* across n. As before, we have made this choice for simplicity, it is easy to extend Points-to-Algo to treat the global variables in a flow-sensitive manner even in the latter situation. Our choice enables Points-to-Algo to compute the precise solution in polynomial-time solvable cases (see Chapter 3).

4. *rdfe* is a *PTA-dfelm*-**b** and there is a potential target of *n* (according to hierarchy analysis) such that there is a path from the entry of the target to the exit of the target such that when this path is followed either (1) there is no uncaught exception or pending transition at the exit of the target or (2) there is a pending transition to a statement contained in *innermost-enclosing-exception-block(n)*.

The solution computed by *mark-reachable* is used to determine if the *ECFInfo* of *rdfe* is preserved across *n*. The *ECFInfo* of *rdfe* is preserved only if the call is not terminated due a reason that causes exit from *innermost-enclosing-exception-block(n)*. For each exception thrown by the call at *n*, *propagate-across-call-PTA-dfelm*-**a** propagates a new *PTA-dfelm*-**a**, whose *ECFInfo* stores the signature of the exception. For example, consider program point **35** in Figure 1.5. Suppose *rdfe* is [*empty*,*Empty-context*,$\langle$*local*,*object*$_{34}\rangle$]. The only data-flow-element computed by *mark-reachable* at the exit node of *method5* is $\langle$*ET1*,*reachable*$\rangle$. As a result, the *ECFInfo* of *rdfe* is not preserved across program point **35**. Further, $\langle$*Empty-context*,*object*$_{31}\rangle \in$ *35.rc-excp-object-pairs*. As a result, [*ET1*,*Empty-context*,$\langle$*local*,*object*$_{34}\rangle$] is propagated to the return-site of the call at program point **35**.

The reason why at some places in *propagate-across-call-PTA-dfelm*-**a** and *propagate-across-call-PTA-dfelm*-**b**, *dfe* is propagated to *n.successor.successor* instead of *n.successor* is to distinguish between the *ECFInfo*'s and exceptions caused by the call and the *ECFInfo*'s and exceptions reaching the top of the call site (which can only happen at a call site directly contained in a finally statement). This will be explained further in Section C.9.2.

C.9 process-method-exit-pta (Figure C.21)

Purpose of *process-method-exit-pta***:** *process-method-exit-pta* is the Phase I-pta transfer function of the exit node of a method. For each call site that is in the current SCC and that has been found to invoke the method whose exit node is being considered, *process-method-exit-pta* instantiates *rdfe* using the actual-to-uiv bindings at the call site and propagates the resulting instantiations to the call site.

```
                       process-method-exit-pta
                                 |
                                 |
                          back-instantiate
                           /     /  \    \
                          /     /    \    \
                         /     /      \ generate-points-tos-due-to-
                   back-bind  /        \            -pot-aliaes
                             /          \
propagate-due-to-new-rc-excp-object-pair \
                                          \
                                     propagate-to-call-site
                                              |
                                              |
                                              |
                               add-to-soln-and-worklist-if-needed-I-pta
```

Figure C.20: Overview of the call graph of process-method-exit-pta

Function Name	Sections Containing Explanation/Pseudocode	Figures Containing Pseudocode
back-instantiate	Sec C.9.1	Fig C.21
back-bind	Sec C.9.1.1	terminal
propagate-due-to-new-rc-excp-object-pair	Sec C.9.1.2	terminal
generate-points-tos-due-to-pot-aliases	Sec C.1.4	Fig C.4
propagate-to-call-site	Sec C.9.2	Fig C.21
add-to-soln-and-worklist-if-needed-I-pta	Sec 2.8.2.2	Fig 2.27

Table C.6: Guide to functions invoked by process-method-exit-pta

```
/*** first case:   rdfe is PTA-dfelm-a [ecfi,rc1,⟨var,val⟩]
      or
  second case:  rdfe is PTA-dfelm-b [rc1,excp-obj] ***/
```
11. process-method-exit-pta(n, rdfe) {
```
/*** n is the exit node of the method n.method ***/

if (first case and var is a local variable of n.method)
  /*** no need to propagate to callers ***/
  return

for each call site c in the current SCC that has been found to invoke n.method {
  back-instantiate( rdfe, c, n, all )
} }
```

---

back-instantiate(rdfe, c, n, actual-to-uiv-binding) {
```
  x = y = z = φ

  x = back-bind( rdfe, c, n, actual-to-uiv-binding )
  /*** back-bind returns [ecfi1,rc3,⟨var1,val1⟩]'s or [rc3,excp-obj1]'s to which rdfe
    maps at c ***/

  /*** update the list exceptions thrown by the call ***/
  if (second case)
    for each [rc3,excp-obj1] ∈ x
      if ( ⟨rc3,excp-obj1⟩ ∉ c.rc-excp-object-pairs
        c.rc-excp-object-pairs = c.rc-excp-object-pairs ∪ { ⟨rc3,excp-obj1⟩ }
        propagate-due-to-new-rc-excp-object-pair( c, ⟨rc3,excp-obj1⟩ )

  if (first case and val is not var_init )
    /*** i.e., the points-to has been generated during the life-time of n.method
    ***/
    y = generate-points-tos-due-to-pot-aliases(c.method, x)

  z = x ∪ y

  propagate-to-call-site( c, z, rdfe)
}
```

---

propagate-to-call-site(c, instantiated-dfes, dfe) {
```
  if (dfe is a PTA-dfelm-a and
        (dfe.ecfi is empty or dfe.ecfi is a label
          contained in innermost-enclosing-exception-block(c)))
    if ( dfe.ecfi is empty )
      add-to-soln-and-worklist-if-needed-I-pta(c.successor.successor, instantiated-dfes)
    else
      add-to-soln-and-worklist-if-needed-I-pta(dfe.ecfi, instantiated-dfes)
  else // the call overrides the active ecfi's and exceptions before the call
    add-to-soln-and-worklist-if-needed-I-pta(c.successor, instantiated-dfes)
}
```

Figure C.21: Phase I-pta transfer function of exit node of a method

Call graph of *process-method-exit-pta***:** The call graph of *process-method-exit-pta* is shown in Figure C.20. Table C.6 shows the section numbers and figure numbers containing the explanation/pseudocode of the functions invoked by *process-method-exit-pta*. If an entry is *terminal*, it means that for simplicity we will not be presenting the pseudocode of the corresponding function. We will only describe the properties of such a function.

Data structures used by *process-method-exit-pta***:** *n* is the exit node *n.method*. Note that *n.method* is either a user-defined method or an anonymous procedure representing a try statement directly contained in a finally statement. For each call node *c*, *c.rc-excp-object-pairs* contains the set of exceptions thrown by the call. Recall that each element of *c.rc-excp-object-pair* contains an exception object and a relevant context under which the exception object is thrown.

Explanation of the pseudocode using an example: If *rdfe* represents the value of a local variable of *n.method*, *process-method-exit-pta* does not propagate *rdfe* to the call sites of *n.method*. Otherwise, for each call site that is in the current SCC and that has been found to invoke *n.method*, *process-method-exit-pta* calls *back-instantiate* to instantiate *rdfe* using the actual-to-uiv bindings at the call site and propagate the resulting instantiations to the call site.

C.9.1 back-instantiate (Figure C.21)

C.9.1.1 back-bind

back-bind(dfe, c, n, actual-to-uiv-binding) uses the actual-to-uiv bindings in *c.actual-to-uiv-bindings* to instantiate the unknown initial values in *dfe* with their values at *c*. *back-bind* returns the set of *PTA-dfelms* resulting from the instantiations. The parameter *actual-to-uiv-binding* is an actual-to-uiv-binding or it is the key word *all*. If *actual-to-uiv-bindings* is *all*, *back-bind* instantiates each unknown initial value in *dfe* with all its bindings contained in *c.actual-to-uiv-bindings*. Otherwise, for the *uiv* in *actual-to-uiv-binding*, *back-bind* uses only *actual-to-uiv-binding* for instantiating *uiv* and for instantiating all other unknown initial values, *back-bind* uses all their bindings in *c.actual-to-uiv-bindings*. For example, consider the *rdfe* $[ET1, Empty\text{-}context, \langle global1, param_{init}\rangle]$

at program point **33** in Figure 1.5. *back-bind(rdfe, 35, 33)* returns $\{[ET1,Empty\text{-}context,\langle global1,object_{34}\rangle]\}$.

C.9.1.2 propagate-due-to-new-rc-excp-object-pair

If *rdfe* represents an exception object (i.e., second case), the list of exceptions thrown by a call to *n.method* may need to be updated. If *rdfe* implies a new pair y of relevant context and exception object at a call site c of *n.method* in the current SCC, *propagate-due-to-new-rc-excp-object-pair(c, y)* is called to propagate *PTA-dfelms*-**a** $\in$ *c.reaching-PTA-dfelms* that need to be propagated across c with new *ECFInfo* and relevant context implied by y (as done by the last for statement in *propagate-across-call-PTA-dfelm*-**a** defined in Figure C.19).

C.9.2 propagate-to-call-site (Figure C.21)

A try statement or call directly contained in a finally statement can cause exceptions and *ECFInfo*'s to stack up, and pending exceptions and *ECFInfo*'s can exist at a call or entry to a try statement nested inside a finally statement. As discussed earlier, the stack of *ECFInfo*'s is maintained implicitly by storing them in actual-to-uiv bindings. As a result, a *PTA-dfelm*-**a** with *empty ECFInfo* can be instantiated by *back-bind* into a *PTA-dfelm*-**a** with non-empty *ECFInfo*, using the *ECFInfo* of an actual-to-uiv binding used for instantiation. Thus, we need to distinguish between an *ECFInfo* that is caused by the call and an *ECFInfo* that results from instantiation. This is done by *propagate-to-call-site(c, instantiated-dfes, dfe) (Figure C.21).* The transfer function of a return-site of a call node (shown in Figure C.24) expects the *ECFInfo* of a *rdfe* passed as input to the transfer function to be caused by the call. Thus, if the *ECFInfo* is caused by instantiation, *propagate-to-call-site* by-passes the return-site of c (because the return-site of a call node expects all *ECFInfo*'s to be caused by the call) and propagates the instantiated *PTA-dfelm*-**a** to the successor of the return-site of c. This is the reason why the solution of each return-site is augmented in Figure 2.24. Since exit from a try statement directly nested inside a finally statement is treated like a return from a call to an anonymous procedure, *dfe.ecfi* can be a label.

```
/*** first case:  rdfe is PTA-dfelm-a [ecfi,rc1,⟨var,val⟩]
             or
second case:  rdfe is PTA-dfelm-b [rc1,excp-obj] ***/
```

12. process-break-pta(n, rdfe) {

```
let y = target of break
if (y is contained in innermost-enclosing-exception-block(n) )
  add-to-soln-and-worklist-if-needed-I-pta(y,{rdfe})
else
  if ( first case )
    new-dfe = [y,rc1,⟨var,val⟩]
    x = exit of innermost-enclosing-exception-block(n)
    add-to-soln-and-worklist-if-needed-I-pta(x,{new-dfe})
  // else second case, need not propagate as the exception is overridden by
  // the break statement
}
```

---

13. process-continue-pta(n, rdfe) {

```
let y = target of continue
if (y is contained in innermost-enclosing-exception-block(n) )
  add-to-soln-and-worklist-if-needed-I-pta(y,{rdfe})
else
  if ( first case )
    new-dfe = [y,rc1,⟨var,val⟩]
    x = exit of innermost-enclosing-exception-block(n)
    add-to-soln-and-worklist-if-needed-I-pta(x,{new-dfe})
  // else second case, need not propagate as the exception is overridden by
  // the continue statement
}
```

Figure C.22: Phase I-pta transfer functions for break and continue statements

C.10 Other Phase I-pta transfer functions

This section presents the pseudocode for the remaining kinds of ICFG nodes. The Phase I-pta transfer functions of a a node representing a break statement, a node representing a catch statement, a node representing a return statement, a return-site and an object creation site are given in Figures C.22 to C.25. We skip the explanation of the pseudocode as most of the pseudocode is self-explanatory and the concepts used in this pseudocode have already been explained in the preceding sections.

```
/*** first case:  rdfe is PTA-dfelm-a [ecfi,rc1,⟨var,val⟩]
            or
second case:  rdfe is PTA-dfelm-b [rc1,excp-obj] ***/
```

14. process-return-statement-pta(n, rdfe) {

```
if ( first case )
  y = exit of n.method
  if ( innermost-enclosing-exception-block(n)  is body of n.method )
    /*** ecfi must be empty ***/
    add-to-soln-and-worklist-if-needed-I-pta(y,{rdfe})
    return
  /*** else innermost-enclosing-exception-block(n)  is the body of a try,
      catch or finally ***/
  new-dfe = [y,rc1,⟨var,val⟩]
  x = exit of innermost-enclosing-exception-block(n)
  add-to-soln-and-worklist-if-needed-I-pta(x,{new-dfe})
/*** else second case, need not propagate as the exception is overridden by
    the return statement ***/
}
```

