Complexity of Points-to Analysis of Java in the Presence of Exceptions*

Ramkrishna Chatterjee

Oracle Corporation
One Oracle Drive
Nashua, NH 03062, USA
ph: +1 603 891 3515
{rachatte}@us.oracle.com

Barbara G. Ryder†

†Dept. of Computer Science
Rutgers University
110 Frelinghuysen Road
Piscataway, NJ 08854, USA
{ryder}@cs.rutgers.edu

William A. Landi‡

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

Abstract

At each program point, *points-to analysis* for statically typed object-oriented programming languages (e.g., Java, C^{++}) determines those objects to which a reference may refer (or a pointer may point) during execution. Points-to analysis is necessary for any semantics-based software tools for object-oriented systems. Our new complexity results for points-to analysis distinguish the difficulty of intraprocedural and interprocedural points-to analyses for languages with combinations of single-level types (i.e., types with data members only of primitive type), exceptions with or without subtyping, and dynamic dispatch. Our results include:

- The first polynomial-time algorithm for points-to analysis in the presence of exceptions that handles a robust subset of Java without threads and can be applied to C^{++};
- Proofs that the above algorithm is safe in general and provably precise on programs with single-level types and exceptions without subtyping, but not dynamic dispatch; thus this case is in **P**;
- Proof that interprocedural points-to analysis problem with single-level types and exceptions with subtyping, but without dynamic dispatch, is **PSPACE-hard**, while the intraprocedural problem is **PSPACE-complete**.

Other complexity characterizations of points-to analysis in programs without exceptions are presented, including an algorithm with worst-case bound of $O(n^5)$, which improves over the $O(n^7)$ worst-case bound achievable from previous approaches of Reps, Horwitz and Sagiv [RHS95] and Landi and Ryder [LR91].

Keywords: Points-to Analysis, Java, Exceptions, Complexity.

*The research reported here was supported, in part, by NSF grants GER-9023628, CCR97-04703 and the Hewlett-Packard Corporation. An abridged version of this paper appeared as [CRL98].

1 Introduction

Compile-time analysis of statically-typed object-oriented programming languages (e.g., Java, C++) is rendered difficult by several factors. To trace flow of control through call statements which use dynamic binding (e.g., a virtual call in C++, a call to a non-static Java method) requires knowledge of the possible receiver object types, since the run-time type of the receiver object is used to determine the method which is called. In addition, user-thrown exceptions in Java codes result in flow of control through the program that is dependent on the run-time type of exception objects. Use of exception subtyping can make matching exception objects to specific catch clauses handling them quite difficult at compile-time.

At each program point, *points-to analysis*, which we refer to as *PTA* hereafter, determines those objects to which a reference may refer (or a pointer may point) during execution. By summarizing those objects to which an object reference may refer, points-to analysis bounds the type of that object reference at compile-time.[1] This enables better estimate of possible interprocedural execution flow through dynamically bound calls.

Analyses must be performed interprocedurally in object-oriented codes, since the key operation is method application, and almost all flow of control is interprocedural. Even more than in imperative languages (e.g., C, Pascal), this requires a good approximation of calling context in order to keep information propagation to possible execution paths, and thus preserve analysis precision.

Points-to analysis information is used as a first pass by many compile-time analyses to obtain information that is crucial for software engineering tools. These analyses include:

- side-effect analysis [LRZ93, CBC93, MLR+93,

[1]Note: we are assuming a whole-program points-to analysis here, which is inadequate to handle dynamic class creation or loading.

EGH94, GH98, Ruf95, HP98, SH97a, ZRL98, ZRL96],

- data-flow-based testing [HS91, BH93, HFGO94, Ost90, Wey94, CR99, FW93, FI98],
- interprocedural def-use[2] associations [PLR94, HS94, GH98, CR99] and
- effective static interprocedural program slicing [GL91, HRB90, OO84, RR95, GS96, LH96, HC98, SHR99, TCFR96, Tip96, Ven91, Wei84, TAFM97, AG96, AG98].

Side-effect analysis allows the building of semantic code browsers which can answer questions such as *Where was this variable's value set previous to this use?* or *Where will this newly-written value be read?* Side effect information is effectively *may write* information; therefore it is useful to regard this information as indicating its complement, namely *must preserve* information. This allows maintenance programmers to be certain that specific method calls will not modify certain variables, valuable information when enhancing code written by others. Program slicers are tools that use interprocedural def-use relations plus control dependence information to focus programmer attention on code that affects some computation of interest. Static slicers work from summary descriptions of all possible executions of the code, whereas dynamic slicers refine an execution trace through the code. Both types of slicers can use compile-time information to present to the user to enhance his/her understanding of the program. For a language such as Java, program slicers must deal with control flow along exception handling paths[SH98], because exceptions are ubiquitous. Points-to analysis of exception object references allows more precise modelling of potential control flow in handling exceptions.

In data-flow-based testing, it is assumed that forcing execution through some possible def-uses in a program will increase our ability to find bugs, since the flow of values from writes to reads is not necessarily exercised in white box testing. As shown in Figure 1, we need to know the solution for *PTA* to find def-uses. `B` is a derived class of `A` and the method `update` defined in `A` is overriden in `B`. The statement *3* is a dynamically dispatched call site. Syntactically, statement *3* can either invoke `A::update` or `B::update` depending upon the run-time type of the object to which `var` points at statement *3*. Thus, without *PTA* it is not clear whether there exists a def-use association between statements *1* and *5*. The solution of *PTA* tells us that at the top of statement *3*, `var` can only point to the object allocated at statement *2*. Since the type of the object allocated at statement *2* is `B`, it can be inferred that only `B::update` gets invoked at statement *3* and hence there is no def-use association between statements *1* and *5* (while there is a def-use association between statements *4* and *5*).

```
class A {
  public void update() { }
};
class B extends A {
  public void update() {
    4:  test.global1 = 2;
  }
}
class test {
  public static int global1;
  public void method1() {
    A var;
    int x;
    1:  test.global1 = 1;
    2:  var = new B();
    3:  var.update();
    5:  x = test.global1;
  }
}
```

Figure 1: Finding def-uses with *PTA*

Increasingly, systems are being built from frameworks, which are essentially packages constructed to be extended. Such packages often use exceptions to provide alerts of unexpected behaviors. For a programmer to understand how to write appropriate exception handlers, it is necessary to have some understanding of the use of exceptions within that package. Recent research by Robillard and Murphy[RM99] is aimed at providing information for a program understanding tool which allows browsing of potential exception flow through a Java program.

Software engineers working with object-oriented applications need to understand the necessary role played by points-to analysis in uncovering possible control flow through their systems; however, the tradeoff between the precision required of analysis and its cost depends on its intended application. Our previous work has presented practical points-to and def-use analyses for Java without threads and C++ [CRL99, Cha99]; this paper presents results which classify the difficulty of *PTA* for different models of object-oriented languages containing different sorts of constructs, thus explaining why polynomial-time techniques will necessarily be of approximate precision.

Main Contributions. The main contributions of this paper are:

- The first polynomial-time algorithm for *PTA* in the presence of exceptions that handles a robust subset of Java without threads, (and C++[3]),

[2] Intuitively, there exists a def-use association between two program points if there exists a program execution for which a value is written into a memory location at the first program point and then subsequently read from that location at the second program point.

[3] We present our algorithm only for Java; changes needed for

- Proofs that the above algorithm is always safe and provably precise on programs with single-level types, exceptions without subtyping, and without dynamic dispatch; thus this case is in **P**,

- Proof that intraprocedural *PTA* for programs with single-level types and exceptions with subtyping is **PSPACE-complete**, while the interprocedural problem (even) without dynamic dispatch is **PSPACE-hard**.

- New complexity characterizations of *PTA* in the absence of exceptions.

These results are summarized in Table 1 which also gives the corresponding sections of the paper.

1.1 Analysis Concepts

1.1.1 Flow- and Context-sensitivity.

A flow-insensitive algorithm ignores the ordering of statements within a method; by contrast, a flow-sensitive algorithm follows the control flow order of statements within a method, and computes different solutions for a variable at distinct program points. The example in Figure 2 illustrates this difference. We represent objects created at statement `i` by $object_i$. At statement *s2*, a flow-sensitive algorithm kills the assignment made to `p` at statement *s1* and thus, computes program-point-specific solutions. On the other hand, a flow-insensitive algorithm ignores the ordering of statements *s1* and *s2* (or considers all possible orderings of *s1* and *s2*) and thus, computes the same solution for all program points in `method`.

A context-sensitive algorithm considers (sometimes approximately) only interprocedurally *realizable paths* [RHS95, LR91, SP81]: paths along which calls and returns are properly matched; a context-insensitive algorithm does not make this distinction as the example in Figure 3 illustrates. A path which goes from call site *c1* to the entry node of `method1` and then comes back to call site *c1* from the exit node of `method1` is an example of a realizable path, as calls and returns are properly matched along this path. In contrast, a path which goes from call site *c2* to the entry node of `method1` and then comes back to call site *c1* from the exit node of `method1` is an example of an unrealizable path, as calls and returns are not properly matched along this path. We will use the term *flow-sensitive* to refer to a flow- and context-sensitive analysis.