Figure C.23: Phase I-pta transfer function for return statement

```
/*** first case:  rdfe is PTA-dfelm-a [ecfi,rc1,⟨var,val⟩]
            or
second case:  rdfe is PTA-dfelm-b [rc1,excp-obj] ***/
```

15. process-return-site-pta(n, rdfe) {

```
if ( first case )
  if ( ecfi is a label )
    if ( ecfi is not contained in innermost-enclosing-exception-block(n)  )
      x = exit of innermost-enclosing-exception-block(n)
      add-to-soln-and-worklist-if-needed-I-pta(x,{rdfe})
    else
      new-dfe = [empty,rc1,⟨var,val⟩]
      add-to-soln-and-worklist-if-needed-I-pta(ecfi,{new-dfe})
    return
  if ( ecfi is empty )
    add-to-soln-and-worklist-if-needed-I-pta(n.successor,{rdfe})
    return
  // else ecfi represents an exception
  x = exit of innermost-enclosing-exception-block(n)
  add-to-soln-and-worklist-if-needed-I-pta(x,{rdfe})
else
  x = exit of innermost-enclosing-exception-block(n)
  add-to-soln-and-worklist-if-needed-I-pta(x,{rdfe})
}
```

Figure C.24: Phase I-pta transfer function for return-site

16. process-new-node-pta(n, rdfe) {

```
if ( rdfe is a PTA-dfelm-a [ecfi,rc1,⟨var,val⟩] and var is n.lhs )
  /*** rdfe is killed, need not propagate further ***/
  return
add-to-soln-and-worklist-if-needed-I-pta(n.successor,{rdfe})
if ( rdfe is a PTA-dfelm-b )
  /*** rdfe carries a new exception context to n.  So the points-tos propagated
    by propagate-values-of-pointer-fields-of-newly-created-object need to be
    propagated in this new exception context.  ***/
  let rdfe be [rc1,excp-obj]
  /*** propagate newly created object ***/
  y = [get-ecfi(excp-obj),rc1,⟨n.lhs,object_n⟩]
  add-to-soln-and-worklist-if-needed-I-pta(n.successor, {y})
  /*** propagate values of pointer fields of newly created object ***/
  for each pointer field f of object_n
    y = [get-ecfi(excp-obj),rc1,⟨object_n.f,null⟩]
    add-to-soln-and-worklist-if-needed-I-pta(n.successor, {y})
}
```

Figure C.25: Phase I-pta transfer function for object creation site

Appendix D

Demand Driven Computation

As stated in Chapter 2, Points-to-Algo generates fields of unknown initial values lazily, only when they are found to be used. Similarly, it only considers those globals that are used during the lifetime of a method. During Phase III-pta, at a call node *C*, for each points-to *pt* that represents a value of the field of a heap-name at *C*, the tuple $\langle pt, C\rangle$ is stored in a global table called *DemandTable*. Similarly, for each points-to *pt* that represents a value of a global variable that is not used in at least one of the methods invocable from *C*, the tuple $\langle pt, C\rangle$ is stored in *DemandTable*. The following example[1] shows how Points-to-Algo 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;
};

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-pta computes [*empty*,*Empty-context*,$\langle p, e1_{init}.field1_{init}\rangle$]

[1] For clarity this example is not written in *RL*.

and [*empty*,*Empty-context*,$\langle e2_{init}.field1, e2_{init}.field1_{init}\rangle$] at program points 6 and 7 respectively; and $e1_{init}.field2_{init}$ and $e2_{init}.field2_{init}$ do not appear in the Phase I-pta solutions of *foo0* and *foo1* respectively. As a result, during Phase II-pta, the value of $e2_{init}.field2_{init}$ at the call site 4, which is $object_3$, is not propagated to the entry node of *foo1*. Instead, the tuple $\langle\langle object_1.field2,\ object_3\rangle,\ 4\rangle$ is stored in *DemandTable*. If the value of $e1_{init}.field2_{init}$ is needed at program point *5*, then, since $e1_{init}$ is $object_1$, Points-to-Algo looks up *DemandTable* for the values of $object_1.field2$. The result of this lookup says that $object_1.field2$ points to $object_3$ at the call site 4. Now Points-to-Algo checks if the entry node of *foo0* is reachable from call site 4. Since it is reachable, $object_3$ is a valid value for $object_1.field2$ at program point *5*.

D.1 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, Points-to-Algo 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 m, it computes the set of methods (say *reach-set*) reachable from m and relevant contexts, in terms of the unknown initial values of m, under which these methods are reachable. Points-to-Algo 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 *reach-sets* of the methods invocable from C. The relevant contexts of the elements of these *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 values computed at the entry node of m. The set of methods reachable from C is determined by those elements whose relevant contexts evaluate to *true* or whose relevant contexts are instantiated into relevant contexts that involve the interface initial values of a single *public interface method* and that can evaluate to true by making conservative, worst-case assumptions about these interface initial values.

Consider the example given in Figure 1.5. The final call graph has edges from call site **5** to *A::update* and *C::update*. Under this assumption, the *reach-set* of *method1*, say *rs*, is

$\{ \langle\langle empty, (type(a1_{init}) \in A::update), empty\rangle, A::update\rangle,$
$\langle\langle empty, (type(a1_{init}) \in C::update), empty\rangle, C::update\rangle \}$

Now, suppose the set of methods reachable from call site **9**, in *method2*, is needed. The only method invocable from call site **9**, 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 **9**, i.e., $a1_{init}$ is replaced by $a4_{init}$. This yields the set *rs1*:

$\{ \langle\langle empty, (type(a4_{init}) \in A::update), empty\rangle, A::update\rangle,$
$\langle\langle empty, (type(a4_{init}) \in C::update), empty\rangle, C::update\rangle \}$

Now $a4_{init}$ is an interface initial value and the relevant contexts of both the elements of *rs1* can evaluate to true by making conservative, worst-case assumption about $a4_{init}$. Thus the set of methods reachable from call site **9** is:

{ *method1, A::update, C::update* }.

Next, suppose the methods reachable from call site **12** are needed. Now, $a1_{init}$ in the relevant contexts of the elements of *rs* is instantiated by $object_{10}$. 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 **12** is

{ *method1, A::update* }.

Appendix E

Safety and Complexity of Points-to-Algo

This appendix presents outlines of proofs of safety and complexity of Points-to-Algo. We show that for programs written in *RL*, Points-to-Algo computes a safe solution and it is is polynomial-time under reasonable assumptions.

E.1 Definitions

This section presents some definitions needed to explain the proofs of safety and complexity of Points-to-Algo.

execution-path-from-a-start-node(r,e,s,l): Let l be a program point and s be a start-node. execution-path-from-a-start-node(r,e,s,l) denotes an execution path r from the start-node s to l, starting with environment e at s.

execution-path-from-entry-node(r,e,M,l): Let l be a program point in a method M. execution-path-from-entry-node(r,e,M,l) denotes an execution path r from the entry-node of M to l, starting with environment e at the entry-node of M. Here M can be an anonymous procedure that represents a *try-catch-finally* directly contained in a finally statement. If M represents an anonymous procedure, then it is the innermost such procedure containing l.

execution-path-from-entry-node-without-exception-and-label(r,e,M,l): Let l be a program point in a method M. execution-path-from-entry-node-without-exception-and-label(r,e,M,l) denotes an execution-path-from-entry-node(r,e,M,l) such that when r is executed, at l, (1) there is no uncaught exception thrown [1] during the execution of r

[1] Due to a call or try statement nested inside a finally statement, there may be active, uncaught exceptions at the entry node of M.

and (2) there is no incomplete, pending transition that is generated during the execution of r and that is due to a break statement, a continue statement, a return statement or falling through a try statement or catch statement associated with a finally statement.

execution-path-from-entry-node-with-exception(r,e,M,l,excp-type): Let l be a program point in a method M.

execution-path-from-entry-node-with-exception(*r,e,M,l,excp-type*) denotes an execution-path-from-entry-node(*r,e,M,l*) such that when r is executed, at l, there is an uncaught exception object (1) that is thrown during the execution of r, (2) that is allocated during the execution of r and (3) whose type is *excp-type*.

execution-path-from-entry-node-with-exception(r,e,M,l,iv): Let l be a program point in a method M and let *iv* be an initial value in e. execution-path-from-entry-node-with-exception(*r,e,M,l,iv*) denotes a an execution-path-from-entry-node(*r,e,M,l*) such that when r is executed, at l, there is an uncaught exception object *iv* that is thrown during the execution of r.

execution-path-from-entry-node-with-label(r,e,M,l,label): Let l be a program point in a method M. execution-path-from-entry-node-with-label(*r,e,M,l,label*) denotes a an execution-path-from-entry-node(*r,e,M,l*) such that when r is executed, at l, there is an incomplete, pending transition to statement numbered *label* and (1) this pending transition is generated during the execution of r and (2) this pending transition is due to a break statement, a continue statement, a return statement or falling through a try statement or a catch statement associated with a finally statement.

static-representation-of(*v,e,r*) : Let e be an environment at the entry node of a method M and r be an execution path starting from the entry node of M. If v is a user-defined variable, static-representation-of(*v,e,r*) is v. If v is an initial value in e, static-representation-of(*v,e,r*) is the corresponding unknown initial value, which could be a *representative* unknown initial value in the presence of recursive types. If v is a heap allocated object that is allocated during the execution of r at program point n, static-representation-of(*v,e,r*) is $object_n$. If v is *x.f*, static-representation-of(*v,e,r*) is *static-representation-of(x,e,r).f*.

Lemma 13 *Let l be a program point in a method M, let m be the entry node of M, let s be a start-node and let r be an execution-path-from-a-start-node(r,e,s,l). Then there exists a unique decomposition of r such that (1) $r = r1 + r2$ (+ indicates concatenation), (2) r1 is an execution-path-from-a-start-node(r1,e,s,m), (3) r2 is an execution-path-from-entry-node(r2,e1,M,l) and (4) e1 is the initial environment at m when r1 is executed.*

proof The proof is by straight forward induction on the length of r. □

E.2 Safety of *mark-reachable*

This section shows that for programs written in *RL*, *mark-reachable* computes a safe solution. Here by precise reachability we mean the following: a node l contained in a method M is said to be reachable from the entry node of M if and only if there exists an execution-path-from-entry-node(r,e,M,l). We show that if there exists an execution-path-from-entry-node(r,e,M,l), then l is marked reachable by *mark-reachable*.

Lemma 14 *Suppose there exists an execution-path-from-entry-node(r,e,M,l). The following hold with respect to it.*

1. *If r is an execution-path-from-entry-node-without-exception-and-label(r,e,M,l), then ⟨empty,reachable⟩ is computed by mark-reachable at l.*
2. *If r is an execution-path-from-entry-node-with-exception(r,e,M,excp-type), then ⟨excp-type,reachable⟩ is computed by mark-reachable at l.*
3. *If r is an execution-path-from-entry-node-with-exception(r,e,M,uiv), then ⟨typeof(uiv),reachable⟩ is computed by mark-reachable at l.*
4. *If r is an execution-path-from-entry-node-with-label(r,e,M,label), then ⟨label,reachable⟩ is computed by mark-reachable at l.*

proof The 4 claims can be proved simultaneously by straight forward induction on the length of r. □

Theorem 14 *mark-reachable computes a safe reachability solution for programs written in RL.*

proof: Immediate from Lemma 14. □

E.3 Safety of Phase I-pta

Again, to prove that phase I-pta is safe, we need to define what we mean by *precision* of this phase for programs written in *RL*. Let l be a program point in a method M. Let

e be an environment at the entry node of M. Let v be a variable of pointer type that is read at l. Then $\langle v, object \rangle$ belongs to the precise Phase I-pta solution at l if and only if there exists an execution-path-from-entry-node(r,e,M,l) such that if r is executed, at l, v points to *object*. Here v could be a local, a global, a field of a heap allocated object that is allocated during the execution of r or a field of an initial value in e. *object* is either a heap allocated object that is allocated during the execution of r or it is an initial value in e.

Lemma 15 *Suppose there exists an execution-path-from-entry-node(r,e,M,l). Let var be one of the following: (a) a parameter of M of pointer type that is read in M, (b) a global variable of pointer type that is in the used set of M (defined in Figure B.3), (c) a local variable of pointer type of M, (d) a pointer type field of a heap allocated object that is allocated during the execution of r or (e) a pointer type field of an initial value in e that is read either in M or in a method invoked during the lifetime of M. Let val be one of the following: (a) a heap allocated object that is allocated during the execution of r or (b) an initial value in e. Further suppose when r is executed, at l, var points to val. The following hold with respect to r.*

1. *If the execution-path-from-entry-node(r,e,M,l) is the execution-path-from-entry-node-without-exception-and-label(r,e,M,l), then* [*empty*,*rc*,⟨*var1*,*val1*⟩] *is computed by Phase I-pta at l.*
2. *If the execution-path-from-entry-node(r,e,M,l) is the execution-path-from-entry-node-with-label(r,e,M,l,label), then* [*label*,*rc*,⟨*var1*,*val1*⟩] *is computed by Phase I-pta at l.*
3. *If the execution-path-from-entry-node(r,e,M,l) is the execution-path-from-entry-node-with-exception(r,e,M,l,excp-type), then* [*excp-type*,*rc*,⟨*var1*,*val1*⟩] *is computed by Phase I-pta at l.*
4. *If the execution-path-from-entry-node(r,e,M,l) is the execution-path-from-entry-node-with-exception(r,e,M,l,iv), then* [*static-representation-of(iv,e,r)* ,*rc*,⟨*var1*,*val1*⟩] *is computed by Phase I-pta at l.*

Where rc is a relevant context which is true in e, var1 is static-representation-of(var,e,r) and val1 is static-representation-of(val,e,r).

proof The 4 claims can be proved simultaneously using straight forward induction on the length of r.