To solve the problem of *PTA* precisely requires both a flow- and context-sensitive technique; however, there are approaches with varying degrees of flow and context sensitivity for this problem. Although some of these have been used for pointer analysis of C, they can be adapted for *PTA* of Java without exceptions

C++ are briefly discussed in Section 9.

```
class B { };
class C extends B { };
class A {
    public void method( ) {
      B p;
      s1:  p = new B();
      // solution computed by a flow-insensitive
      // algorithm: ⟨p,object_s1 ⟩, ⟨p,object_s2 ⟩
      // solution computed by a flow-sensitive
      // algorithm: ⟨p,object_s1 ⟩
      s2:  p = new C();
      // solution computed by a flow-insensitive
      // algorithm: ⟨p,object_s1 ⟩, ⟨p,object_s2 ⟩
      // solution computed by a flow-sensitive
      // algorithm: ⟨p,object_s2 ⟩
    }
}
```

Figure 2: Illustration of flow sensitivity

```
class B { };
class A {
    public static B method1( B param ) {
      return param;
    }
    public static void method2( ) {
      B p,q;
      l1:  p = new B();
      c1:  q = method1( p );
      // solution computed by a context-sensitive
      // algorithm: ⟨q, object_l1 ⟩
      // solution computed by a context-sensitive
      // algorithm:
      // ⟨q, object_l1 ⟩, ⟨q, object_l2 ⟩
      // Note: ⟨q, object_l2 ⟩results from the
      // unrealizable path:
      // method3 → method1 → method2
    }
    public static void method3( ) {
      B p,q;
      l2:  p = new B();
      c2:  q = method1( p );
      // solution computed by a context-sensitive
      // algorithm: ⟨q, object_l2 ⟩
      // solution computed by a context-sensitive
      // algorithm:
      // ⟨q, object_l1 ⟩, ⟨q, object_l2 ⟩
    }
}
```

Figure 3: Illustration of context sensitivity

results	paper section	single-level types	exceptions without subtypes	exceptions with subtypes	dynamic dispatch
interprocedural *PTA* in **P**, $O(n^7)$	sec 6	x	x		
intraprocedural *PTA* **PSPACE**-complete	sec 5	x		x	
interprocedural *PTA* **PSPACE**-hard	sec 5	x		x	
interprocedural *PTA* **PSPACE**-hard	sec 5	x			x
interprocedural *PTA* in **P**, $O(n^5)$	sec 4	x			
intraprocedural *PTA* in **NC**	sec 4.3	x			

Table 1: Complexity results for *PTA* summarized

and threads, or C++ without exceptions. At the one end of the spectrum are flow- and context-insensitive approaches [Cou86, Gua88, BCCH94, And94, SH97b, Ste96a, Wei80, ZRL96, HP98], which are the least expensive, but also the most imprecise. At the other end are flow- and context-sensitive approaches [Coo89, Gua88, GH98, WL95, Ruf95, HA90, SFRW90, CRL99, LR92, EGH94, WL95, CBC93, MLR+93, Ruf95], which are the most precise, but also the most expensive. Approaches like [PS91, PC94, Age95] are in between the above two extremes.

1.1.2 Exceptions.

We are interested in a flow-sensitive algorithm for *PTA* of a robust subset of Java with exceptions, but without threads (this subset is described in Section 2). None of the flow-sensitive pointer analysis algorithms for C/C++ mentioned previously handle exceptions; however, unlike in C++, exceptions are *frequently* used in Java programs, making it an *important* problem for Java. Consider the example shown in Figure 4. This example consists of two classes: a `driver` class and a `lib` (library) class. The driver method `method_drv` calls the library method `lib::display` which performs check on the size of the object to be displayed and then displays it provided the object is of proper size. If the object it too big, `lib::display` throws an exception whose run-time depends on the run-time type of the object to be displayed. Syntactically, call site *2* can throw an exception of type `E` or `E1`. As a result, without data-flow analysis it is not possible to determine whether there exists a def-use relationship between statements *4* and *5*. However, using data-flow analysis (i.e., *PTA* that takes into account flow due to exceptions), it can be determined that call site *2* can only throw an exception of type `E1`, and only the def-use associations between statements *3* and *5* and statements *1* and *5* are possible. If we use the simplistic alternative and replace `catch`es using if-then-else statements, the def-use association between statements *4* and *5* would be reported as a possible def-use association. Thus, although replacing `catch`es with if-then-else statements will result in a safe solution, such a solution may not be precise enough for many software engineering applications (such as data-flow-based testing).

1.1.3 Complexity Analysis.

For systematic algorithm development, complexity characterizations are needed to isolate difficult constructs, such as exceptions, and to indicate approximations that any efficient algorithm has to make. Experience shows that in many cases complexity analysis provides ideas for designing efficient algorithms. In this paper, first we prove hardness results which isolate difficult features; then, we use the insight gained through complexity analysis to design a polynomial-time algorithm which computes a safe solution in all cases and the precise solution in some restricted cases, thus characterizing these polynomial-time solvable cases.

Figure 5 shows the hierarchy of complexity classes [Pap94]. *NC* is the class of *efficiently parallelizable* problems and *P-Space* consists of problems that can be solved using polynomial amount of space. It is known that *NC* is a proper subset of *P-Space*; apart from this, it is not known whether any other containment shown in the figure is proper or not. Similar to *NP-complete* problems there are *P-Space-complete* and *P-complete* problems, these are the hardest problems in their respective classes such that if an efficient algorithm can be found for a complete problem then all problems in the corresponding complexity class can be solved efficiently.

```
class E extends Exception {};
class E1 extends E {};
class A {
  public void throw_excp() throws E {
    E excp; excp = new E();
    throw excp;
  }
  ...
};
class B extends A {
  public void throw_excp() throws E {
    E1 excp; excp = new E1();
    throw excp;
  }
  ...
};

class lib {
  public static void display(A p)
        throws E {
    if (p.size() ≥ 100)
      p.throw_excp();
    else
      p.display();
  }
};

class driver {
  public void method_drv() {
    B var;
    int d,u;
    1:  d = 1;

    var = new B();
    try {
      2:  lib.display(var);
    }
    catch(E1 e1) { 3:  d = 2; }
    catch(E e) { 4:  d = 3; }
    5:  u = d;
  }
}
```

Figure 4: Flow due to exceptions

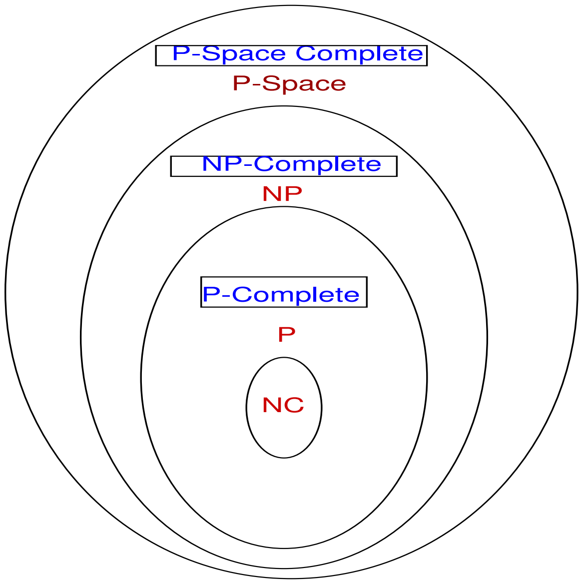

Figure 5: Hierarchy of complexity classes

1.2 Paper Outline

The rest of this paper is organized as follows. First, we present a flow-sensitive algorithm for *PTA* in the absence of exceptions, the *basic* algorithm, and prove our results about complexity of *PTA* in the absence of exceptions. Next, we prove **PSPACE**-hardness results about *PTA* in the presence of exceptions. Finally, we present an extension of the *basic* algorithm for *PTA* in the presence of exceptions, and prove its complexity and correctness.

2 Basic definitions

2.1 Program representation

Our algorithm operates on an interprocedural control flow graph or ICFG [LR91]. An ICFG contains a control flow graph (CFG) for each method in the program. Each statement in a method is represented by a node in the method's CFG and edges between the nodes represent control flow between the corresponding statements. Each method's CFG has an *entry-node* representing the entry point of the method and an *exit-node* representing the exit point of the method. A *pointer-assignment-node* represents an assignment statement that updates a location of reference type. Any other kind of assignment statement (which is pass-through for *PTA*) is represented by a *non-pointer-assignment-node*. The test expression of an *if* or *while* statement is represented by a *condition-node*, which has two successors: the node that represents the statement which is executed if the

the condition is true and the node that represents the statement which is executed if the condition is false. Each call site is represented using a pair of nodes: a *call-node* and a *return-node*. Information flows from a call-node to the entry-node of a target method and comes back from the exit-node of the target method to the return-node of the call-node. Due to dynamic dispatch, interprocedural edges are constructed iteratively during data-flow analysis as in [EGH94]. Details of this construction are shown in Figure 19 of Appendix A. We will denote the entry-node of *main* by *start-node*.

2.2 Representation of dynamically created objects

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, Ste96b], we represent all the run-time objects (or arrays) created at a program point n with the single name $object_n$, a **heap-name**. No distinction is made between different elements of an array. Thus, if an array is created at n, $object_n$ represents all the elements of the array.

2.3 Points-to

A points-to has the form $\langle var,obj \rangle$; where *var* is one of the following: (1) a static variable of reference type, (2) a local variable of reference type, (3) $object_m$ - an array object created at program point m or (4) $object_n.f$ - field f, of reference type, of an object created at a program point n; and *obj* is $object_s$ - an object created at a program point s.

2.4 Precise solution for *PTA*

A *reference variable* is one of the following:

1. a static variable (class variable) of reference[4] type;
2. a local variable of reference type;
3. Av, where Av is $V[t_1]...[t_d]$, and
    - V is a static/local variable or V is an array $object_n$ allocated at program point n, such that V is either a d-dimensional array of reference type or an array of any type having more than d dimensions and
    - each t_i is a non-negative integer; or
4. $V.s_1...s_k$, where
    - V is either a static/local variable of reference type or V is Av or V is object $object_n$ created at program point n,
    - for $1 \leq i \leq k$, each $V.s_1...s_{i-1}$ (V for $i = 1$) has the type of a reference to a class T_i and each s_i is a field of reference type of T_i or $s_i = f_i[t_{i_1}]...[t_{i_{r_i}}]$ and f_i is a field of T_i and f_i is an array having at least r_i dimensions and each t_{i_j} is a non-negative integer, and
    - $V.s_1...s_k$ is of reference type.

[4] may refer to an instance of a class or an array.

```
class A {
  A f1;
  A f2[];
}
class rv {
  public void method() {
    A v1[][];
    A v2;

    1:  v1 = new A[2][3];
    2:  v2 = new A();
  }
}
```

Figure 6: Examples of reference variables

For example, consider the program fragment in Figure 6. *v1[1][2]* and $object_1$*.[1]* are examples of reference variables of the second kind (Av) described above. *v2.f1*, *v2.f2[1].f1*, $object_2$*.f1* and $object_2$*.f2[2].f1* are examples of reference variables of the fourth kind described above.