Theorem 15 *Points-to-Algo computes a safe Phase I-pta solution for programs written in RL.*

proof: Immediate from Lemma 15. □

E.4 Safety of Phase II-pta

To prove that phase II-pta is safe, first we need to define what we mean by *precision* of this phase for programs written in *RL*. Let m be the entry-node of a method *M*. Let *uiv* be an unknown initial value at the entry node of *M* such that there exists an execution path t starting from m, beginning with environment e at m, and when t is executed, *uiv* is accessed. Then $\langle$*uiv,object*$\rangle$ belongs to the precise Phase II-pta solution at m if and only if there exists a start-node s and an execution-path-from-a-start-node(*r,e1,s,m*) such that if r is executed, at m, *uiv* is *object*. Here *object* is either a heap allocated object that is allocated during the execution of r or it is an initial value in *e1*.

Lemma 16 *Let m be the entry-node of a method M. Let uiv be an unknown initial value at the entry node of M such that there exists an execution path t starting from m, beginning with environment e at m, and when t is executed, uiv is accessed. Let s be a start-node. Suppose there exists an execution-path-from-a-start-node(r,e1,s,m). Let object be either a heap allocated object that is allocated during the execution of r or an initial value in e1. Then the following hold with respect to r.*

1. *If when r is executed, at m, uiv is object, then* [*rc*,*static-representation-of(uiv,e,t)*,*static-representation-of(object,e1,r)*] *is computed by Phase II-pta of Points-to-Algo at m. Here rc is a relevant context which is true in e1.*

2. *M is marked reachable by Phase II-pta.*

proof The above 2 claims can be proved simultaneously by using Theorem 15 and straight forward induction on the length of r. □

Theorem 16 *Points-to-Algo computes a safe Phase II-pta solution for programs written in RL.*

proof: Immediate from Lemma 16. □

E.5 Safety of Phase III-pta

To prove that phase III-pta is safe, first we need to define what we mean by *precision* of this phase for programs written in *RL*. Let l be a program point in a method *M*. Let x be a location that is read at l. $\langle$*x,object*$\rangle$ belongs to the precise Phase III-pta solution at l if and only if there exists a start-node s and an execution-path-from-a-start-node(*r,e,s,l*) such that if r is executed, at l, x is read and x points to *object*. Here (1) x is a user-defined pointer variable or (2) x is a field of a pointer type of a heap-allocated object that is allocated during the execution of r and the expression used for reading x at l is $v \rightarrow f$ where v is a user-defined variable of pointer type or (3) x is a field of a pointer type of an initial value in e and the expression used for reading x at l is $v \rightarrow f$ where v is a user-defined variable of pointer type. Here *object* is either a heap allocated object that is allocated during the execution of r or it is an initial value in e.

Lemma 17 *Let l be a program point in a method M. Let m be the entry node of M. Let v be an expression of pointer type that is read at l. Here v is either a user-defined variable or v is $x \rightarrow f$ where x is a user-defined variable. Suppose there exists a start-node s and an execution-path-from-a-start-node(r,e,s,l) such that if r is executed, at l, v points to object. Then ⟨v,static-representation-of(object,e,r) ⟩ is computed by Phase III-pta of Points-to-Algo. Here object is either a heap allocated object that is allocated during the execution of r or it is an interface initial value in e.*

proof Using Lemma 13, $r = r1 + r2$, where *r1* is an execution-path-from-a-start-node(r1,e,s,m) and *r2* is an execution-path-from-entry-node(r2,e1,M,l). Here *e1* is the environment at m when *r1* is executed.

Case 1: v is a user-defined variable.

Suppose *object* is a heap object that is allocated during the execution of *r2*. Thus, when *r2* is executed, at l, v points to static-representation-of(*object,e1,r2*) and static-representation-of(*object,e1,r2*) is same as static-representation-of(*object,e,r*). Then using Using Lemma 15, [*ecfi*,*rc*,⟨*v,static-representation-of(object,e1,r2)* ⟩] ∈ *l.reaching-PTA-dfelms*. Here *rc* is a relevant context true in *e1*. Thus, using Lemma 16, there exists an instantiation of unknown initial values in *rc* with their concrete values computed by Phase II-pta, for which *rc* evaluates to true. Hence Phase III-pta computes ⟨*v,static-representation-of(object,e1,r2)*⟩ at program point l.

Suppose when *r1* is executed, at m, static-representation-of(*object,e,r1*) is the concrete value of a unknown initial value *uiv* and when *r2* is executed, at l, v points to *uiv*. Thus static-representation-of(*object,e,r1*) is same as static-representation-of(*object,e,r*). Now, using Lemma 16, [*rc1*,*static-representation-of(object,e,r2)* ,*uiv*] ∈ *M.cv-to-uiv-bindings*, and using Lemma 15, [*ecfi*,*rc*,⟨*v,uiv*⟩] ∈ *l.reaching-PTA-dfelms*.

Here *rc* is a relevant context true in *e1*. Thus, using Lemma 16, there exists an instantiation of unknown initial values in ⟨*rc,uiv*⟩ with their concrete values computed by Phase II-pta, for which *rc* can evaluate to true and *uiv* is instantiated with static-representation-of(*object,e,r1*). As a result, Phase III-pta computes ⟨*v,static-representation-of(object,e,r1)*⟩ at program point *l*.

Case 2: *v* is *x*→*f* and *x* is a user-defined variable.

Suppose when *r* is executed, at *l*, *x* points to *object1*. Using the proof for case 1, ⟨*x,static-representation-of(object1,e,r)*⟩ is computed by Phase III-pta.

Let *static-representation-of(object1,e1,r2).f* be *z*.

Suppose *object* is a heap object that is allocated during the execution of *r2*. Thus, when *r2* is executed, at *l*, *z* points to static-representation-of(*object,e1,r2*) and static-representation-of(*object,e1,r2*) is same as static-representation-of(*object,e,r*). Using Lemma 15, [*ecfi*,*rc*,⟨*z,static-representation-of(object,e1,r2)* ⟩] ∈ *l.reaching-PTA-dfelms*. Here *rc* is a relevant context true in *e1*. Thus, using Lemma 16, there exists an instantiation of unknown initial values in ⟨*rc,z*⟩ with their concrete values computed by Phase II-pta, for which *rc* can evaluate to true and *z* is instantiated to *static-representation-of(object1,e,r).f*. Hence Phase III-pta computes ⟨*static-representation-of(object1,e,r).f,static-representation-of(object,e1,r2)*⟩ at program point *l*. Combining the values of *x* and *static-representation-of(object1,e,r).f*, we get *v* points to static-representation-of(*object,e1,r2*) at *l*.

Suppose when *r1* is executed, at *m*, static-representation-of(*object,e,r1*) is the concrete value of an unknown initial value *uiv* and when *r2* is executed, at *l*, *z* points to *uiv*. Thus static-representation-of(*object,e,r1*) is same as static-representation-of(*object,e,r*). Now, using Lemma 16, [*rc1*,*static-representation-of(object,e,r1)* ,*uiv*] ∈ *M.cv-to-uiv-bindings*, and using Lemma 15, [*ecfi*,*rc2*,⟨*z,uiv*⟩] ∈ *l.reaching-PTA-dfelms*. Here *rc2* is a relevant context true in *e1*. Thus, using Lemma 16, there exists an instantiation of unknown initial values in $\langle$*rc2,uiv*$\rangle$ with their concrete values computed by Phase II-pta, for which *rc2* evaluates to true, *uiv* is instantiated with static-representation-of(*object,e,r1*) and z is instantiated to *static-representation-of(object1,e,r).f*. Hence Phase III-pta computes $\langle$*static-representation-of(object1,e,r).f,static-representation-of(object,e,r1)*$\rangle$ at program point l. Combining the values of x and *static-representation-of(object1,e,r).f*, we get v points to static-representation-of(*object,e,r1*) at l. □

Theorem 17 *Points-to-Algo computes a safe Phase III-pta solution for programs written in RL.*

proof: Immediate from Lemma 17. □

E.6 Complexity of Points-to-Algo

In this section, we briefly discuss the complexity of the various steps of Points-to-Algo for programs written in *RL*. We will focus on the important, dominating terms and, for simplicity, ignore less important terms. We will show that under reasonable assumptions Points-to-Algo is polynomial-time for programs written in *RL*.

We will use the following notations:

- Let $\underline{T_{max}}$ be the maximum number of targets of a call site in the initial call graph,
- let $\underline{N_c}$ be the total number of call nodes,
- let $\underline{N_{proc}}$ be the total number of procedures/methods,
- let the total number of statements in the program be n_1,
- let the sum of the numbers of arguments passed at call sites be n_2,
- let $\underline{N_{nodes}}$ be $n_1 + n_2$,
- let the total number of method exit nodes be E,
- let the total number of exception types be $\underline{N_{excp-type}}$,

- let the total number of exception objects be $\underline{N_{excp-obj}}$ (identified by their creation sites),
- let the maximum number of labels in a method be N_{label},
- let the maximum number of catch statements associated with a try statement be N_{catch},
- let the total number of unknown initial values generated by Points-to-Algo be $\underline{N_{uiv}}$,
- let the number of user-defined pointer variables be $\underline{N_{var}}$,
- let the maximum number of fields of a class (including inherited fields) be $\underline{F_{max}}$,
- let the total number of classes be $\underline{N_{class}}$,
- let the total number of heap-names be $\underline{N_h}$,
- let the bound on the number of conjuncts in an alias context be $\underline{k1}$,
- let the bound on the number of conjuncts in a type context be $\underline{k2}$,
- let the bound on the number of conjuncts in an exception type context be $\underline{k3}$, and
- let l be a program point in a method M.

E.6.1 Complexity of Phase 0

The complexity of Phase 0 is $O(T_{max}N_c + N_{proc})$.

E.6.2 Complexity of Phase I-pta

First we consider the complexity of *mark-reachable*. The total number of possible data-flow elements (computed by *mark-reachable*) at a node is $O(N_{excp-type} + N_{label} + 1)$ because each data-flow element has the form $\langle$*ecfi,reachable*$\rangle$, where *ecfi* is an exception type, a label or *empty*. At each node, except a call node, the exit node of a method or the exit node of a try statement, constant amount of work is done for each new data-flow element reaching that node. At each exit node of a try statement, at most

$O(N_{catch})$ amount of work is done in propagating a data-flow element to the apropriate catch statements. For each data-flow element that needs to be propagated along an interprocedural edge, at most $O(N_{label}+1)$ amortized amount of work is done. Thus the worst-case complexity of *mark-reachable* is $O(N_{nodes}*(N_{excp-type}+N_{label}+1)N_{catch}+C(N_{excp-type}+N_{label}+1)(N_{label}+1))$

The scheme for dealing with recursive types ensures that the total number of unknown initial values and hence the total number of possible *PTA-dfelms* is finite, even if no bound is imposed on the number of conjuncts in a relevant context. Since Points-to-Algo does only a finite amount of work for each *PTA-dfelm* at a program point and at each step Points-to-Algo considers a new *PTA-dfelm* at a program point, Points-to-Algo always terminates.

Let $\underline{N_{rc}}$ be the number of possible relevant contexts. N_{rc} is at most $O(pa^{k1}tc^{k2}etc^{k3})$, where $pa = 2N_{uiv}^2$, $tc = N_{uiv}T_{max}$ and $etc = 2N_{uiv}N_{excp-type}$. Here pa is an upper bound on the number of possible potential aliases and potential non-aliases, tc is an upper bound on the number of possible type constraints that can be contained in a type context and etc is an upper bound on the number of possible type constraints that can be contained in an *exception type context*.

Let $\underline{N_{pt}}$ be the number of possible *points-tos*. 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.

Let N_{ecfi} be the number of possible *ECFInfo*'s. N_{ecfi} is at most $O(N_{excp-type}+N_{uiv}+1)$.

Now, let $\underline{N_{pdfea}}$ be the total number of possible *PTA-dfelms*-**a**. N_{pdfea} is at most $O(N_{ecfi}N_{rc}N_{pt})$.

Now, let $\underline{N_{pdfeb}}$ be the total number of possible *PTA-dfelms*-**b**. N_{pdfeb} is at most $O(N_{rc}(N_{excp-obj}+N_{uiv}))$.

Hence the total number of possible *PTA-dfelms* is polynomial in N_{uiv}, N_{var}, N_h,

$N_{excp-obj}$, $N_{excp-type}$, N_{label}, T_{max} and F_{max}, assuming $k1, k2, k3$ are constants.