Using these definitions, the precise solution for *PTA* can be defined as follows: given a *reference variable RV* and an object $object_n$, $\langle RV,object_n \rangle$ belongs to the precise solution at a program point n if and only if RV is visible at n and there exists an executable path from the *start-node* of the program to n such that if this path is followed, RV points to $object_n$ at n (i.e., at the top of n). 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 paper, we use *precise* to mean *precise up to symbolic execution.*

2.5 Safe solution

An algorithm is said to compute a *safe* solution for *PTA* if and only if at each program point, the solution computed by the algorithm is a superset of the precise solution.

2.6 Single-level type

A single-level type is one of the following: (1) a primitive type defined in [GJS96] (e.g., int, float etc.), (2) a class that has all non-static data-members of primitive types (e.g., `class A { int i,j; }`) or (3) an array of a primitive type.

Remark. Since any class in Java has to extend the predefined class *Object* and *Object* is not a single-level type, strictly speaking no class in Java is a single-level type. When we use the phrase *programs with only single-level types* in this paper, we mean programs that only make single-level extensions to predefined classes (like *Object*) and do not utilize non-single-level aspects of these predefined classes. Formally these are programs in which all the user-defined types are single-level types if the predefined classes are assumed to be single-level types, and in which no user-defined variable of reference type is assigned the value of a non-static field of reference type of a predefined class, either directly or transitively through static fields of predefined classes. Further, for these programs, we assume the predefined classes to be single-level types for the purpose of analysis and the definition of the precise solution for PTA, (i.e., both in our analysis as well as in the definition of the precise solution, we ignore (1) the non-static fields of reference type of these predefined classes and (2) the static fields of reference type of these predefined classes that are assigned the value of a non-static field of reference type of a predefined class, either directly or transitively).

2.7 Subtype

We use Java's definition of subtyping: a class A is a subtype of another class B if A *extends* B, either directly or indirectly through inheritance.

2.8 Exception Block

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

2.9 Innermost Enclosing Exception Block

Given a statement S, *innermost-enclosing-exception-block(S)* is the innermost *exception block* containing S.

2.10 Subset of Java Considered

We essentially consider a subset that excludes threads, but in some cases we may need to exclude three other features: finalize methods, static initializations and dynamically defined classes. We will refer to this subset as *JavaWoThreads.*

Since finalize methods are called (non-deterministically) during garbage collection or unloading of classes, we exclude *finalizers* that modify reference-type locations accessible outside the *finalizers*. Static initializations complicate analysis due to dynamic loading of classes. We exclude static initializations that depend upon the order in which files are loaded (extremely rare). Our algorithm can handle only those static initializations for which it is safe (with respect to PTA) 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, our algorithm cannot handle any class whose code is constructed on the fly and not known statically.

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

3 PTA in the absence of exceptions

Our *basic* algorithm for PTA is an iterative worklist algorithm [KU76]. It operates on an ICFG and is similar to the Landi-Ryder algorithm [LR92] for alias analysis, but instead of aliases, it computes points-to's. In Section 6, we will extend this algorithm to handle exceptions.

3.1 Lattice for data-flow analysis

In order to restrict data-flow only to realizable paths, points-to's are computed conditioned on *assumed-points-to's* (akin to reaching alias in [LR92] [PR96]), which represent points-to's reaching the entry of a method, and approximate the calling context in which the method has been called. A points-to along with its *assumed-points-to* is called a *conditional-points-to.* A conditional-points-to has the form ⟨*condition, points-to*⟩, where *condition* is an assumed-points-to or *empty* (meaning this points-to is applicable to all calling contexts). Consider the example in Figure 7. References to $object_4$ and $object_7$ are passed as the value of *param* at call sites *5* and *8* respectively. Thus, the solution at the top of statement *1* consists of the following two conditional-points-to's: $\langle\langle param,object_4\rangle,\langle param,object_4\rangle\rangle$ and $\langle\langle param,object_7\rangle,\langle param,object_7\rangle\rangle$. Statement *1* copies the value of *param* to *global1*. Thus, at the top of statement *2*, we have the following conditional-points-to's which are respectively generated using the two conditional-points-to's in the solution at the top of statement *1*: $\langle\langle param,object_4\rangle,\langle global1,object_4\rangle\rangle$ (meaning *global1* points to $object_4$ at the top of statement *2* if *param* points to $object_4$ at the entry of *method1*) and $\langle\langle param,object_7\rangle,\langle global1,object_7\rangle\rangle$ (meaning *global1* points to $object_7$ at the top of statement *2* if *param* points to

```
class A { ... };
class cpt_example {
  public static A global1;
  public static A global2;
  public static void method1(A param) {
    1:  global1 = param;
    2:  global2 = new A();
    3:
  }
  public static void method2() {
    A local1;
    4:  local1 = new A();
    5:  method1( local1 );
    6:
  }
  public static void method3() {
    A local2;
    7:  local2 = new A();
    8:  method1( local2 );
    9:
  }
};
```

Figure 7: Example for conditional-points-to's

$object_7$ at the entry of *method1*). These two conditional-points-to's (representing the values of *global1*) are propagated from the exit node of *method1* to the call sites *5* and *8*, where the assumed-points-to's are used to figure out whether a conditional-points-to applies to the call site or not. For example, the assumed-points-to $\langle param,object_4\rangle$ holds at call stie *5*; thus, $\langle empty,\langle global1,object_4\rangle\rangle$ is propagated to program point *6*, meaning *global1* points to $object_4$ at program point *6* independent of the calling context of *method2*. In contrast, the assumed-points-to $\langle param,object_7\rangle$ does not hold at call site *5*; thus, the conditional-points-to $\langle\langle param,object_7\rangle,\langle global1,object_7\rangle\rangle$ does not apply to call site *5*. Next consider statement *2*. It generates the conditional-points-to $\langle empty,\langle global2,object_2\rangle\rangle$, because *global2* points to $object_2$ at program point *3* in all calling contexts of *method1*.

For simplicity, we will write $\langle$*empty,points-to*$\rangle$ as *points-to*. Also a special data-flow element *reachable* is used to check whether a node is reachable from the start-node through a realizable path. This ensures that only such reachable nodes are considered during data-flow analysis and only points-to's generated by them are put on the worklist for propagation. The lattice for data-flow analysis (associated with a program point) is a subset lattice consisting of sets of such conditional-points-to's and the data-flow element *reachable*.

3.2 Query

Using these conditional-points-to's, a query for *PTA* is answered as follows. Given a *reference variable* V and a program point l, the conditional-points-to's with *compatible* assumed-points-to's computed at l are combined to determine the possible values of V. Assumed-points-to's are *compatible* if and only if they do not imply different values for the same user-defined variable. For example, suppose V is *p.f1*, and the solution computed at l contains $\langle\langle q,obj1\rangle, \langle p,\ obj1\rangle\rangle$, $\langle\langle s,obj2\rangle, \langle obj1.f1,\ obj2\rangle\rangle$ and $\langle\langle q,obj3\rangle, \langle obj1.f1,\ obj3\rangle\rangle$. Then *obj2* is a possible value of V because the conditional-points-to's $\langle\langle q,obj1\rangle, \langle p,\ obj1\rangle\rangle$ and $\langle\langle s,obj2\rangle, \langle obj1.f1,\ obj2\rangle\rangle$ can be combined as their assumed-points-to's $\langle q,obj1\rangle$ and $\langle s,obj2\rangle$ are compatible. In contrast, *obj3* is not a possible value of V because the conditional-points-to's $\langle\langle q,obj1\rangle, \langle p,\ obj1\rangle\rangle$ and $\langle\langle q,obj3\rangle, \langle obj1.f1,\ obj3\rangle\rangle$ cannot be combined as their assumed-points-to's $\langle q,obj1\rangle$ and $\langle q,obj3\rangle$ are not compatible.

3.3 Algorithm description

Figure 9 contains a high-level description of the main loop of the *basic* algorithm. Here *node* is a method entry-node, a pointer-assignment-node, a non-pointer-assignment-node, a condition node, a call-node, a return-node or a method exit-node. *apply* computes the semantic effect of a node (which could be a method entry-node, a pointer-assignment-node, a non-pointer-assignment-node, a condition node or a return-node) on an incoming conditional-points-to. A method entry-node, a non-pointer-assignment-node, a condition node and a return-node are pass-through for *PTA*; thus, *apply* for these kinds of nodes simply returns the incoming conditional-points-to.

For a pointer-assignment-node, *apply* returns the set of conditional-points-to's generated by the corresponding pointer assignment statement as a result of the incoming conditional-points-to. For example, suppose node l represents the pointer assignment statement $p.f1 = q$, *ndf_elm* (i.e. the conditional-points-to reaching the top of l) is $\langle z, \langle p,\ object_s\rangle\rangle$ and $\langle u, \langle q,\ object_n\rangle\rangle$ is present in the solution computed so far at the top of l. Here z and u are two assumed-points-to's. Assuming z and u are compatible, *apply* generates $\langle object_s.f1,\ object_n\rangle$ under the condition that both z and u hold at the entry-node of the method containing l. Then either z or u is chosen as the assumed-points-to for the generated data-flow element. For example, if u is chosen then $\langle u, \langle object_s.f1,\ object_n\rangle\rangle$ will belong to the set returned by *apply*. Note that choosing only one of the assumed-points-to's does not compromise safety. When a conjunction of conditions is associated with a points-to, any fixed-size subset of these conditions may be stored without affecting safety. This is because at a call site where all the conjuncts are true, any subset of the conjuncts is also true. Thus the data-flow element will still be propagated to these call sites. However, this does approximate context sensitivity as the data-flow element will also be propagated to those call sites where only a proper subset of the conjuncts is true. At present, we store only the first member of the list of assumed-points-to's, other heuristics are possible. *apply* is defined in Appendix B.

add_to_solution_and_worklist_if_needed checks whether a data-flow element is present in the solution set (computed so far) of a node. If not, it adds the data-flow element to the solution set, and puts the node along with this data-flow element on the worklist.