Now consider the work done by Points-to-Algo at a pointer assignment node. For each *PTA-dfelm*-**b** reaching such a node, a constant amount of work is done. For each *PTA-dfelm*-**a** reaching such a node, $O(nl * nr)$, where $\{nl = N_{rc} * N_{ecfi} * (N_{uiv} + N_h)$, $nr = nl\}$, is an upper bound on the work done for *PTA-dfelms*-**a** directly generated by the pointer assignment statement. Here *nl* is an upper bound on the number of elements in *lhs_rc_loc_pairs* (see Figure C.2) and similarly, *nr* is an upper bound on the number of elements in *rhs_rc_loc_pairs*. $O(N_{uiv} * cs)$, where $\{cs = N_{rc}N_{uiv}(N_{uiv} + N_h)\}$, is an upper bound on the work done in generating *PTA-dfelms*-**a** due to potential aliases. Here *cs* is an upper bound on the number of *dfelms* in *new_generated_dfes* (see Figure C.2) that may generate *PTA-dfelms*-**a** due to potential aliases. Finally, $O(nl * cs)$ is an upper bound on the work done in generating *dfelms* due to potential non-aliases. Here *cs* is an upper bound on the number of *PTA-dfelms*-**a** in the current solution of the pointer assignment node that may generate *PTA-dfelms*-**a** due to potential non-aliases.

Similarly, it can be shown that at any node, for each *PTA-dfelm* reaching that node, the work done by Points-to-Algo is polynomial in $N_{uiv}, N_h, N_{var}, T_{max}, N_{excp-type}, N_{label}$, $N_{excp-obj}$ and F_{max} (assuming $k1, k2, k3$ are constants and a constant bound on the number of intraprocedural successors of a node). Let $\underline{N_{work}}$ be the maximum amount of work done by Points-to-Algo for a *PTA-dfelm* at a program point.

Since at each step Points-to-Algo considers a new *PTA-dfelm* at a program point, the total amount of work done by Points-to-Algo is $O(N_{pdfe} * N_{nodes} * N_{work})$. Hence the total work done by Points-to-Algo is polynomial in $N_{uiv}, N_h, N_{var}, T_{max}, F_{max}, N_{excp-type}$, $N_{label}, N_{excp-obj}$ and N_{nodes}, assuming $k1, k2, k3$ are constants.

$N_h, N_{var}, T_{max}, F_{max}, N_{excp-type}, N_{label}, N_{excp-obj}$ and N_{nodes} are obviously bounded by the size of the program. However, in theoretically contrived cases, Points-to-Algo can generate an exponential number of unknown initial values. Figure E.1 shows such a case. At program point s in Figure E.1, p_n can point to an exponential number of unknown initial values. We never encountered such cases in practice. In principle this exponential cases can be easily avoided by enforcing a bound t on the lengths of *access*

class A_1 {
  public: A_2 $*f_1$;
  public: A_2 $*f_2$;
};
...
class A_i {
  public: A_{i+1} $*f_1$;
  public: A_{i+1} $*f_2$;
};
...
class A_{n-1} {
  public: A_n $*f_1$;
  public: A_n $*f_2$;
};
class A_n { };

void method(A_1 *a) {
  A_1 $*p_1$;
  ...
  A_i $*p_i$;
  ...
  A_n $*p_n$;
  p_1 = a;
  if (_) $p_2 = p_1 \rightarrow f_1$;
    else $p_2 = p_1 \rightarrow f_2$;
  ...
  if (_) $p_i = p_{i-1} \rightarrow f_1$;
    else $p_i = p_{i-1} \rightarrow f_2$;
  ...
  if (_) $p_n = p_{n-1} \rightarrow f_1$;
    else $p_n = p_{n-1} \rightarrow f_2$;
  s:
}

Figure E.1: Exponential no. of unknown initial values

paths of unknown initial values (analogous to *k-limiting* [JM82a]). 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 $N_{proc}, N_{var}, F_{max}$, N_{class} and t.

E.6.3 Complexity of Phase II-pta

Let $\underline{C_{max}}$ be the maximum number of call nodes in a procedure/method. Let $\underline{N_{map}}$ be the maximum number of actual-to-unknown-initial-value bindings stored at a call node. N_{map} is at most $necfi * nr * np$, where $necfi = N_{ecfi}$, $nr = N_{rc}$ and $np = (N_h + N_{uiv})N_{uiv}$. Here $necfi$ is the maximum number of *ECFInfo*'s, nr is the maximum number of relevant contexts associated with each actual-to-unknown-initial-value binding and np is the maximum number of pairs of actuals and unknown initial values. Let $\underline{N_{prop}}$ be the maximum amount of work done by Points-to-Algo at a call node for each concrete value of an unknown initial value computed at the entry node of the method containing the call node. N_{prop} is at most $O(N_{map} * (ce + cp))$, where

$\{ce = (N_h + N_{uiv})^{2k}k * (N_h + N_{uiv}),\ cp = N_{rc}(N_h + N_{uiv})\}$. Here ce is the worst case cost of evaluating a relevant context and the associated actual. $k = k1 + k2 + k3$. This is because each unknown initial value can have at most $N_h + N_{uiv}$ concrete values (recall an interface initial value can be a concrete value). 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 $O((N_{rc}(N_h + N_{uiv})N_{uiv})C_{max}N_{prop})$. Here $O(N_{rc}(N_h + N_{uiv})N_{uiv})$ is an upper bound on the sum of the numbers of pairs of cv-to-uiv bindings computed at the entry nodes of methods.

E.6.4 Complexity of Phase III-pta

The worst-case cost of evaluating a *PTA-dfelm* during this phase is
$O(er * ep)$, where $\{er = (N_h + N_{uiv})^{2k}k,\ ep = (N_h + N_{uiv})^2\}$. Here $k = k1 + k2 + k3$, er is the worst case cost of evaluating the relevant context of the *PTA-dfelm* and ep is the worst-case cost of evaluating the *points-to* of the *PTA-dfelm*. Hence the worst case cost of this phase is $O(er * ep * N_{nodes} * N_{pdfe})$.

Appendix F

Worklist initialization for Phase I-dua of Def-Use-Algo

This appendix presents the pseudocode for worklist initialization done during Phase I-dua. Recall that the function *initialize-worklist-dua* performs worklist initialization during Phase I-dua.

F.1 initialize-worklist-I-dua (Figure F.2)

Purpose of *initialize-worklist-I-dua***:** *initialize-worklist-I-dua (Figure F.2)* is very similar to *initialize-worklist-I-pta* (described in Appendix B). It initializes the worklist with the definitions generated at entry nodes, and reachable (marked by *mark-reachable*) assignment nodes, throw nodes, object creation sites and call nodes.

Call graph of *initialize-worklist-I-dua***:** The call graph of *initialize-worklist-I-dua* is shown in Figure F.1. Table F.1 shows the section numbers and figure numbers containing the explanation/pseudocode of the functions invoked by *initialize-worklist-I-dua*. If an entry is *terminal*, it means that for simplicity we will not be presenting the pseudocode of the corresponding function. We will only describe the properties of such a function.

Explanation of the pseudocode using an example: The pseudocode of the functions invoked by *initialize-worklist-I-dua* is given in Section F.2; here we will explain these functions informally.

propagate-unknown-init-defs-from-entry-node (Figure F.2) initializes the worklist with *DUA-dfelms*-**a** representing defs of parameters, and unknown initial defs of globals and fields of unknown initial values. It considers only those parameters and fields of unknown initial values that have been found to be read. It considers only those

```
                    initialize-worklist-I-pta
                             |
   ------------------------------------------------------------
   |     |     |              |                               |
 func1 func2 func3          func4                           func5
   |     |     |            / | \\_____________   __________/ |
   |     |     |           /  |  \             \ /            |
   |     |     | back-bind-dua |   \           func6           |
   |     |     |              | propagate-to-call-site-dua    |
   |     |     |              |                               |
   |     |     |              |          ____________________/
   |     |     |              |          |
    \    |     |              |          |
     \   |     |              |          |
      add-to-soln-and-worklist-if-needed-I-dua
```

```
Here func1 is propagate-unknown-init-defs-from-entry-node,
     func2 is propagate-exception-objects-from-throw-node,
     func3 is propagate-defs-of-fields-of-newly-created-object,
     func4 is back-bind-using-def-use-summary-transfer-function,
     func5 is propagate-defs-from-assignment-node
and  func6 is generate-defs-due-to-pot-aliases.
```

Figure F.1: Overview of the call graph of initialize-worklist-I-dua

Function Name	Sections Containing Explanation/Pseudocode	Sections Containing Pseudocode
func1	Sec F.1	F.2.1
func2	Sec F.1	F.2.2
func3	Sec F.1	F.2.3
func4	Sec F.1	F.2.4
func5	Sec F.1	F.2.6
func6	Sec F.2.5	F.2.5
back-bind-dua	Sec F.2.4	terminal
add-to-soln-and-worklist-if-needed-I-dua	Sec 4.7.4	Fig 4.19
propagate-to-call-site-dua	Sec G.9 in Appendix G	Fig G.11

Table F.1: Guide to functions invoked by initialize-worklist-I-dua

initialize-worklist-I-dua( ) {
worklist = *empty*

for each method entry node n in the current SCC
1: propagate-unknown-init-defs-from-entry-node(n)

for entry node n of each reachable *try* t in the current SCC
if (*innermost-enclosing-exception-block(t)* is a *finally*)
propagate-unknown-init-defs-from-entry-node(n)

for each reachable throw node n in the current SCC
2: propagate-exception-objects-from-throw-node(n)

for each reachable object creation site n in the current SCC {
3: propagate-defs-of-fields-of-newly-created-object(n)
}

for each reachable call site n in the current SCC
if (the result of the call is stored in a variable)
Let *n.lhs* be the variable p
/*** generate DUA-*dfelms*-**a** representing def of p at n ***/
for each [*ecfi*,*rc*,⟨*p*,*val*⟩] ∈ *n.successor.reaching-PTA-dfelms*
y = [*ecfi*,*Empty-context*,⟨*p*,*n*,⟨*rc*,*val*⟩⟩]
add-to-soln-and-worklist-if-needed-I-dua(*n.successor.successor,* {y})
for each method m invocable from n
if (m is not in the current SCC)
4:back-bind-using-def-use-summary-transfer-function(*m*,*n*)

for each reachable assignment node n in the current SCC
5: propagate-defs-from-assignment-node(n)
}

Figure F.2: worklist initialization for Phase I-dua globals that are in the *used-set* of the method.

propagate-exception-objects-from-throw-node (Figure F.2) initiates the propagation of *DUA-dfelms*-**b** representing definitions of exception objects. It uses the points-to solution to determine the exception objects thrown by n. For example, at statement **15** in Figure 1.5, there is only one possible exception object, $e1_{init}$; hence, [**excp-def,***Empty-context*,$e1_{init}$,*15*] is propagated to the exit node of *innermost-enclosing-exception-block(15)*, i.e., program point **16**, the exit node of the try statement.

propagate-defs-of-fields-of-newly-created-object(Figure F.2) initializes with *null* the fields of the heap-name representing the objects allocated at an object creation site, and adds *DUA-dfelms*-**a** representing these definitions to the *worklist*. It also adds to the *worklist* *DUA-dfelms*-**a** representing assignments to the left-hand-side variable of n (in which a reference to the newly created object is stored). For the definitions of the fields of the heap-name and the left-hand-side variable of n, at n, it adds to the worklist *DUA-dfelms*-**a** representing these definitions in different exception contexts and reachability contexts.

back-bind-using-def-use-summary-transfer-function (Figure F.2) instantiates the *DUA-dfelms* at the exit node of a non-same-SCC method m using actual-to-uiv bindings at a call site of m in the current SCC, and propagates the instantiations to the successor of the call site. It uses the Phase I-pta points-to solution to determine the methods invocable from a dynamically dispatched call site. Actual-to-uiv bindings are computed by *phase-I-pta*. For example, [*empty*,*Empty-context*,⟨$a1_{init}$*.next,4*,⟨*Empty-context*,$a3_{init}$⟩⟩] reaches the exit of *method1* and hence this *DUA-dfelm*-**a** is part of the def-use summary transfer function of *method1*. Thus, when Phase I-dua is done on *method2*, at call site **9**, the above *DUA-dfelm*-**a** is instantiated to [*empty*,*Empty-context*,⟨$a4_{init}$*.next,4*,⟨*Empty-context*,$a6_{init}$⟩⟩], which is added to the worklist.

propagate-defs-from-assignment-node (Figure F.2) initializes the worklist with *DUA-dfelms*-**a** representing defs generated at reachable assignment nodes. It uses the points-to solution computed by *Phase-I-dua* to determine the defs generated by an assignment node. For each def generated by an assignment node, it also computes the potential defs due to potential aliases at the entry of a method. For example, at statement **4** in Figure 1.5, using the points-to solution it generates the def [*empty*,*Empty-context*,$\langle a1_{init}.next,4,\langle Empty\text{-}context,a3_{init}\rangle\rangle$], and this def implies the potential def [*empty*,$\langle (a1_{init}$ **eq** $a2_{init}),empty,empty\rangle,\langle a2_{init}.next,4,\langle Empty\text{-}context,a3_{init}\rangle\rangle$], both of which are added to the worklist.

F.2 Auxiliary functions of initialize-worklist-I-dua

This section presents pseudocode for the functions invoked by initialize-worklist-I-dua. We skip the detailed description of the pseudocode as these functions are very similar to the corresponding functions described in Appendix B.

F.2.1 propagate-unknown-init-defs-from-entry-node

```
1. propagate-unknown-init-defs-from-entry-node( n ) {
/*** it propagates DUA-dfelms-a ***/
/*** n is the entry node of a method ***/

  for each field u.f where u is an unknown initial value at n and u.f
      has been marked as read
    y=[empty,Empty-context,⟨u.f,u.f_initdef,⟨Empty-context,u.f_init⟩⟩]
    add-to-soln-and-worklist-if-needed-I-dua(n.successor, {y})

  /*** n is the entry node of n.method ***/
  for each global g ∈ used-set of n.method
    y = [empty,Empty-context,⟨g,g_initdef,⟨Empty-context,g_init⟩⟩]
    add-to-soln-and-worklist-if-needed-I-dua(n.successor, {y})

  for each parameter p of n.method read in n.method
    y = [empty,Empty-context,⟨p,n,⟨Empty-context,p_init⟩⟩]
    add-to-soln-and-worklist-if-needed-I-dua(n.successor, {y})
}
```

Data structures used by *propagate-unknown-init-defs-from-entry-node***:** n is the entry node of a method. *used-set* of a method is defined in Section B.2.

F.2.2 propagate-exception-objects-from-throw-node

2. propagate-exception-objects-from-throw-node(n) {
```
/*** n is represents a throw statement ***/
  gen-DUA-dfelms-b = ϕ
  /*** eo is an exception object thrown by n ***/
  for each ⟨ecfi1,rc3,eo ⟩ ∈ n.rc-excp-object-pairs
    gen-DUA-dfelms-b = gen-DUA-dfelms-b ∪ {[excp-def,rc3,eo,n]}
    successor = exit node of innermost-enclosing-exception-block(n)
  add-to-soln-and-worklist-if-needed-I-dua(successor, gen-DUA-dfelms-b)
}
```

Data structures used by *propagate-exception-objects-from-throw-node***:** n is an ICFG node that represents a throw statement. Recall that in *RL* a throw statement has the form *throw p*. Each element of *n.rc-excp-object-pairs* stores an exception object thrown by n and the relevant context and *ECFInfo* under which this exception object is thrown by n. Recall that *n.rc-excp-object-pairs* is computed during Phase I-pta, as described in Section C.2.

F.2.3 propagate-defs-created-at-an-object-creation-site

3. propagate-defs-created-at-an-object-creation-site(n) {

```
/*** n represents an object creation site or a new statement ***/
  gen-DUA-dfelms-a = ϕ

  for each ⟨tag,reachable⟩ computed by mark-reachable at n,
      such that tag is empty or a label

    /*** propagate def of left-hand-side variable ***/
    y = [tag,Empty-context,⟨n.lhs,n,⟨Empty-context,object_n⟩⟩]
    gen-DUA-dfelms-a = gen-DUA-dfelms-a ∪ {y}

    /*** propagate defs of the fields of the newly created object ***/
    for each field f of object_n
      y = [tag,Empty-context,⟨object_n.f,n,⟨Empty-context,null⟩⟩]
      gen-DUA-dfelms-a = gen-DUA-dfelms-a ∪ {y}


  /*** propagate the defs in exception contexts ***/
  for each [rc,excp-obj] ∈ n.reaching-PTA-dfelms

    /*** propagate def of left-hand-side variable ***/
    y = [get-ecfi(excp-obj),rc,⟨n.lhs,n,⟨Empty-context,object_n⟩⟩]
    gen-DUA-dfelms-a = gen-DUA-dfelms-a ∪ {y}

    /*** propagate defs of the fields of the newly created object ***/
    for each field f of object_n
      y = [get-ecfi(excp-obj),rc,⟨object_n.f,n,⟨Empty-context,null⟩⟩]
      gen-DUA-dfelms-a = gen-DUA-dfelms-a ∪ {y}


  add-to-soln-and-worklist-if-needed-I-dua(n.successor, gen-DUA-dfelms-a)
}
```

Data structures used by *propagate-defs-created-at-an-object-creation-site***:** n represents an object creation site or a *new* statement. Recall that in *RL* each new statement is of the form p = *new A(...);*. *n.lhs* represents the variable in which a reference to the newly created object is stored.

F.2.4 back-bind-using-def-use-summary-transfer-function

```
4. back-bind-using-def-use-summary-transfer-function(m, c) {
  for each dfe ∈ m.exit-node.reaching-DUA-dfelms

    /*** dfe is DUA-dfelm-a [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩] or
    dfe is DUA-dfelm-b [excp-def,rc1,excp-obj,throw-point] ****/

    if ((dfe is a DUA-dfelm-b) or (var is not a local variable and
        def is a program point))

      x = back-bind-dua(dfe, c, m.exit-node)
      /*** back-bind returns [ecfi1,rc3,⟨var1,def,⟨rc4,val1⟩⟩]'s
      or [excp-def,rc3,excp-obj1,throw-point]'s to which dfe maps at c ***/

      z = x

      for each DUA-dfelm-a dfe1 ∈ x
        z = z ∪ generate-defs-due-to-pot-aliases(c.method,dfe1)

      propagate-to-call-site-dua(c, z, dfe)
}
```

Data structures used by *back-bind-using-def-use-summary-transfer-function*: *c* is an ICFG node representing a statically or a dynamically dispatched call site and *m* is a method invoked from *c* such that *m* is not contained in the SCC containing *c*. *dfe* is *DUA-dfelm*-**a** [*ecfi*,*rc1*,⟨*var,def,*⟨*rc2,val*⟩⟩] or *dfe* is *DUA-dfelm*-**b** [**excp-def,***rc1*,*excp-obj*,*throw-point*]. *x* is a set of *DUA-dfelms*. *m.exit-node* is the exit node of *m*. *generate-defs-due-to-pot-aliases* is similar to *generate-points-tos-due-to-pot-aliases* (Section C.1.4) and it is defined later in Section F.2.5.

back-bind-dua(dfe, c, n) uses the bindings in *c.actual-to-uiv-bindings* to instantiate the unknown initial values in *dfe* with their values at *c*. *back-bind* returns the set of *DUA-dfelms* resulting from the instantiations.

propagate-to-call-site-dua (Figure G.11) is very similar to *propagate-to-call-site* (Section C.9.2), except that *propagate-to-call-site-dua* propagates *DUA-dfelms* instead of *PTA-dfelms*.

F.2.5 generate-defs-due-to-pot-aliases

```
generate-defs-due-to-pot-aliases(method, dfe) {
  generated-dfes = φ
  let dfe be [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩]
  let rc1 be ⟨ac1, tc1, etc1⟩
  if ( var is u.f and u is an unknown initial value ) {
    for each unknown initial value v at method.entry compatible with u
      if ( v.f has been marked as read ) {
        rc3 = ⟨ac1 ∧ (u eq v), tc1, etc1⟩
        new-dfe = [ecfi,rc3,⟨v.f,def,⟨rc2,val⟩⟩]
        generated-dfes = generated-dfes ∪ {new-dfe}
      }
  }
  return generated-dfes }
```

Purpose of *generate-defs-due-to-pot-aliases***:** *generate-defs-due-to-pot-aliases* is very similar to *generate-points-tos-due-to-pot-aliases* (Section C.1.4), except *generate-defs-due-to-pot-aliases* generates *DUA-dfelms*-**a** instead of *PTA-dfelms*-**a**.

Data structures used by *generate-defs-due-to-pot-aliases***:**

dfe is a *DUA-dfelm*-**a** [*ecfi*,*rc1*,⟨*var*,*def*,⟨*rc2*,*val*⟩⟩]. *method.entry* is the entry node of *method*. Recall that two classes are *compatible* if and only if they are the same class or they are related by subtype-supertype relationship, and two unknown initial values are *compatible* if their declared classes are *compatible*. *generate-defs-due-to-pot-aliases* returns a set of *DUA-dfelms*-**a**.

Explanation of the pseudocode using an example: *generate-defs-due-to-pot-aliases* generates *DUA-dfelms*-**a** representing definitions of fields of unknown initial values due to potential aliasing between *compatible* unknown initial values at the entry node of a method. For example, in Figure 1.5, *generate-defs-due-to-pot-aliases* generates the *DUA-dfelms*-**a** [*empty*, ⟨($a1_{init}$ *eq* $a2_{init}$),*empty*,*empty*⟩, ⟨$a2_{init}$.*next*,*4*,⟨*Empty-context*, $a3_{init}$⟩⟩] when *method* is *method2* and *dfe* is [*empty*, *Empty-context*, ⟨$a1_{init}$.*next*,*4*,⟨*Empty-context*, $a3_{init}$⟩⟩].

F.2.6 propagate-defs-from-assignment-node

```
5. propagate-defs-from-assignment-node( n ) {
  gen-DUA-dfelms-a = φ

  if ( n.lhs does not have dereference )

    if ( n.lhs is a pointer )
      if ( n.rhs is 0 )
        for each ⟨tag,reachable⟩ computed by mark-reachable at n,
          such that tag is empty or a label
          y = [tag,Empty-context,⟨n.lhs,n,⟨Empty-context,null⟩⟩]
          gen-DUA-dfelms-a = gen-DUA-dfelms-a ∪ {y}
        for each [rc,excp-obj] ∈ n.reaching-PTA-dfelms
          y = [get-ecfi(excp-obj),rc,⟨n.lhs,n,⟨Empty-context,null⟩⟩]
          gen-DUA-dfelms-a = gen-DUA-dfelms-a ∪ {y}
      else
        for each ⟨ecfi,rc,val ⟩ ∈ n.rhs-rc-loc-pairs
          y = [ecfi,Empty-context,⟨n.lhs,n,⟨rc,val⟩⟩]
          gen-DUA-dfelms-a = gen-DUA-dfelms-a ∪ {y}

    else /*** n.lhs is not a pointer ***/
      for each ⟨tag,reachable⟩ computed by mark-reachable at n,
        such that tag is empty or a label
        y = [tag,Empty-context,⟨n.lhs,n,⟨Empty-context,don't-care⟩⟩]
        gen-DUA-dfelms-a = gen-DUA-dfelms-a ∪ {y}
      for each [rc,excp-obj] ∈ n.reaching-PTA-dfelms
        y = [get-ecfi(excp-obj),rc,⟨n.lhs,n,⟨Empty-context,don't-care⟩⟩]
        gen-DUA-dfelms-a = gen-DUA-dfelms-a ∪ {y}

    add-to-soln-and-worklist-if-needed-I-dua(n.successor, gen-DUA-dfelms-a)
    return


  /*** here we know that n.lhs has indirection ***/
  if ( n.lhs is not of pointer type )
    /*** n represents assignment to a non-pointer variable ***/
    Let n.lhs be p→f

    /*** compute n.lhs-rc-loc-pairs because Phase I-pta has not computed
      n.lhs-rc-loc-pairs as n is pass-through for points-to analysis ***/
    n.lhs-rc-loc-pairs = φ
    for each [ecfi1,rc1,⟨p,obj⟩] ∈ n.reaching-PTA-dfelms
      n.lhs-rc-loc-pairs = n.lhs-rc-loc-pairs ∪ ⟨ecfi1,rc1,obj.f ⟩

    for each ⟨ecfi,rc,var ⟩ ∈ n.lhs-rc-loc-pairs
      y = [ecfi,rc,⟨var,n,⟨Empty-context,don't-care⟩⟩]
      gen-DUA-dfelms-a = gen-DUA-dfelms-a ∪ {y}
      gen-DUA-dfelms-a = gen-DUA-dfelms-a ∪
        generate-defs-due-to-pot-aliases(n.method, y)

    add-to-soln-and-worklist-if-needed-I-dua(n.successor, gen-DUA-dfelms-a)
    return
```

```
  /*** here we know that n represents a pointer assignment ***/
  if ( n.rhs is 0 )
    for each ⟨ecfi,rc,var ⟩ ∈ n.lhs-rc-loc-pairs
      y = [ecfi,rc,⟨var,n,⟨Empty-context,null⟩⟩]
      gen-DUA-dfelms-a = gen-DUA-dfelms-a ∪ {y}
      gen-DUA-dfelms-a = gen-DUA-dfelms-a ∪
        generate-defs-due-to-pot-aliases(n.method, y)

    add-to-soln-and-worklist-if-needed-I-dua(n.successor, gen-DUA-dfelms-a)
    return

  for each ⟨ecfi,rc1,var ⟩ ∈ n.lhs-rc-loc-pairs
    for each ⟨ecfi,rc2,val ⟩ ∈ n.rhs-rc-loc-pairs
      y = [ecfi,rc1,⟨var,n,⟨rc2,val⟩⟩]
      gen-DUA-dfelms-a = gen-DUA-dfelms-a ∪ {y}
      gen-DUA-dfelms-a = gen-DUA-dfelms-a ∪
        generate-defs-due-to-pot-aliases(n.method, y)

  add-to-soln-and-worklist-if-needed-I-dua(n.successor, gen-DUA-dfelms-a)
}
```

Data structures used by *propagate-defs-from-assignment-node***:** *n* is an ICFG node that represents an assignment statement. Recall that *n.lhs* represents the left-hand-side expression of the assignment statement and *n.rhs* represents the right-hand-side expression of the assignment statement. *n.lhs-rc-loc-pairs* and *n.rhs-rc-loc-pairs* are as defined in Section C.1. Recall that if *n.rhs* is of an arithmetic type, i.e., it does not represent any address, then *n.rhs-rc-loc-pairs* is **{**⟨*empty, Empty-context, don't-care*⟩**}**.