process_exit_node propagates data-flow elements from the exit-node of a method to the return-node of a call site of this method. Suppose the conditional-points-to $\langle z,\ u\ \rangle$ (where z is an assumed-points-to and u is a points-to) belongs to the solution set at the exit-node of a method M. Consider a return-node R of a call site C of M. For each assumed-points-to x such that the conditional-points-to $\langle x,\ t\ \rangle$ is in the solution set computed so far at C and t implies (e.g., due to an actual to formal binding) z at the entry-node of M (i.e., $x \Rightarrow t \Rightarrow z \Rightarrow u$), $\langle x,u\ \rangle$ is propagated by *process_exit_node* to R. *process_exit_node* is defined in Figure 18. Consider the example in Figure 8. The conditional-points-to $\langle empty,\langle local3,object_6\rangle\rangle$ holds at call site *7*. This conditional-points-to implies the conditional-points-to $\langle\langle param2,object_6\rangle,\langle param2,object_6\rangle\rangle$ at the entry-node of *method2*, which in turn implies the conditional-points-to $\langle\langle param2,object_6\rangle,\langle local2,object_6\rangle\rangle$ at the call site *4*, which in turn implies the conditional-points-to $\langle\langle param1,object_6\rangle,\langle param1,object_6\rangle\rangle$ at the entry-noe of *method1*, which in turn implies the conditional-points-to $\langle\langle param1,object_6\rangle,\langle global,object_6\rangle\rangle$ at program point *2*, the exit-node of *method1*. Thus, *proces_exit_node* substitutes the assumed-points-to $\langle param1,object_6\rangle$ by the assumed-points-to $\langle param2,object_6\rangle$ and propagates the conditional-points-to $\langle\langle param2,object_6\rangle,\langle global,object_6\rangle\rangle$ from the exit-node of *method1* to the return-node of call site *4*.

The pseudocode for *process_exit_node* is given in Figure 18 in Appendix A.

process_call_node propagates data-flow elements from a call site to the entry-node of a method callable from this site. Due to dynamic dispatch, the set of methods callable from a call site is iteratively computed during the data-flow analysis as in [EGH94]. Suppose the conditional-points-to $\langle x,\ t\ \rangle$ holds at a call site C which has a method M in its set of callable methods computed so far. If the points-to t implies a points-to z at the entry-node of M (e.g., through an actual to formal binding), the conditional-points-to $\langle z,z\ \rangle$ is forwarded to the entry-node of M. *process_call_node* also remembers at C the association between the assumed-points-to x and the assumed-points-to z because this is used by *process_exit_node* as described above. *process_call_node* is defined in Figures 19 and 20. Consider the exam-

```
class A { ... };
class cont_sensitivity_example {
  public static void method1(A param1) {
    1: global = param1;
    2:
  }

  public static void method2( A param2) {
    A local2;
    3: local2 = param2;
    4: method1(local2);
    5:
  }

  public static void method3( ) {
    A local3;
    6: local3 = new A();
    7: method2(local3);
    8:
  }
}
```

Figure 8: Context-sensitivity through conditional-points-to's

ple in Figure 10. Initially, the set of methods invocable from call site *3* is ϕ. When the conditional-points-to $\langle empty,\langle p,object_1\rangle\rangle$ reaches call site *3* from statement *1*, *B::foo* is added to the set of methods invocable from call site *3*. Similarly, when the conditional-points-to $\langle empty,\langle p,object_2\rangle\rangle$ reaches call site *3* from statement *2*, *C::foo* is added to the set of methods invocable from call site *3*. Thus, when fixed point is reached, the set of methods found to be invocable from call site *3* is $\{B::foo, C::foo\}$.

The pseudocode for *process_call_node* is given in Figure 19 in Appendix A.

3.4 Example for the *basic* algorithm

The following self-explanatory example shows some of the points-to solution obtained from the *basic* algorithm.

```
// Note: due to lack of space only a part of the solution is shown

class A {};

class B {
    public B field1;
};

class C {
    public B field1;
};

class base {
 public static A a;
 public void method( ){
   l1:  a = new A();

   // ⟨empty,⟨a,object_l1⟩⟩
   exit_node:
 };
};

class derived extends base {
 public void method( ) {
   // overrides base::method
   l2:  a = new A();

   // ⟨empty,⟨a,object_l2⟩⟩
   exit_node:
 };
};

class caller {
  public static void call( base param ) {
    // ⟨⟨param,object_l4⟩,⟨param,object_l4⟩⟩,
    // ⟨⟨param,object_l5⟩,⟨param,object_l5⟩⟩,

    l3:  param.method();

    // ⟨param,object_l4⟩ => base::method is called.
    // ⟨param,object_l5⟩ => derived::method is called.
    //⟨empty,⟨a,object_l1⟩⟩ is changed to
    // ⟨⟨param,object_l4⟩,⟨a,object_l1⟩⟩ because base::method is
    // called only when ⟨param,object_l4⟩.
    // ⟨empty,⟨a,object_l2⟩⟩ is changed to
    // ⟨⟨param,object_l5⟩,⟨a,object_l2⟩⟩ because derived::method is
    // is called only when ⟨param,object_l5⟩.
  };
}; // end of class caller

class potpourri {
  public static void example( C param ) {
    C local;
    // Let S = { ⟨⟨param,object_l6⟩,⟨param,object_l6⟩⟩,
    // ⟨⟨param,object_l7⟩,⟨param,object_l7⟩⟩,
    // ⟨⟨object_l6.field1,null⟩,⟨object_l6.field1,null⟩⟩,
    // ⟨⟨object_l7.field1,null⟩,⟨object_l7.field1,null⟩⟩}
    // solution at this point is S
    local = param;

    // Let S1 = S U {⟨⟨param,object_l6⟩,⟨local,object_l6⟩⟩,
    //               ⟨⟨param,object_l7⟩,⟨local,object_l7⟩⟩ }
    // solution at this point is S1
    l8:  local.field1 = new B();

    // solution =
    // S1 U { ⟨⟨param,object_l6⟩,⟨object_l6.field1,object_l8⟩⟩,
    //        ⟨⟨param,object_l7⟩,⟨object_l7.field1,object_l8⟩⟩,
    //        ⟨empty,⟨object_l8.field1,null⟩⟩ }
    exit_node:
  };
};

class test {
  public void test1( ) {
     base p;
     l4:  p = new base();

     // ⟨empty,⟨p,object_l4⟩⟩
     l9:  caller.call(p);

     // ⟨empty,⟨p,object_l4⟩⟩ .
     // At l9 ⟨p,object_l4⟩
     // => ⟨empty,⟨a,object_l1⟩⟩.
     exit_node:
  };

  public void test2( ) {
     derived p;
     l5:  p = new derived();

     l10:  caller.call(p);

     // ⟨empty,⟨p,object_l5⟩⟩ .
     // At l10 ⟨p,object_l5⟩ =>
     // ⟨empty,⟨a,object_l2⟩⟩.
     exit_node:
  };

  public void test3( ) {
     C q;
     l6:  q = new C();
     potpourri.example( q );
  };

  public void test4( ) {
     C q;
     l7:  q = new C();
     potpourri.example( q );
  };
};
```

```
class A {
  public void foo() { ... };
};
class B extends A {
  public void foo() { ... };
};
class C extends A {
  public void foo() {...};
};
class test {
  public void method() {
    A p;

    if ( ... )
      1:  p = new B();
    else
      2:  p = new C();

    3:  p.foo();
  }
};
```

Figure 10: Iterative computation of targets at a dynamically dispatched call site

4 *PTA* for programs with only single-level types and without dynamic dispatch, exceptions or threads

In this section, we show that the *basic* algorithm computes the precise solution for programs with only single-level types and without dynamic dispatch, exceptions or threads. We will show that the *basic* algorithm's worst-case complexity in this case is $O(n^5)$, hence this case is in P. This an improvement over $O(n^7)$ worst-case bound achievable by applying previous approaches of [RHS95] and [LR91] to this case.

4.1 Precision

definition 1 *Let l be a program point in a method M. rp(r,t,p,l,M) denotes a realizable path $r = t + p$ [5] from the start-node to l, such that p is a balanced [6] path from the entry-node of M to l and t is a realizable path from the start-node to the entry-node of M.*

Lemma 1 *Let l be a program point in a method M and r be a realizable path from the start-node to l. Then r can be written as $r = t + p$, where t is a realizable path from the start-node to the entry-node of M and p is a balanced path from the entry-node of M to l.*

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

[5] + indicates concatenation.

[6] Every entry-node along it has a matching exit-node except possibly for the entry-node of M [RHS95].

Lemma 2 *Let l be a program point in a method M*

1. *If $\langle reachable \rangle$ is computed by the algorithm at l, there exists a realizable path r from start-node to l.*
2. *If a conditional-points-to $\langle z,\ u \rangle$ [7] is computed by the algorithm at l, then*
   - *there exists a rp(r,t,p,l,M) such that u holds at l, and*
   - *for any rp(q,s,p,l,M), u holds at l if and only if z holds at the first node of p.*

proof The two claims can be proved simultaneously by induction on the number of iterations needed to compute $\langle z,u \rangle$ or $\langle reachable \rangle$. □*Remark:* For single-level types, a conditional-points-to at a program point can never require the simultaneous occurrence of multiple conditional-points-to's at (any of) its predecessors. This is why the above proof works. Note that multi-level types, dynamically dispatched calls or exceptions with subtyping violate this condition and hence the above lemma is not true in their presence.

Lemma 3 *Suppose there exists an rp(r,t,p,l,M). The following hold with respect to it.*

1. *$\langle reachable \rangle$ is computed by the algorithm at l.*
2. *When r is followed, if a points-to u holds at l, then*
   - *there exists a points-to z such that $\langle z,\ u \rangle$ is computed by the algorithm at l, and*
   - *for any rp(q,s,p,l,M), u holds at l if and only if z holds at the first node of p.*

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

Theorem 1 *The* basic *algorithm computes the precise solution for programs with only single-level types and without dynamic dispatch, exceptions or threads.*

proof Lemma 2 (claim 2) implies that the solution computed by the basic algorithm is a subset of the precise solution, while lemma 3 (claim 2) and lemma 1 imply that the precise solution is a subset of the solution computed by the algorithm. Hence we have the theorem. □

4.2 Complexity

In this section, we show that the complexity of the *basic* algorithm for programs with only single-level types and without dynamic dispatch is $O(n^5)$. We will need the following lemma:

[7] z could be empty.

```
// initialize worklist. Each worklist node contains a data-flow element, which
// is a conditional-points-to or reachable, and an ICFG node.
create a worklist node containing the entry-node of main
and reachable, and add it to the worklist;

while ( worklist is not empty ) {
  WLnode = remove a node from the worklist;
  ndf_elm = WLnode.data-flow-element;
  node = WLnode.node;

  if ( node ≠ a call_node and node ≠ exit_node of a method ) {
    // compute the effect of the statement associated with node on ndf_elm.
    generated_data_flow_elements = apply( node, ndf_elm );

    for ( each successor succ of node ) {
      for ( each df_elm in generated_data_flow_elements )
        add_to_solution_and_worklist_if_needed( df_elm, succ );
    }
  } // end of if

  if ( node is an exit_node of a method )
    process_exit_node( node, ndf_elm );
  if ( node is a call_node )
    process_call_node( node, ndf_elm );
} // end of while
```

Figure 9: High-level description of the *basic* algorithm

Lemma 4 *Suppose* $\langle\langle var1, obj1\rangle, \langle var2, obj2\rangle\rangle$ *is computed by the* basic *algorithm at a program point* l*. Then obj1 and obj2 are the same.*

proof is by induction on the number of iterations needed to compute the conditional-points-to. □

Let the total number of statements in the program be n_1, the sum of the numbers of arguments passed at call sites be n_2, n be $n_1 + n_2$, the maximum number of arguments passed at a call site be A, the total number of user-defined variables of reference type be V, the total number of dynamically created objects (identified by their creation sites) be L and the total number of call sites be C.

Now consider a program point l. There could be at most V user-defined variables of reference type visible at l. Each of them may point to at most L different objects. Each such points-to may be implied by at most V assumed-points-to's at the entry-node of the method containing l, this follows from lemma 4. This implies that the solution set at l may contain up to $O(V^2L)$ conditional-points-to's [8]. Suppose l is neither an exit-node nor a call-node. For each conditional-points-to reaching l, *apply* takes constant time and a constant amount of work is done along an edge to a successor of l. Since there are $O(n)$ edges, the total amount of work done for such nodes is $O(nV^2L)$.

The work done at call-nodes in passing data-flow elements to the entry-nodes of called methods is $O(C(A+1)V^2L)$ because each points-to at a call-node can imply at most $A+1$ points-to's at the entry-node of the called method. In addition $O(CV^2L)$ work may be done in passing points-to's representing values of local variables from call-nodes to their respective return-nodes (note that part of this work may be done at an exit-node).

For each conditional-points-to at an exit-node, the assumed-points-to may expand into $O(V)$ assumed-points-to's at a corresponding return-node, again we use lemma 4. So for each conditional points-to at an exit-node, up to $O(V)$ amount of work may be done along an edge to a corresponding return-node (note that part of this work may be done at a call-node). The total amount of work along such an edge is at most $O(V^3L)$ as there could be at most $O(V^2L)$ conditional-points-to's at an exit-node. There are C such edges. Thus, at most $O(CV^3L)$ amount of work is done along such interprocedural edges.

This means that the worst-case complexity (assuming V is at least 1) is $O((C(V+A)+n)V^2L)$. Now each of C, V, A and L is at most $O(n)$, hence the the worst-case complexity is $O(n^5)$. In [CR97], we present an example for which the algorithm attains this worst-case bound.

4.3 Complexity of intraprocedural *PTA* for programs with only single-level types and without dynamic dispatch, exceptions or threads

Theorem 2 *Intraprocedural PTA for programs with only single-level types and without dynamic dispatch, exceptions or threads 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] and **NC** is the class of efficiently parallelizable problems which contains non-

[8] The cost of propagating the data-flow element used to check reachability is at most $O(n)$, so we ignore it in our analysis.

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. □

5 Complexity of *PTA* in the presence of exceptions

In this section we show that intraprocedural *PTA* for programs with only single-level types and exceptions with subtyping is **PSPACE**-complete, while the interprocedural case without dynamic dispatch is **PSPACE**-hard.

5.1 Semantics of *finally*

Before describing our results about *PTA* in the presence of exceptions, we will briefly describe the semantics [GJS96] of a *finally* statement in Java as it is important for understanding the rest of this paper. The semantics of exception handling in Java is more complicated than other languages like C++ because of the *finally* statement. A *try* statement can optionally have a *finally* statement associated with it. 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 an exception thrown in the corresponding *try* statement or one of the corresponding *catch* statements, or leaving the *try* statement or one of its *catch* clauses by a *return*, (labelled) *break* or (labelled) *continue*, or by falling through. This reason is remembered on entering a *finally*, and unless the *finally* statement itself creates its own reason to exit the *finally*, at the exit-node of the *finally* this reason is used to decide control flow. If the *finally* itself 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*. Also, nested *finally* statements cause reasons for entering them to stack up.

5.2 PSPACE-hardness

In [Lan92a], Landi showed that intraprocedural may-alias analysis for a subset of C in the presence of pointers with at most 4 levels of indirection[9] and without dynamic allocation is **PSPACE**-complete. The only control-flow statements used in the proof are *if* and *while*. The same proof shows that intraprocedural *PTA* in the presence of pointers with at most 4 levels of indirection and without dynamic allocation (C-PTA for short)[10] is **PSPACE**-complete.

We will reduce C-PTA in polynomial time to the problem of *PTA* for programs with only single-level types (here we allow dynamic allocation because we are considering Java programs and the only way to initialize a reference variable is through dynamic allocation) and exceptions with subtyping (J-PTA for short). For the ease of presentation we will show the reduction for an instance of C-PTA with at most 2 levels of indirection. The reduction of an instance of C-PTA with higher levels of indirection is a straight forward 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 type of the parameter.

Consider an instance of C-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 J-PTA in Java is as follows:

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


class J-PTA {
  static void proc2( )
  {
    int_type r1, ..., rm;
```

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

[10] 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.

```
    int_type q1_var, ..., qn_var;
    q1_type q1_addr;
    ...
    qn_type qn_addr;

    multi_level_pointer p1, ..., ps;