Appendix G

Phase I-dua transfer functions

This appendix presents pseudocode for the Phase I-dua node transfer functions. We will skip detailed description of the pseudocode as most of the pseudocode in this appendix is very similar to the pseudocode in Appendix C. Familiarity with the pseudocode in Appendix C is a prerequisite for understanding the pseudocode in this appendix.

n.def-uses represents the set of def-use associations computed at an ICFG node n and *n.reaching-DUA-dfelms* represents the Phase I-dua solution computed (so far) at an ICFG node n.

G.1 Assignment, condition and object creation nodes (Figure G.1)

Figure G.1 defines the def-use-analysis transfer functions of assignment, condition and new (object creation site) nodes. *generate-DUA-dfelms-***a***-due-to-pot-non-aliases(n, rdfe)* (whose pseudocode is given in Section G.11.1) is very similar to *generate-points-tos-due-to-pot-non-aliases* defined earlier in Section C.1.5. *generate-DUA-dfelms-***a***-due-to-pot-non-aliases(n, rdfe)* conditions the propagation of *rdfe* across n on potential non-aliases. For example, when *rdfe* is the *DUA-dfelm-***a** $[\mathit{empty}, \mathit{Empty\text{-}context}, \langle a2_{init}.next, a2_{init}.next_{initdef}, \langle \mathit{Empty\text{-}context}, a2_{init}.next_{init}\rangle\rangle]$ at statement **4** in Figure 1.5, *generate-DUA-dfelms-***a***-due-to-pot-non-aliases* generates the set of *DUA-dfelms-***a**

$\{[\mathit{empty}, \langle (a1_{init}\ neq\ a2_{init}), \mathit{empty}, \mathit{empty}\rangle, \langle a2_{init}.next, a2_{init}.next_{initdef},$
$\langle \mathit{Empty\text{-}context}, a2_{init}.next_{init}\rangle\rangle]\}$.

```
1. process-assignment-dua( n, rdfe ) {
  if (rdfe is a DUA-dfelm-a)
    if ( var is read in n )
      n.def-uses = n.def-uses ∪ [rc1,⟨var,def,n⟩]

    if ( n must update var )
      /*** rdfe is killed by n ***/
      return
    else
      x = generate-DUA-dfelms-a-due-to-pot-non-aliases(n, rdfe)
      add-to-soln-and-worklist-if-needed-I-dua(n.successor, x)
  else // rdfe is a DUA-dfelm-b
    add-to-soln-and-worklist-if-needed-I-dua(n.successor, {rdfe})
}
```

```
2. process-condition-dua( n, rdfe ) {
  if ( rdfe is a DUA-dfelm-a and var is read in n ) {
    /*** p uses ***/
    n.def-uses = n.def-uses ∪ [rc1,⟨var,def,(n,true-successor)⟩]
    n.def-uses = n.def-uses ∪ [rc1,⟨var,def,(n,false-successor)⟩]
  }

  for each successor succ of n
    add-to-soln-and-worklist-if-needed-I-dua(succ,{rdfe})
}
```

```
3. process-new-dua( n, rdfe ) {
  if ( rdfe is a DUA-dfelm-a and var is read in n )
    n.def-uses = n.def-uses ∪ [rc1,⟨var,def,n⟩]
  if ( rdfe is a DUA-dfelm-a and var is n.lhs ) {
    /*** rdfe is killed, need not propagate further ***/
    return
  }
  else
    add-to-soln-and-worklist-if-needed-I-dua(n.successor,{rdfe})
}
```

Figure G.1: Phase I-dua transfer functions for assignment, condition and new nodes

4. process-throw-dua(n, rdfe) {
if (*rdfe* is a *DUA-dfelm*-**b**)
  /*** the exception represented by *rdfe* is overriden by the throw ***/
  return

new-*DUA-dfelms*-**a** = ϕ
if (*n* represents *throw var*)
  n.def-uses = *n.def-uses* $\cup$ [*rc1*,⟨*var,def,n*⟩]
for each ⟨*ecfi,rc3*,**eo** ⟩ $\in$ *n.rc-excp-object-pairs*
    rc4 = *rc1* $\wedge$ *rc3*
    /* store exception signature ***/**
    new-*DUA-dfelms*-**a** = new-*DUA-dfelms*-**a** $\cup$ {[*get-ecfi(***eo***)*,*rc4*,⟨*var,def*,⟨*rc2,val*⟩⟩]}

succ = exit node of *innermost-enclosing-exception-block(n)*

add-to-soln-and-worklist-if-needed-I-dua(succ,new-*DUA-dfelms*-**a**)
}

Figure G.2: Phase I-dua transfer function for throw node

G.2 process-throw-dua (Figure G.2)

process-throw-dua uses the points-to solution computed by Phase I-pta at the *throw* statement to determine (1) the exception objects thrown and (2) the *ECFInfo*'s and relevant contexts in which these objects are thrown. It stores the signature of the exception in the *ECFInfo* of the generated *DUA-dfelm*-**a**. The *ECFInfo* determines the future propagation of the generated *DUA-dfelm*-**a** from the exit node of *innermost-enclosing-exception-block(n)*. For example, at statement **15** in Figure 1.5, there is only one *eo*, $e1_{init}$, and the corresponding *ECFInfo* is *empty* and the corresponding *rc3* is *Empty-context*. As a result, when the *rdfe* [*empty*,*Empty-context*,⟨*global3,14*,⟨*Empty-context*,$object_{14}$⟩⟩] reaches statement **15** (from statement **14**), the *DUA-dfelm*-**a** [$e1_{init}$,*Empty-context*,⟨*global3,14*,⟨*Empty-context*,$object_{14}$⟩⟩] is generated.

5. process-try-entry-dua(n, rdfe) {
```
  if ( innermost-enclosing-exception-block(n.try)  is a finally )
    // try nested inside a finally
    // treat like a call to an anonymous procedure
    process-call-dua( n.call-node, rdfe )
  else
    add-to-soln-and-worklist-if-needed-I-dua(n.successor,{rdfe})
}
```

Figure G.3: Phase I-dua transfer function for entry node of try

G.3 process-try-entry-dua (Figure G.3)

As stated earlier, a try statement directly contained in a finally statement is treated like a call to an anonymous procedure because it can cause *ECFInfo*'s and exceptions to stack up. If n is the entry node of a try statement directly contained in a finally statement, *n.call-node* is the call node that represents the call to the anonymous procedure.

G.4 process-try-exit-dua (Figure G.4)

Data structures used by *process-try-exit-dua***:** n is an ICFG node representing the exit node of a try statement, *n.try*. If *n.try* is directly contained in a finally statement, *n.proc-exit* represents the exit node of the anonymous procedure that corresponds to the try statement. If *n.try* has a finally statement associated with it, *n.finally* represents this finally statement and *n.finally.entry* represents the entry node of this finally statement. For each catch statement c, *c.entry* is the ICFG node representing the entry node of c.

Explanation of the pseudocode using an example: There are two cases: either (1) *rdfe* represents a variable definition or (2) *rdfe* represents an exception object. If *rdfe* represents a variable definition and it flows to n along a path without any uncaught exception (i.e., *ecfi* is *empty*), *process-try-exit-dua* propagates *rdfe* to the entry node of the finally statement associated with the try statement (if any) or to the successor of n (i.e., the statement following the *try-catch* construct). Otherwise, it propagates *rdfe* to the entry nodes of the catch statements that are associated with the try statement and that can catch the exception represented by *ecfi* or *excp-obj*. If the exception can escape all of the catch statements associated with the try statement, *process-try-exit-dua* calls *propagate-to-finally-if-needed-dua* (Figure G.5) to propagate a new *DUA-dfelm*. *get-escape-tcs* returns a conjunction of type constraints which say under what conditions the exception can escape all of the catch statements associated with the try statement. For example, (*first case*) let *rdfe* be the *DUA-dfelm-***a** [$e1_{init}$,*Empty-context*,⟨*global3,14,*⟨*Empty-context,object*$_{14}$⟩⟩] at program point **16** in Figure 1.5, the exit node of the try statement in *method3*. Since *typeof(*$e1_{init}$*)* is *compatible* with *ET2*, *catch-entry-nodes* is **{17}** and *rdfe* is propagated to the entry node of the catch at program point **17**. Moreover, the exception ($e1_{init}$) can escape the catch statement at statement **17** because the concrete type of the exception can be *ET1* or *ET4*. As a result, *get-escape-tcs* returns *(type(*$e1_{init}$*)* $\not\leq$ *ET2)* and *propagate-to-finally-if-needed-dua* propagates the *DUA-dfelm-***a** [$e1_{init}$,⟨*empty,empty,(type(*e_{init}*)* $\not\leq$ *ET2)*⟩,⟨*global3,14,*⟨*Empty-context,object*$_{14}$⟩⟩] to program point 19, i.e., exit node

6. process-try-exit-dua(n, rdfe) {

```
if (rdfe is a DUA-dfelm-a and ecfi is empty or a label)
  /*** termination without exception ***/
  propagate-to-finally-if-needed-dua( n, rdfe )
else {
  /*** termination due to exception ***/
  /*** either first case with ecfi representing an exception or second case ***/

  /*** find appropriate catches ****/
  catch-entry-nodes = φ
  for each catch c associated with n that can catch
      the exception represented by ecfi or excp-obj
    catch-entry-nodes = catch-entry-nodes ∪ { c.entry }
  for each c-entry ∈ catch-entry-nodes
    add-to-soln-and-worklist-if-needed-I-dua(c-entry,{rdfe})

  /*** Can the exception ESCAPE all the catches?  ***/
  if (the exception represented by ecfi or excp-obj
      can escape all the catches associated with n)
    a:  propagate-escaped-exception-dua(n, rdfe)
}
}
```

6.a. propagate-escaped-exception-dua(n, rdfe) }

```
  let rc1 be ⟨ac1, tc1, etc1⟩
  if (first case:  rdfe is a DUA-dfelm-a)
    new-rc = ⟨ac1, tc1, etc1 ∧ get-escape-tcs(n, ecfi)⟩
    new-DUA-dfelm = [ecfi,new-rc,⟨var,def,⟨rc2,val⟩⟩]
  else
    /*** second case:  rdfe is a DUA-dfelm-b ***/
    new-rc = ⟨ac1, tc1, etc1 ∧ get-escape-tcs(n, excp-obj)⟩
    new-DUA-dfelm = [excp-def,new-rc,excp-obj,throw-point]

  propagate-to-finally-if-needed-dua( n, new-DUA-dfelm  )
}
```

Figure G.4: Phase I-dua transfer function for exit node of try

```
propagate-to-finally-if-needed-dua( n, dfe ) {
if ( dfe is a DUA-dfelm-a and dfe.ecfi is empty )
  let dfe be [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩]
  succ = successor of the try-catch-finally construct associated with n
  dfe = [succ,rc1,⟨var,def,⟨rc2,val⟩⟩]

if ( n has a finally associated with it )
  add-to-soln-and-worklist-if-needed-I-dua( n.finally.entry, {dfe} )
else
  if ( innermost-enclosing-exception-block(n.try) is a finally )
    /*** try nested inside a finally ***/
    /*** treat like a return from an anonymous procedure ***/
    process-method-exit-dua( n.proc-exit, dfe )
    return
  if ( (dfe is a DUA-dfelm-a and dfe.ecfi represents an exception) or
    dfe is a DUA-dfelm-b )
    /*** termination due to exception ***/
    x = exit node of innermost-enclosing-exception-block(n.try)
    add-to-soln-and-worklist-if-needed-I-dua( x, {dfe} )
    return

  /*** dfe is a DUA-dfelm-a and dfe.ecfi is a label ***/
  if ( dfe.ecfi is contained in innermost-enclosing-exception-block(n.try) )
    new-dfe = [empty,rc1,⟨var,def,⟨rc2,val⟩⟩]
    add-to-soln-and-worklist-if-needed-I-dua( dfe.ecfi, {new-dfe} )
  else
    x = exit node of innermost-enclosing-exception-block(n.try)
    add-to-soln-and-worklist-if-needed-I-dua( x, {dfe} )
}
```

Figure G.5: auxiliary function of process-try-exit-dua of *innermost-enclosing-exception-block(16)*. Next, (*second case*) let *rdfe* be the *DUA-dfelm*-**b** [**excp-def,***Empty-context*,e1_{init},*15*] at program point **16**. Due to the same reasons as in the *first case*, *rdfe* is propagated to program point **17** and the *DUA-dfelm*-**b** [**excp-def,**$\langle$*empty,empty,(type(*e_{init}*)* $\not\leq$ *ET2)*$\rangle$,e1_{init},*15*] is propagated to program point 19.