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

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

    ...
  }
}
```

The statements in C-PTA are translated to equivalent statements in J-PTA 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 5 *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 $\langle q_i\_var,\ object\_l_j\ \rangle$ holds at f_k in proc2.*
2. *$\langle p_i,\ q_j\ \rangle$ holds at f_k if and only if $\langle p_i,\ object\_t_j\ \rangle$ holds at f_k in proc2.*

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

5.3 PSPACE-completeness

Now we will show that intraprocedural J-PTA is in **PSPACE**, and hence **PSPACE**-complete. Savitch's theorem [HU79] implies that **PSPACE= NPSPACE**[11]. Hence, it is enough to show that intraprocedural J-PTA is in **NPSPACE**. Our proof is similar to Landi's proof [Lan92a] that C-PTA is in **NPSPACE**. We present a non-deterministic algorithm for J-PTA, 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 $nn^n 2^m$ (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 $nn^n 2^{(n+2)^2}$. 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

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

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

Theorem 3 *Intraprocedural PTA 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.*

Using a construction similar to the one given above, we can show the following[12]:

Theorem 4 *PTA for programs with only single-level types and dynamic dispatch, and without exceptions or threads is* **PSPACE***-hard.*

In this construction [Cha99], the addresses of pointer variables 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.

This shows that in order to explore the additional complexity for *PTA* 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**.

6 What is solvable in polynomial-time ?

In this section, we show that the problem of *PTA* for programs with only single-level types and exceptions without subtyping, and without dynamic dispatch, is in **P**. First, we present an extension of the *basic* algorithm for *PTA* in the presence exceptions, then we prove that this extended algorithm computes the precise solution for programs with only single-level types and exceptions without subtyping, and without dynamic dispatch, and finally we prove that the extended algorithm's worst-case complexity for this case is $O(n^7)$, hence proving that this case is in **P**.

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 if there is a user-defined variable v of exception type e, no object whose type is a proper subtype of e (i.e. its class extends e) is allocated in the program.

6.1 Algorithm for *PTA* in the presence of exceptions

This algorithm is an extension of the *basic* algorithm described in Section 3. We will call this algorithm the *extended* algorithm.

6.1.1 Data-flow Elements.

The data-flow elements propagated by this extended algorithm have one of the following forms:

1. $\langle$*reachable*$\rangle$,
2. $\langle$*excp-type, reachable*$\rangle$,
3. $\langle$*label, reachable*$\rangle$,
4. $\langle z,u \rangle$,
5. $\langle$*excp-type*, z, $u\rangle$,
6. $\langle$*label*, z, $u\rangle$,
7. $\langle$**excp**, z, $obj\rangle$.

Here z is an assumed-points-to and u is a points-to. Note: the *basic* algorithm only used the first and fourth elements. The lattice for data-flow analysis associated with a program point is a subset lattice consisting of sets of these data-flow elements. In the rest of this section, we present definitions of these data-flow elements and a brief description of how these are propagated.

The first three kinds of data-flow elements are used for finding reachable nodes. As stated earlier, this increases precision by restricting flow to reachable nodes. As we will see later, for polynomial-time solvable cases, the extended algorithm finds the precise set of reachable nodes. For arbitrary programs, the set of nodes found reachable by the extended algorithm is a superset of the precise set of reachable nodes. Consider the example shown in Figure 11. The entry-node of *main* is reachable by default. Thus, the the entry-node of of *main* is initialized with $\langle$*reachable*$\rangle$. From the entry-node of *main* $\langle$*reachable*$\rangle$ is propagated to program point *11*

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

and from there ⟨*reachable*⟩ is propagated to call site *12*. From call site *12*, ⟨*reachable*⟩ is propagated to the entry-node of *method1*. From the entry-node of *method1*, ⟨*reachable*⟩ reaches program point *3* via program points *1* and *2*. Statement *3* throws $object_2$ which is of type *excp_t1*. Thus, statement *3* transforms ⟨*reachable*⟩ into ⟨*excp_t1,reachable*⟩ and propagates it to program point *4*, the exit-node of the try statement. The presence of ⟨*excp_t1,reachable*⟩ at program point *4* means that program point *4* is reachable from the entry-node of *main* along paths that have an uncaught exception of type *excp_t1*. In the absence of finally statements, there is never more than one active exception at a program point. Thus, only one exception type in a data-flow element is enough to summarize any uncaught exception along any path through which the data-flow element reaches the program point. As we shall see later, a call statement or a try statement nested inside a finally statement can cause exceptions to stack up; however, only one exception type in a data-flow element is still enough to summarize any uncaught exception because stacking of exceptions can be handled by remembering them in actual-to-formal bindings and by treating a try statement nested inside a finally statement like a call to an anonymous procedure. This is a *key aspect* of the extended algorithm, which makes it feasible.

At program point *4*, the exception type in a data-flow element (if any) is used to determine the future propagation of the the data-flow element (i.e., the extended algorithm mimics the run-time behavior of Java). Thus, ⟨*excp_t1,reachable*⟩ is propagated to the entry-node of the first catch statement as it catches any exception object whose type is *excp_t1* or whose type is a subtype of *excp_t1*. The entry node of the catch statement transforms ⟨*excp_t1,reachable*⟩ to ⟨*reachable*⟩ (as the exception has been caught by the catch statement) and propagates ⟨*reachable*⟩ to program point *5*, which in turn propagates ⟨*reachable*⟩ to program point *7*, the exit-node of *method1*. Note that program point *6* is unreachable and no reachability information is propagated to it. Thus, the conditional-points-to generated by statement *6*, ⟨*empty*,⟨*g*,$object_6$⟩⟩ is not considered for propagation and $object_6$ not reported as a value of *g* at any program point.

Considering all nodes as reachable would be safe but imprecise. From the exit-node of *method1*, ⟨*reachable*⟩ is propagated to the return-node of call site *12*, which in turn propagates ⟨*reachable*⟩ to call site *13*. From call site *13*, ⟨*reachable*⟩ is propagated to the entry node of *method2*, from where ⟨*reachable*⟩ reaches statement *9* via program point *8*. Statement *9* throws $object_8$ which is of type *excp_t2*. Thus, ⟨*excp_t2,reachable*⟩ is propagated to program point *10*, from program point *9*. The exit-node of *method2*, program point *10*, propagates ⟨*excp_t2,reachable*⟩ to the return-node of call site *13*, which in turn propagates it to program point *14*, the exit-node of the try statement. The exit-node of the try statement propagates ⟨*excp_t2,reachable*⟩ to the entry node of the catch at program point *15* because an exception of type *excp_t2* is caught by the catch statement. The entry-node of the catch statement transforms ⟨*excp_t2,reachable*⟩ to ⟨*reachable*⟩ (as the exception has been caught) and propagates ⟨*reachable*⟩ to program point *16*, which in turn propagates ⟨*reachable*⟩ to program point *17*.

Next consider another example given in Figure 12 which illustrates the purpose of *label* stored in a data-flow element. Suppose, for the sake of illustration, ⟨*reachable*⟩ is propagated to the entry-node of *method1*. From the entry-node of *method1*, ⟨*reachable*⟩ reaches statement *2*, via program points *l1* and *1*. Statement *2* transforms ⟨*reachable*⟩ to ⟨*l1,reachable*⟩ and propagates it to program point *3*, the exit-node of the try statement. The presence of ⟨*l1,reachable*⟩ at program point *3* means that program point *3* is reachable from the entry-node of *main* along paths which have an incomplete, pending transition to program point *l1*. The reason why only one label in a data-flow element is enough is very similar to the reason why only one exception type is enough. From the exit-node of the try statement ⟨*l1,reachable*⟩ is propagated to program point *4* and then to program point *5*, the exit-node of the finally statement. At an exit-node of a finally statement, the label (or exception type) stored in a data-flow element is used to decide the future propagation of the data-flow element. Thus, at program point *5*, ⟨*l1,reachable*⟩ is transformed to ⟨*reachable*⟩ and ⟨*reachable*⟩ is propagated to program point *l1*, signifying that the pending transition has been completed. Again note that statement *6* is unreachable and no reachability information is propagated to it and $object_6$ is not reported as a value of *g* at any program point.

Now consider the data-flow elements of kinds 4, 5 and 6. These are used for propagating values of pointer locations. A data-flow element of the fourth kind is a conditional-points-to and its meaning is same as explained in the case of the *basic* algorithm. A data-flow element of fifth or sixth kind is called an exceptional-conditional-points-to and it is used for propagation in the presence of exceptions and finally statements. An *excp-type* (exception type) in an exceptional-conditional-points-to summarizes an uncaught exception along any path through which the data-flow element reaches a program point, as explained above in the case of reachability information. Similarly, a *label* in an exceptional-conditional-points-to summarizes a pending, incomplete transition along any path through which the data-flow element reaches a program point, as explained above in the case of reachability information.

Consider the example in Figure 13. A reference to $object_{10}$ is passed as the value of *param* at call site *11*. As a result, the conditional-points-to ⟨⟨*param*,$object_{10}$⟩,⟨*param*,$object_{10}$⟩⟩ is propagated to the entry-node of *method1*. From the entry-node of *method1*, ⟨⟨*param*,$object_{10}$⟩,⟨*param*,$object_{10}$⟩⟩ is propagated to statement *2* via program points *1* and *1a*. As

```
class A { ... };
class excp_t1 extends Exception {};
class excp_t2 extends Exception {};
class example1 {
  public static A g;
  public static void method1() {
    Exception p1;
    1:  try {
      2:  p1 = new excp_t1();
      3:  throw p1;
      4:
    }
    catch(excp_t1 a1) {
      5:
    }
    catch(Exception a2) {
      6:  g = new A();
    }
    7:
  }
  public static void method2() throws excp_t2 {
    excp_t2 p2;
    8:  p2 = new excp_t2();
    9:  throw p2;
    10:
  }
  public static void main() {
    11:  try {
      12:  method1();
      13:  method2();
      14:
    }
    15:  catch(excp_t2 a3) {
      16:
    }
    17:
  }
} /* end of class example1 */
```

Figure 11: Propagation of reachability information in the presence of exceptions

```
class A { ... };
class example2 {
  public static A g;
  public static void method1() {
    l1:  while ( ... )  {
      1:  try {
        2:  continue l1;
        3:
      }
      4:  finally {
        5:
      }
      6:  g = new A();
    }
  }
}
```

Figure 12: Incomplete, pending transitions in the presence of finally statements

a result, statement *2* generates $\langle\langle param,object_{10}\rangle,\langle global,object_{10}\rangle\rangle$ which is propagated to statement *4* via statement *3*. Statement *4* throws $object_3$, whose type is *excp_t1*. As a result, statement *4* transforms $\langle\langle param,object_{10}\rangle,\langle global,object_{10}\rangle\rangle$ to the exceptional-conditional-points-to $\langle excp\_t1, \langle param,object_{10}\rangle,\langle global,object_{10}\rangle\rangle$ and propagates the exceptional-conditional-points-to to program point *7a*, the exit-node of the try statement. Since the catch at program point *8* cannot catch an exception of type *excp_t1*, the exit-node of the try statement propagates $\langle excp\_t1,\langle param,object_{10}\rangle,\langle global,object_{10}\rangle\rangle$ to the exit-node of *method1*. The exit-node of *method1* substitutes the assumed-points-to $\langle param,object_{10}\rangle$ by *empty* (as $\langle empty,\langle local2,object_{10}\rangle\rangle$ holds at the call site and local2 is the actual for *param* at the call site) and propagates the resulting exceptional-conditional-points-to $\langle excp\_t1,empty,\langle global,object_{10}\rangle\rangle$ to the return-node of call site *11*, which in turn propagates the exceptional-conditional-points-to to program point *12a*, the exit-node of the try statement. Since an exception of type *excp_t1* is caught by the catch at program point *13*, program point *12a* propagates the exceptional-conditional-points-to to the entry-node of the catch at program point *13*. The entry-node of the catch statement transforms the exceptional-conditional-points-to to the conditional-points-to $\langle empty,\langle global,object_{10}\rangle\rangle$ (as the exception has been caught by the catch statement) and propagates the conditional-points-to to program point *14*. Using this conditional-points-to, statement *14* generates $\langle empty,\langle local4,object_{10}\rangle\rangle$ and propagates it to the exit-node of *method2* via the exit-node of the catch statement.