G.5 process-catch-entry-dua (Figure G.6)

Data structures used by ***process-catch-entry-pta*****:** *n* is an ICFG node representing the entry node of a catch statement that catches any exception whose type is *catch-type* or a subtype of *catch-type*. *rc1*, the relevant context of *rdfe*, is ⟨ac1, tc1, etc1⟩. If *rdfe* is a *PTA-dfelm*-**a**, *rdfe.ecfi* is the *ECFInfo* of the *PTA-dfelm*-**a**. If *rdfe* is a *PTA-dfelm*-**b**, *rdfe.excp-obj* is the exception object represented by the *PTA-dfelm*-**b**.

Explanation of the pseudocode using an example:

If *rdfe* represents a variable definition, *process-catch-entry-dua* catches the exception represented by the *ECFInfo* of *rdfe*. If *rdfe* represents an exception object, *process-catch-entry-dua* generates a def-use association between the throw statement that threw the exception object and the entry node of the catch statement. In the second case, *process-catch-entry-dua* also generates a *DUA-dfelm*-**a** representing a definition of the parameter of the catch statement because the parameter is assigned the caught exception object. If *ECFInfo* or *excp-object* is an unknown initial value, *get-catch-tcs* is used to generate the appropriate type constraints under which the exception is caught by the catch statement. For example, consider program point **17** in Figure 1.5. Consider two different *rdfe*'s at this program point: *DUA-dfelm*-**a** [$e1_{init}$,*Empty-context*,⟨*global3,14*,⟨*Empty-context,object*$_{14}$⟩⟩] and *DUA-dfelm*-**b** [**excp-def,***Empty-context*,$e1_{init}$,*15*]. In the first case, *ECFInfo* is $e1_{init}$ and hence *get-catch-tcs* returns *(type(*$e1_{init}$*)* $\leq$ *ET2)*. As a result, *new-dfe* is the *DUA-dfelm*-**a**
[*empty*,⟨*empty,empty,(type(*$e1_{init}$*)* $\leq$ *ET2)*⟩,⟨*global3,14*,⟨*Empty-context,object*$_{14}$⟩⟩]. In the second case, *excp-object* is $e1_{init}$ and again *get-catch-tcs* returns *(type(*$e1_{init}$*)* $\leq$ *ET2)*. As a result, the *DUA-dfelm*-**d** [⟨*empty,empty,(type(*$e1_{init}$*)* $\leq$ *ET2)*⟩,⟨$e1_{init}$*,15,17*⟩] is added to the solution at program point **17** and
new-dfe = [*empty*,⟨*empty,empty,(type(*$e1_{init}$*)* $\leq$ *ET2)*⟩,⟨excp,17,⟨*Empty-context*,$e1_{init}$⟩⟩]is generated to represent a definition of the parameter of the catch statement.

7. process-catch-entry-dua(n, rdfe) {
/*** n is the entry node of a catch statement that catches any exception whose type is *catch-type* or a subtype of *catch-type* ***/

if (*first case*)
  /*** *rdfe* represents a variable definition ***/
  a: *new-dfe* = **catch-variable-def**(*n, rdfe*)
else
  /*** *second case*: *rdfe* represents an exception object ***/
  b: *new-dfe* = **catch-exception-object**(*n, rdfe*)

add-to-soln-and-worklist-if-needed-I-dua(n.successor, {*new-dfe*})
}

7.a. catch-variable-def(n, rdfe) {
if (*rdfe* is [*exp-type*,*rc1*,⟨*var*,*def*,⟨*rc2*,*val*⟩⟩])
  /*** *process-try-exit-dua* has ensured that *exp-type* is either same
    as the *catch-type* or it is a subtype of the *catch-type* ***/
  new-dfe = [*empty*,*rc1*,⟨*var*,*def*,⟨*rc2*,*val*⟩⟩]
else
  /*** *ecfi* is an unknown initial value *uiv*.
    process-try-exit-dua has ensured that *typeof(uiv)*
    and the *catch-type* are *compatible* ***/
  /*** let *rc1* be ⟨ac1, tc1, etc1⟩***/
  new-rc = ⟨ac1, tc1, etc1 $\wedge$ get-catch-tcs(*n, uiv*)⟩
  new-dfe = [*empty*,*new-rc*,⟨*var*,*def*,⟨*rc2*,*val*⟩⟩]
return *new-dfe* **}**

7.b. catch-exception-object(n, rdfe) {
if (*rdfe.excp-obj* is a heap-name)
  new-rc = *rc1*
else
  /*** *rdfe.excp-obj* is an unknown initial value ***/
  /*** let *rc1* be ⟨ac1, tc1, etc1⟩***/
  new-rc = ⟨ac1, tc1, etc1 $\wedge$ get-catch-tcs(*n, rdfe.excp-obj*)⟩

  n.def-uses = *n.def-uses* $\cup$ [*new-rc*,⟨*rdfe.excp-obj*,*throw-point*,*n*⟩]

  /*** param is the parameter of the catch ***/
  new-dfe = [*empty*,*new-rc*,⟨*param*,*n*,⟨*Empty-context*,*rdfe.excp-obj*⟩⟩]
return *new-dfe* **}**

Figure G.6: Phase I-dua transfer function for catch entry node

```
8. process-catch-exit-dua( n, rdfe ) {
  if ( rdfe is DUA-dfelm-a and var is parameter of the catch )
    /*** no need to propagate further as var is local to the catch ***/
    return
  propagate-to-finally-if-needed-dua( n, rdfe )
}
```

Figure G.7: Phase I-dua transfer function for exit node of a catch

G.6 process-catch-exit-dua (Figure G.7)

process-catch-exit-dua calls *propagate-to-finally-if-need-dua*, which is defined in Figure G.5.

10. process-finally-exit-dua(n, rdfe) {

```
  /*** n.try is the try statement associated with n ***/
  if ( innermost-enclosing-exception-block(n.try)  is a finally )
    /*** try nested inside a finally ***/
    /*** treat like a return from an anonymous procedure ***/
    process-method-exit-dua( n.proc-exit, rdfe )
    return
  if ( (rdfe is a PTA-dfelm-a and ecfi represents an exception) or
    rdfe is a PTA-dfelm-b )
    /*** termination due to exception ***/
    x = exit node of innermost-enclosing-exception-block(n.try)
    add-to-soln-and-worklist-if-needed-I-dua( x, {rdfe} )
    return

  /*** rdfe is a DUA-dfelm-a [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩] and ecfi is a label ***/
  if ( ecfi is contained in innermost-enclosing-exception-block(n.try)  )
    new-dfe = [empty,rc1,⟨var,def,⟨rc2,val⟩⟩]
    add-to-soln-and-worklist-if-needed-I-dua( ecfi, {new-dfe} )
  else
    x = exit node of innermost-enclosing-exception-block(n.try)
    add-to-soln-and-worklist-if-needed-I-dua( x, {rdfe} )
}
```

Figure G.8: Phase I-dua transfer function for exit node of a finally

G.7 process-finally-exit-dua (Figure G.8)

process-finally-exit-dua is very similar to *process-finally-exit-pta* defined in Section C.7, except *process-finally-exit-dua* propagates *DUA-dfelms* instead of *PTA-dfelms*.

11. process-call-dua(n, rdfe) {
  if (*rdfe* is a *DUA-dfelm*-**a**)
    if (*var* is an *actual*)
      n.def-uses = *n.def-uses* ∪ {[*rc1*,⟨*var,def,n*⟩]}

    /*** compute p-uses due to dynamic dispatch ***/
    if (*var* is the receiver variable)
      /*** e.g., in *p*→foo(), *p* is the receiver variable ***/
      a: compute-p-uses-due-to-dynamic-dispatch(*n, rdfe*)

    /*** compute new bindings for unknown initial defs ***/
    b: *new-def-uidef-bindings* = compute-new-bindings-of-defs-and-u-i-defs(n, rdfe)
    n.def-uidef-bindings = *n.def-uidef-bindings* ∪ *new-def-uidef-bindings*

    back-propagate-dua(n, new-def-uidef-bindings)
    if (*var* is *n.left-hand-side*)
      /*** *rdfe* is killed as the result of the call is stored in *var* ***/
      return
    if (*rdfe.var* is a local, a field of a heap-name or a field of an unknown initial value)
      propagate-across-call-*DUA-dfelm*-**a**(n, rdfe)
    else // *rdfe.var* is a global
      if (*rdfe.var* ∉ the *used-set* of at least one method invocable from *n* in the
        initial call graph)
        propagate-across-call-*DUA-dfelm*-**a**(n, rdfe)
  else // rdfe is a *DUA-dfelm*-**b** and it represents an exception object
    propagate-across-call-*DUA-dfelm*-**b**(n, rdfe)
}

Figure G.9: Phase I-dua transfer function for call node

G.8 process-call-dua (Figure G.9)

11.a. compute-p-uses-due-to-dynamic-dispatch uses *rdfe.val* to compute the p-uses due to dynamic dispatch mentioned in Section 4.1. For example, when *rdfe* is [*empty*,*Empty-context*,⟨*r,3*,⟨*Empty-context*,$a1_{init}$⟩⟩] at statement **5** in Figure 1.5, *val* is $a1_{init}$ and the possible targets with $a1_{init}$ as receiver are *A::update* and *C::update*. As a result, the *DUA-dfelms*-**e**
[⟨*empty*,(*type*($a1_{init}$)∈*A::update*),*empty*⟩,⟨*r,3*,⟨*5,A::update*⟩⟩] and
[⟨*empty*,(*type*($a1_{init}$)∈*C::update*),*empty*⟩,⟨*r,3*,⟨*5,C::update*⟩⟩] are added to **5**.*def-uses*.

11.b. compute-new-bindings-of-defs-and-u-i-defs computes the bindings between

back-propagate-dua(n, new-def-to-uidef-bindings) {

```
  for each [ecfi,rc,⟨⟨var,def⟩,uidef⟩] ∈ new-def-to-uidef-bindings
    e = exit node of uidef.method
    for each dfe ∈ e.reaching-DUA-dfelms containing uidef
      if (dfe is a PTA-dfelm-a and dfe.var is not a local variable)
        back-instantiate-dua( dfe, n, e )
}
```

---

propagate-across-call-*DUA-dfelm*-a(n, dfe) {

```
/*** dfe is DUA-dfelm-a [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩] ***/
  for each target t of n
    for each ⟨tag,reachable⟩ computed by mark-reachable at exit of t
      if (tag is empty)
        add-to-soln-and-worklist-if-needed-I-dua( n.successor.successor, { dfe} )
      if (tag is a label contained in innermost-enclosing-exception-block(n) )
        add-to-soln-and-worklist-if-needed-I-dua( tag, { dfe} )

  for each ⟨rc3,excp-obj1⟩ ∈ n.rc-excp-object-pairs
    rc4 = rc1 ∧ rc3
    new-dfe = [get-ecfi(excp-obj1),rc4,⟨var,def,⟨rc2,val⟩⟩]
    add-to-soln-and-worklist-if-needed-I-dua( n.successor, { new-dfe} )
}
```

---

propagate-across-call-*DUA-dfelm*-b(n, dfe) {

```
/*** dfe is DUA-dfelm-b [excp-def,rc1,excp-object,throw-point] ***/
  for each target t of n
    for each ⟨tag,reachable⟩ computed by mark-reachable at the exit of t
      if (tag is empty)
        add-to-soln-and-worklist-if-needed-I-dua( n.successor.successor, { dfe} )
      if (tag is a label contained in innermost-enclosing-exception-block(n) )
        add-to-soln-and-worklist-if-needed-I-dua( tag, { dfe} )
}
```

Figure G.10: Auxiliary functions of process-call-dua the unknown initial defs at the entry nodes of the potential targets of n and the values of these unknown initial defs at n. Each def-uidef binding at n has the form [*ecfi*,*rc*,⟨⟨*var,def*⟩,*uidef*⟩]; here *ecfi* and *rc* respectively are *ECFInfo* and relevant context under which the binding holds, *uidef* is an unknown initial def at the entry of a target of n, *var* is a location whose reaching definitions are represented by *uidef* at the entry of the target, and *def* is a definition point or an unknown initial def that represents a definition of *var*. The reason why *var* is also stored in the binding is to increase precision in those cases where the reaching definition is that of a field of a heap-name or an unknown initial value. This ensures that the different objects and defs reaching the entry node of a method along different paths are not mixed. This is a technical detail whose usefulness is described in Sections 4.5.4 and 4.8. *n.def-uidef-bindings* contains these bindings between the unknown initial defs and their values at n. For example in Figure 1.5, [*empty*,*Empty-context*,⟨⟨$object_{11}$.*next*,*11*⟩,$a2_{init}.next_{initdef}$⟩] $\in$ **12**.*def-uidef-bindings*.

Pseudocode of *compute-p-uses-due-to-dynamic-dispatch* and *compute-new-bindings-of-defs-and-u-i-defs* is given later in Section G.11.2. *back-instantiate-dua* is explained later in Section G.9.

propagate-across-call-DUA-dfelm-**a** and *propagate-across-call-DUA-dfelm*-**b** are respectively similar to *propagate-across-call-PTA-dfelm*-**a** and *propagate-across-call-PTA-dfelm*-**b** defined in Section C.8, except *propagate-across-call-DUA-dfelm*-**a** and *propagate-across-call-DUA-dfelm*-**b** propagate *DUA-dfelms* instead of *PTA-dfelms*.

G.9 process-method-exit-dua (Figure G.11)

Figure G.11 defines Phase I-dua transfer function for method exit nodes. *back-bind-dua(dfe, c, n)* uses the bindings in *c.actual-to-uiv-bindings* and *c.def-uidef-bindings* to instantiate the unknown initial values and the unknown initial defs in *dfe* with their values at *c*. *back-bind-dua* returns the set of *DUA-dfelms* resulting from the instantiations. *back-instantiate-dua* and *propagate-to-call-site-dua* are similar to *back-instantiate* (Figure C.21) and *propagate-to-call-site* (Figure C.21) defined in Section C.9, except that *back-instantiate-dua* and *propagate-to-call-site-dua* propagate *DUA-dfelms* instead of *PTA-dfelms*.

```
12. process-method-exit-dua( n, rdfe ) {
/*** n is the exit node of the method n.method ***/

if (rdfe is a DUA-dfelm-a and var is a local variable of n.method)
  /*** no need to propagate to callers ***/
  return

for each call site c in the current SCC that invokes n.method {
  back-instantiate-dua( rdfe, c, n )
} }
```

```
back-instantiate-dua( rdfe, c, n) {
  x = y = z = φ

  x = back-bind-dua( rdfe, c, n )
  /*** back-bind returns [ecfi1,rc3,⟨var1,def1,⟨rc4,val1⟩⟩]'s
    or [excp-def,rc3,excp-obj1,throw-point]'s to which rdfe maps at c ***/

  if (first case and rdfe.def is a program point )
    /*** i.e., the definition has been generated during
        the life-time of n.method ***/
    y = generate-defs-due-to-pot-aliases(c.method, x)

  z = x ∪ y

  propagate-to-call-site-dua( c, z, rdfe)
}
```

```
propagate-to-call-site-dua( c, instantiated-dfes, dfe) {
  if ( dfe is a DUA-dfelm-a and
        (dfe.ecfi is empty or dfe.ecfi is a label contained in
            innermost-enclosing-exception-block(c) ) )
    if ( dfe.ecfi is empty )
      add-to-soln-and-worklist-if-needed-I-dua(c.successor.successor, instantiated-dfes)
    else
      add-to-soln-and-worklist-if-needed-I-dua(dfe.ecfi, instantiated-dfes)
  else // the call overrides the active ecfi's and exceptions before the call
    add-to-soln-and-worklist-if-needed-I-pta(c.successor, instantiated-dfes)
}
```

Figure G.11: Phase I-dua transfer function of exit node of a method

13. process-break-dua(n, rdfe) {

```
let y = target of break
if (y is contained in innermost-enclosing-exception-block(n) )
  add-to-soln-and-work-list-if-needed-I-dua(y,{rdfe})
else
  if ( first case )
    new-dfe = [y,rc1,⟨var,def,⟨rc2,val⟩⟩]
    x = exit of innermost-enclosing-exception-block(n)
    add-to-soln-and-work-list-if-needed-I-dua(x,{new-dfe})
  // else second case, need not propagate as the exception is overriden by
  // the break statement
}
```

---

14. process-continue-dua(n, rdfe) {

```
let y = target of continue
if (y is contained in innermost-enclosing-exception-block(n) )
  add-to-soln-and-worklist-if-needed-I-dua(y,{rdfe})
else
  if ( first case )
    new-dfe = [y,rc1,⟨var,def,⟨rc2,val⟩⟩]
    x = exit of innermost-enclosing-exception-block(n)
    add-to-soln-and-worklist-if-needed-I-dua(x,{new-dfe})
  // else second case, need not propagate as the exception is overriden by
  // the continue statement
}
```

Figure G.12: Phase I-dua transfer functions for break and continue statements

G.10 Break, continue, return site and return statement

Figures G.12, G.13 and G.14 respectively define def-use-analysis transfer functions for break and continue statements, return sites (i.e., successors of call nodes) and return statements. Recall that *rdfe* is either (1) (*first case*) a *DUA-dfelm*-**a** [*ecfi*,*rc1*,⟨*var*,*def*,⟨*rc2*,*val*⟩⟩] or (2) (*second case*) a *DUA-dfelm*-**b** [**excp-def,***rc1*,*excp-obj*,*throw-point*].

```
15. process-return-site-dua( n, rdfe ) {
if ( first case )
  if ( ecfi is a label )
    if ( ecfi is not contained in innermost-enclosing-exception-block(n) )
      x = exit of innermost-enclosing-exception-block(n)
      add-to-soln-and-worklist-if-needed-I-dua(x,{rdfe})
    else
      new-dfe = [empty,rc1,⟨var,def,⟨rc2,val⟩⟩]
      add-to-soln-and-worklist-if-needed-I-dua(ecfi,{new-dfe})
    return
  if ( ecfi is empty )
    add-to-soln-and-worklist-if-needed-I-dua(n.successor,{rdfe})
    return
  // else ecfi represents an exception
  x = exit of innermost-enclosing-exception-block(n)
  add-to-soln-and-worklist-if-needed-I-dua(x,{rdfe})
else
  x = exit of innermost-enclosing-exception-block(n)
  add-to-soln-and-worklist-if-needed-I-dua(x,{rdfe})
}
```

Figure G.13: Phase I-dua transfer function for return-site

G.11 Auxiliary Functions used by Phase I-dua Transfer Functions

G.11.1 generate-*DUA-dfelms*-a -due-to-pot-non-aliases

```
1.a generate-DUA-dfelms-a-due-to-pot-non-aliases(n, rdfe) {
/*** rdfe is DUA-dfelm-a [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩] ***/
/*** n is an assignment node ***/
  generated-dfes = φ
  if ( n.lhs does not have dereference )
    return { rdfe }
  Let n.lhs be p→f, where the declared type of p is A*
  if ( var is u.f where u is an unknown initial value and
      the declared type of u is compatible with A )
    for each ⟨ecfi,rc3,loc ⟩ in n.lhs-rc-loc-pairs
      if ( loc is v.f and v is an unknown initial value compatible with u )
        rc4 = rc1 ∧ rc3 ∧ ⟨(v neq u), empty, empty⟩
        new-dfe = [ecfi,rc4,⟨var,def,⟨rc2,val⟩⟩]
        generated-dfes = generated-dfes ∪ {new-dfe}
      else
        generated-dfes = generated-dfes ∪ {rdfe}
  else
    generated-dfes = generated-dfes ∪ {rdfe}
  return generated-dfes
}
```

```
16. process-return-statement-dua( n, rdfe ) {
if ( first case )
  if ( n represents the statement return var )
    /*** var is read in n ***/
    n.def-uses = n.def-uses ∪ [rc1,⟨var,def,n⟩]
  y = exit of n.method
  if ( innermost-enclosing-exception-block(n)  is body of n.method )
    /*** ecfi must be empty ***/
    add-to-soln-and-worklist-if-needed-I-pta(y,{rdfe})
    return
  new-dfe = [y,rc1,⟨var,def,⟨rc2,val⟩⟩]
  x = exit of innermost-enclosing-exception-block(n)
  add-to-soln-and-worklist-if-needed-I-dua(x,{new-dfe})
// else second case, need not propagate as the exception is overriden by the
// return statement
}
```

Figure G.14: Phase I-dua transfer function for return statement

G.11.2 Auxiliary functions for process-call-dua

```
/*** rdfe is DUA-dfelm [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩] ***/
4.a compute-p-uses-due-to-dynamic-dispatch(n, rdfe) {
/*** var is the receiver variable ***/
  if ( n does not represent a dynamic dispatch )
    return
  if ( val is an unknown initial value )
    for each possible target t with val as receiver
      cond = type constraint on typeof(val) under which
           t is invoked with val as receiver
      new-rc = rc1 ∧ rc2 ∧ ⟨empty, cond, empty⟩
      /*** add a new DUA-dfelm-e to n.def-uses ***/
      n.def-uses = n.def-uses ∪ {[new-rc,⟨var,def,(n,t)⟩]}
  else /*** val is a heap-name ***/
    let t be the target invoked with val as receiver
    new-rc = rc1 ∧ rc2
    n.def-uses = n.def-uses ∪ {[new-rc,⟨var,def,(n,t)⟩]}
}
```

```
4.b compute-new-bindings-of-defs-and-u-i-defs(n, rdfe) {
new-def-uidef-bindings = φ
if ( var is a global ) {
  4.b.1:  uidefs = unknown initial defs of var at the entry
        nodes of those targets in whose used-sets var is contained
  for each y ∈ uidefs
    if ( [ecfi,rc1,⟨⟨var,def⟩,y⟩] ∉ n.def-uidef-bindings )
      new-def-uidef-bindings =
        new-def-uidef-bindings ∪ {[ecfi,rc1,⟨⟨var,def⟩,y⟩]}
}
if ( var is obj.f ) {
```

```
/*** obj is a heap-name or an unknown initial value ***/
uivs = uidefs = ϕ
for each [ecfi,rc3,⟨obj,uiv1⟩] ∈ n.actual-to-uiv-bindings
  uivs = uivs ∪ {⟨ecfi,rc3,uiv1⟩}
for each ⟨ecfi,rc4,uiv2⟩∈ uivs
  if ( uiv2.f has been marked as read )
    uidefs = uidefs ∪ {⟨ecfi,rc4,uiv2.f_initdef⟩}
for each ⟨ecfi,rc5,uiv3.f_initdef⟩∈ uidefs
  rc6 = rc1 ∧ rc5
  if ([ecfi,rc6,⟨⟨obj.f,def⟩,uiv3.f_initdef⟩] ∉ n.def-uidef-bindings)
    new-def-uidef-bindings = new-def-uidef-bindings ∪
      {[ecfi,rc6,⟨⟨obj.f,def⟩,uiv3.f_initdef⟩]}
}

return new-def-uidef-bindings }
```

4.b compute-new-bindings-of-defs-and-u-i-defs : For example, when *rdfe* is [*empty*,*Empty-context*, ⟨*global1*,*method1.global*$1_{initdef}$,⟨*Empty-context*,*global*1_{init}⟩⟩] at statement **5** in Figure 1.5, *uidefs* (at program point **4.b.1**) is **{***A::update.global*$1_{initdef}$,*C::update.global*$1_{initdef}$**}**, i.e., *uidefs* contains the unknown initial defs of *global1* at the entry nodes of *A::update* and *C::update*. Recall Phase I-pta stores in *n.actual-to-uiv-bindings* the bindings between the unknown initial values at the entry nodes of targets of *n* and their values at *n*; with each binding the *ECFInfo* and relevant context under which the binding holds is also stored. For example, at program point **5** in Figure 1.5, *5.actual-to-uiv-bindings* is

{ [*empty*,⟨*empty*,(*type*($a1_{init}$)∈*A::update*),*empty*⟩,⟨*global*1_{init},*A::update.global*1_{init}⟩],
[*empty*,⟨*empty*,(*type*($a1_{init}$)∈*C::update*),*empty*⟩,⟨*global*1_{init},*C::update.global*1_{init}⟩],
[*empty*,⟨*empty*,(*type*($a1_{init}$)∈*A::update*),*empty*⟩,⟨*global*3_{init},*A::update.global*3_{init}⟩],
[*empty*,⟨*empty*,(*type*($a1_{init}$)∈*C::update*),*empty*⟩,⟨*global*3_{init},*C::update.global*3_{init}⟩],
[*empty*,⟨*empty*,(*type*($a1_{init}$)∈*A::update*),*empty*⟩,⟨$a1_{init}$,*A::update.this*$_{init}$⟩],
[*empty*,⟨*empty*,(*type*($a1_{init}$)∈*C::update*),*empty*⟩,⟨$a1_{init}$,*C::update.this*$_{init}$⟩] **}**.

Index

Vita

Ramkrishna Chatterjee

1993 M.E. in Computer Science and Engineering, First Class with Distinction, Ranked 1st, Indian Institute of Science, Bangalore, India.

1993-94 Associate Programmer, Tata Information Systems Ltd. (Tata-IBM), Bangalore, India.

1994-96 Teaching Assistant, Department of Computer Science, Rutgers, The State University of New Jersey.

1996-99 Research Assistant, Department of Computer Science, Rutgers, The State University of New Jersey.

1998 Ramkrishna Chatterjee, Barbara Ryder and William Landi. Complexity of Concrete Type-Inference in the Presence of Exceptions. In the Proceedings of European Symposium on Programming, April 1998.

1999 Ramkrishna Chatterjee, Barbara Ryder and William Landi. Relevant Context Inference. In Proceedings of ACM SIGPLAN/SIGACT Symposium on the Principles of Programming Languages, January 1999.

2000 Ph.D. in Computer Science, Rutgers, The State University of New Jersey.