```
class A { ...  };
class excp_t1 extends Exception {};
class excp_t2 extends Exception {};
class example3 {
  public static A global;
  public void method1( A param )
      throws Exception {
    Exception p;
    A local1;
    1:  try {
      1a if ( ...  )  {
        2:  global = param;
        3:  p = new excp_t1();
        4:  throw p;
      }
      else {
        5:  global = new A();
        6:  p = new excp_t2();
        7:  throw p;
      }
      7a:
    }
    8:  catch(excp_t2 e1) {
      9:  local1 = global;
    }
  }
  public void method2() throws Exception {
    A local2, local3, local4;
    try {
      10:  local2 = new A();
      11:  method1(local2);
      12:  local3 = global;
      12a:
    }
    13:  catch (excp_t1 e2) {
      14:  local4 = global;
    }
  }
} /* end of example3 */
```

Figure 13: Exceptional-conditional-points-to's

Next, consider statement *5*. It generates $\langle empty,\langle global,object_5\rangle\rangle$ and propagates it to statement *7* via statement *6*. Statement *7* throws $object_6$, whose type is *excp_t2*. As a result, statement *7* transforms $\langle empty,\langle global,object_5\rangle\rangle$ to the exceptional-conditional-points-to $\langle excp\_t2,empty,\langle global,object_5\rangle\rangle$ and propagates the exceptional-conditional-points-to to program point *7a*, the exit-node of the try statement. Since an exception of type *excp_t2* is caught by the catch at program point *8*, program point *7a* propagates the exceptional-conditional-points-to to the entry-node of the catch at program point *8*. The entry-node of the catch statement transforms the exceptional-conditional-points-to to the conditional-points-to $\langle empty,\langle global,object_5\rangle\rangle$ (as the exception has been caught by the catch statement) and propagates the conditional-points-to to the exit-node of *method1* via program point *9* and the exit-node of the catch statement. The exit-node of *method2* propagates $\langle empty,\langle global,object_5\rangle\rangle$ to the return-node of call site *11*, which propagates the conditional-points-to to program point *12a* via statement *12*. Since the conditional-points-to has no exception type or label stored in it, it means that the conditional-points-to has reached program point *12a* along a path without any uncaught exception or incomplete transition. Thus, program point *12a* propagates the conditional-points-to to the exit-node of *method2*. Using this conditional-points-to, statement *12* generates the conditional-points-to $\langle empty,\langle local3,object_5\rangle\rangle$, which is also propagated to the exit-node of *method2* via program point *12a*.

Now consider another example given in Figure 14 which illustrates propagation of exceptional-conditional-points-to's of the form $\langle label,z,u\rangle$. Statement *1* generates $\langle empty,\langle local1,object_1\rangle\rangle$ and propagates it to statement *2*. Statement *2* transforms the conditional-points-to into $\langle label1,empty,\langle local1,object_1\rangle\rangle$ and propagates the exceptional-conditional-points-to to program point *6*, the exit-node of the try statement, signifying that the exceptional-conditional-points-to has reached program point *6* along a path with an incomplete, pending transition to program point *label1*. This is needed because (according to semantics of Java) the finally statement must be executed before making the transition to program point *label1*. Unless the finally statement creates its own reason to exit itself (as we shall see later), the *label* (or a an exception type) in an exceptional-conditional-points-to is used at the exit-node of the finally statement to decide propagation of the exceptional-conditional-points-to. The exit-node of the try statement propagates $\langle label1,empty,\langle local1,object_1\rangle\rangle$ to the entry-node of the finally statement, which in turn propagates the exceptional-conditional-points-to to statement *9*. Using the exceptional-conditional-points-to, statement *9* generates $\langle label1,empty,\langle local2,object_1\rangle\rangle$ and propagates both the exceptional-conditional-points-to's to the exit node of the finally statement. The exit-node of the finally statement transforms the two exceptional-

```
class A { ... };
class excp_t1 extends Exception {};
class excp_t2 extends Exception {};

class example4 {
  public void method1() throws Exception {
    A local1, local2, local3;
    Exception p;

    local1 = null;

    label1: while ( ... ) {
      try {
        if ( ... ) {
          1: local1 = new A();
          2: continue label1;
        }
        else {
          3: p = new excp_t2();
          4: local1 = new A();
          5: throw p;
        }
        6:
      }
      7: catch (excp_t1 e1) {
      }
      8: finally {
        9: local2 = local1;
      }
    }

    10: local3 = local1;
  }
}
```

Figure 14: Propagation in the presence of pending, incomplete transitions

```
class excp_t1 extends Exception {};
class example5 {
  public static void method1() throws Exception {
    Exception p, local;
    try {
      1: p = new excp_t1();
      2: throw p;
    }
    catch (excp_t1 e1) {
      3: local = e1;
    }
  }
}
```

Figure 15: Propagation of exception objects

conditional-points-to's into the conditional-points-to's $\langle \mathit{empty}, \langle \mathit{local1}, \mathit{object}_1 \rangle\rangle$ and $\langle \mathit{empty}, \langle \mathit{local2}, \mathit{object}_1 \rangle\rangle$ and propagates the conditional-points-to's to program point *label1* completing the pending transition. Program point *label1* propagates the two conditional-points-to's to statement *10*, which uses the first conditional-points-to to generate $\langle \mathit{empty}, \langle \mathit{local3}, \mathit{object}_1 \rangle\rangle$. In contrast, program point *5* propagates $\langle \mathit{excp\_t2}, \mathit{empty}, \langle \mathit{local1}, \mathit{object}_4 \rangle\rangle$ to the exit-node of the the try statement, which in turn propagates the exceptional-conditional-points-to (as the catch statement does not catch exceptions of type *excp_t2*) to the entry-node of the finally statement. The entry-node of the finally statement propagates the exceptional-conditional-points-to to the exit-node of the finally statement via statement *9*, which in turn propagates the exceptional-conditional-points-to to the exit-node of *method1*.

Now consider the example in Figure 15 which illustrates the propagation of a data-flow element of the form $\langle \mathbf{excp}, z, \mathit{obj} \rangle$. Here **excp** is a keyword, z is an assumed-points-to and *obj* is an exception object. A data-flow element of this form is used for propagating an uncaught exception object and it is needed because the exception object is assigned to the parameter of the catch statement which catches the exception. Statement *2* propagates $\langle \mathbf{excp}, \mathit{empty}, \mathit{object}_1 \rangle$ to the exit-node of the try statement, representing the exception thrown at statement *2*. The exception is caught by the catch statement as the type of object_1 is *excp_t1*. Thus, the exit-node of the try statement propagates $\langle \mathbf{excp}, \mathit{empty}, \mathit{object}_1 \rangle$ to the entry-node of the catch statement. Since a reference to the caught exception object is assigned to *e1* (the parameter of the catch statement), the entry-node of the catch statement generates $\langle \mathit{empty}, \langle \mathit{e1}, \mathit{object}_1 \rangle\rangle$ from $\langle \mathbf{excp}, \mathit{empty}, \mathit{object}_1 \rangle$ and propagates $\langle \mathit{empty}, \langle \mathit{e1}, \mathit{object}_1 \rangle\rangle$ to program point *3*. The entry-node of the catch statement does not propagate $\langle \mathbf{excp}, \mathit{empty}, \mathit{object}_1 \rangle$ any further as the exception has been caught. Statement *3* generates $\langle \mathit{empty}, \langle \mathit{local}, \mathit{object}_1 \rangle\rangle$ from $\langle \mathit{empty}, \langle \mathit{e1}, \mathit{object}_1 \rangle\rangle$ and propagates both the conditional-points-to's to the exit-node of the catch statement. The exit-node of the catch statement propagates $\langle$*empty*,$\langle$*local*,$object_1\rangle\rangle$ to the exit-node of *method1*. But it does not propagate $\langle$*empty*,$\langle$*e1*,$object_1\rangle\rangle$ any further as *e1* is not visible outside the catch statement.

6.1.2 Summary of Propagation.

In the next few sections we summarize what has been explained through examples above. First we describe how a *throw* statement is handled. Next, we describe propagation at a method *exit-node*. Finally, we describe how a *finally* statement is handled.

6.1.2.1 *throw* **Statement.** In addition to the conditional-points-to's described previously, this algorithm uses another kind of data-flow element, called **exceptional-conditional-points-to**'s, which capture propagation due to exceptions. The conditional part of these points-to's consists of an exception type (or a label) and an assumed-points-to (as before). Consider a *throw* statement *l* in a method *Proc*, which throws an object of type T (run-time type and not the declared type). Moreover let $\langle\langle q,\ obj1\ \rangle,\ \langle p,\ obj2\ \rangle\rangle$ be a conditional-points-to reaching the top of *l*. At the *throw* statement, this points-to is transformed to $\langle T, \langle q, obj1\rangle, \langle p, obj2\rangle\rangle$ and propagated to the exit-node of the corresponding *try* statement, if there is one. The *catch-table* (defined in Section 5.3) at this node is checked to see if this exception (identified by its type T) can be caught by any of the corresponding *catch* statements. If yes, this exceptional-conditional-points-to is forwarded to the entry-node of this *catch* statement, where it is changed back into an ordinary conditional-points-to $\langle\langle q, obj1\rangle\langle p, obj2\rangle\rangle$. If not, this exceptional-conditional-points-to is forwarded to the entry-node of a *finally* statement (if any), or the exit-node of the innermost enclosing *try*, *catch*, *finally* or the method body.

A *throw* statement also generates a data-flow element for the exception. Suppose the thrown object is *obj* and it is the thrown object under the assumed-points-to $\langle$*p*, *obj1* $\rangle$. Then $\langle$**excp**,*p*,*obj1*$\rangle$ representing the exception is generated. Such data-flow elements are handled like exceptional-conditional-points-to's, described above. If such a data-flow element reaches the entry of a *catch* statement, it is used to instantiate the parameter of the *catch* statement.

If the *throw* is not directly contained in a *try* statement, then the data-flow elements generated by it are propagated to the exit-node of the innermost enclosing *catch*, *finally* or method body.

6.1.2.2 *exit-node* **of a method.** At the exit-node of a method, a data-flow element of type 4,5 or 7 is forwarded (after replacing the assumed-points-to as described in Section 3.3) to the return-node of a call site of this method if and only if the assumed-points-to of the data-flow element holds at the call site. At a return-node, ordinary conditional-points-to's (type 4) are handled as before. However, a data-flow element of type 5 or 7 is handled as if it were generated by a *throw* at this return-node. Data-flow elements representing values of local variables are propagated across a call if and only if $\langle$*reachable* $\rangle$ or $\langle$*excp-type,reachable* $\rangle$ reaches the exit-node of one of the called methods. $\langle$*excp-type,reachable* $\rangle$ at the exit-node of a method causes the conditional-points-to's representing values of local variables at a call site of this method to change to exceptional-conditional-points-to's (with *excp-type*) at the return-node of the call site. $\langle$*reachable* $\rangle$ is propagated across a call (not contained in a *finally*) if and only if $\langle$*reachable* $\rangle$ reaches the exit-node of one of the called methods.

6.1.2.3 *finally* **statement.** As described in Section 5, the *finally* statement in Java has involved semantics. Although it does not change the complexity significantly (problems in **P** remain in **P**), to handle *finally* correctly the algorithm needs to take care of some details. For all data-flow elements entering a *finally*, the algorithm remembers the reason for entering it. For data-flow elements of type 2, 5 or 7, the associated exception already represents this reason. A *label* is associated with data-flow elements of type 1 or 4 (to generate data-flow elements of type 3 or 6), which represents the statement number to which control should go after exit from the *finally*. So the data-flow elements in a *finally* have one of the following forms:

1. $\langle$*excp-type, reachable* $\rangle$,
2. $\langle$*label, reachable* $\rangle$,
3. $\langle$*excp-type, z, u* $\rangle$,
4. $\langle$*label, z, u* $\rangle$,
5. $\langle$**excp**, *z, obj* $\rangle$.

When a labelled data-flow element reaches the labelled statement, the label is dropped and it is transformed into the corresponding unlabelled data-flow element.

Inside a *finally*, due to labels and exception types associated with data-flow elements, *apply* uses a different criterion for combining data-flow elements (at an assignment node) than the one given in Section 3.2. Two data-flow elements $\langle$*x1,y1,z1* $\rangle$ and $\langle$*x2,y2,z2* $\rangle$ can be combined if and only if both *x1* and *x2* represent the same exception type or the same label, and *y1* and *y2* are compatible (as defined in Section 3.2).

At a call statement (inside a *finally*), if a data-flow element has a *label* or an exception type associated with it, it is treated as part of the context (assumed-points-to) and not forwarded to the target node. It is put back when assumed-points-to's are expanded at an exit-node of a method. This is illustrated later in the second example given in Section 6.2. For exceptional-conditional-points-to's or data-flow elements representing exceptions, the exceptions associated with them at the exit-node **override** any *label* or exception type associated with their assumed-points-to's at a corresponding call site. Data-flow elements of the form $\langle$*label,reachable*$\rangle$ or $\langle$*excp-type,reachable*$\rangle$ are propagated across a call if and only if $\langle$*reachable*$\rangle$ reaches the exit-node of one of the called methods.

A *try* statement inside a *finally* is handled like a call because it can cause labels and exceptions to stack up. This is also illustrated later in the second example given in Section 6.2. If $\langle x, z, u\rangle$, where x is a label or an exception type, reaches the entry node of such a *try* statement, it is changed to $\langle u,u\rangle$ (just like what is done at a call site) and x,z is remembered as the the condition associated with u in a table at the entry-node of *try*. When this *try-catch-finally* is exited, if u is the assumed-points-to for a conditional-points-to, it is replaced by x,z. For exceptional-conditional-points-to's , u is replaced by z and the exception type overrides x. For labelled-conditional-points-tos, if the label represents a statement inside this *finally* statement, u is replaced by x,z and forwarded to the labelled statement. If the label represents a statement outside this *finally* statement, it overrides x and u is replaced by only z. A data-flow element of the form $\langle$*excp,z,obj*$\rangle$ reaching the entry of a *try* nested inside a *finally* is propagated (across the nested construct) only if $\langle$*reachable*$\rangle$ or $\langle$*label, reachable*$\rangle$, where *label* represents a statement inside this *finally*, reaches the exit of the nested *try-catch-finally* construct. Data-flow elements of the form $\langle$*label,reachable*$\rangle$ or $\langle$*excp-type,reachable*$\rangle$ are treated similarly. If a conditional-points-to with an empty condition reaches the exit of a nested *try-catch-finally*, it produces a labelled-conditional-points-to for each *label* for which $\langle$*label,reachable*$\rangle$ reaches the entry of the nested *try-catch-finally*, and it produces a exceptional-conditional-points-to for each *excp-type* for which $\langle$*excp-type,reachable*$\rangle$ reaches the entry of the nested *try-catch-finally*.

If the *finally* generates a reason of its own for exiting itself, the previous *label/exception-type* associated with a data-flow element is discarded, and the new *label/exception-type* representing this reason for leaving *finally* is associated with the data-flow element.

6.2 Examples for the *extended* algorithm

The following two examples illustrate the *extended* algorithm. The first example is simpler and does not contain any *try* statements nested inside a *finally* statement. The second one is more involved and illustrates data-flow in the presence of call statement or a *try* statement nested inside a *finally* statement.

Example1:

```
// Note: for simplicity only a part of the solution is shown
class A {}; class excp_t extends Exception {};
class base {
  public static A a;
  public static void method( A param ) throws excp_t {
    excp_t unexp;
    a = param;
    l1:  unexp = new excp_t();
    // ⟨empty,⟨unexp,object_l1⟩⟩,
    // ⟨⟨param,object_l2⟩,⟨a,object_l2⟩⟩
    throw unexp;
    // ⟨excp,empty,object_l1⟩,
    // ⟨excp_t,⟨param,object_l2⟩,⟨a,object_l2⟩⟩
  }
};
class test {
  public static void test_method( ) {
    A local;
    l2:  local = new A();
    try {
      base.method( local );
      // ⟨excp,empty,object_l1⟩, ⟨excp_t,empty,⟨a,object_l2⟩⟩
    }
    catch( excp_t par ) {
      // ⟨empty,⟨par,object_l1⟩⟩, ⟨empty,⟨a,object_l2⟩⟩
      l3:
    }
    finally {
      // ⟨l4, empty, ⟨a,object_l2⟩ ⟩
    }
    // ⟨empty,⟨a,object_l2⟩⟩
    l4:
  }
  public static void main( ) {
    test_method();
  }
};
```

Example2:

```
// Note: for simplicity only a part of the solution is shown
class A {};
class excp_t1 extends Exception {};
class excp_t2 extends Exception {};
class base {
  public static A a;
  public static void method1( A param ) throws Exception {
    Exception unexp;
    a = param;
    l1:  unexp = new excp_t1();
    // ⟨empty,⟨unexp,object_l1⟩⟩,
    // ⟨⟨param,object_l2⟩,⟨a,object_l2⟩⟩
    throw unexp;
    // ⟨excp,empty,object_l1⟩,
    // ⟨excp_t1,⟨param,object_l2⟩,⟨a,object_l2⟩⟩
  }
  public static void method2() {
      // ⟨⟨a,object_l2⟩,⟨a,object_l2⟩⟩, ⟨reachable ⟩.
  }
};
class test {
  public static void test_method( ) throws Exception {
    A local;
    l2:  local = new A();
```

```
    try {
      base.method1( local );

      // ⟨excp,empty,object_l1⟩, ⟨excp_t1,empty,⟨a,object_l2⟩⟩
    }
    catch( excp_t2 param1 ) {
    }
    finally {
      // ⟨excp,empty,object_l1⟩, ⟨excp_t1,empty,⟨a,object_l2⟩⟩,
      // ⟨excp_t1,reachable⟩.

      // Note that ⟨excp_t1,empty,⟨a,object_l2⟩⟩ is propagated as ⟨⟨a,object_l2⟩,⟨a,object_l2⟩⟩
      // to the entry-node of base.method2 and ⟨excpt_t1,empty⟩ is remembered at this
      // call site as the context for the assumed-points-to ⟨a,object_l2⟩.
      base.method2();

      // ⟨excp,empty,object_l1⟩, ⟨excp_t1,empty,⟨a,object_l2⟩⟩,
      // ⟨excp_t1,reachable⟩.
      try {
        // ⟨⟨a,object_l2⟩,⟨a,object_l2⟩⟩, ⟨reachable ⟩.

        // Note that ⟨excp_t1,empty,⟨a,object_l2⟩⟩ is propagated
        // as ⟨⟨a,object_l2⟩,⟨a,object_l2⟩⟩ and ⟨excpt_t1,empty⟩ is
        // remembered as the context for the assumed-points-to
        // ⟨a,object_l2⟩ at the entry-node of the try statement.

        excp_t1 unexp;

        l3:  unexp = new excp_t1();

        throw unexp;
        // ⟨excp,empty,object_l3⟩, ⟨excp_t1,⟨a,object_l2⟩,⟨a,object_l2⟩⟩,
        // ⟨excp_t1,reachable⟩.
      }
      catch( excp_t1 param2 ) {
        // ⟨empty,⟨param2,object_l3⟩⟩, ⟨⟨a,object_l2⟩,⟨a,object_l2⟩⟩,
        // ⟨reachable ⟩.
      }
      finally {
        // ⟨l4, ⟨a,object_l2⟩, ⟨a,object_l2⟩ ⟩, ⟨l4,reachable⟩.
      }

      // ⟨excp_t1,empty,⟨a,object_l2⟩⟩,
      // ⟨excp,empty,object_l1⟩, ⟨excp_t1,reachable⟩.

      // ⟨l4, ⟨a,object_l2⟩, ⟨a,object_l2⟩ ⟩ at the exit-node
      // of the nested finally changes to ⟨excp_t1,empty,⟨a,object_l2⟩⟩
      // because ⟨excp_t1,empty⟩ is associated with the
      // assumed-points-to ⟨a,object_l2⟩ at the entry-node of
      // the nested try. Similarly, ⟨excp,empty,object_l1⟩ and
      // ⟨excp_t1,reachable⟩ are propagated because ⟨l4,reachable⟩
      // reaches the exit-node of the nested finally, l4 is within the
      // finally containing the nested try, and ⟨excp,empty,object_l1⟩
      // and ⟨excp_t1,reachable⟩ hold at the entry-node of the nested try.
      l4:
    } // End of outer finally

    // ⟨excp_t1,empty,⟨a,object_l2⟩⟩,
    // ⟨excp,empty,object_l1⟩, ⟨excp_t1,reachable⟩.
    l5:

  }
  public static void main( ) throws Exception {
    test_method();
  }
};
```

6.3 Proof of precision

In Appendix C we present an outline of a proof for the following theorem:

Theorem 5 *The algorithm described in section 6.1 computes the precise solution for programs with only single-level types and exceptions without subtyping, and without dynamically-dispatched calls.*

6.4 Complexity

In this section, we show that the worst-case complexity of the extended algorithm for programs with only single-level types and exceptions without subtyping and without dynamically-dispatched calls is $O(n^7)$. If we disallow *try*s nested inside a *finally*, the worst-case complexity is $O(n^6)$. We will need the following lemma:

Lemma 6 *Suppose*
(1) $\langle\langle$*var1,obj1*$\rangle$, $\langle$*var2,obj2*$\rangle\rangle$,
(2) $\langle$*label,* $\langle$*var1,obj1*$\rangle$, $\langle$*var2,obj2*$\rangle\rangle$,
(3) $\langle$*excp-type,* $\langle$*var1,obj1*$\rangle$, $\langle$*var2,obj2*$\rangle\rangle$ *or*
(4) $\langle excp, \langle var1, obj1\rangle, obj2\rangle$
is computed by the algorithm at a program point l. Then obj1 and obj2 are the same.

proof is by induction on the number of iterations needed to compute the data-flow element. □

Let the total number of exception types be XT, the total number of exception objects be XO[13], the total number of labels used by the algorithm be B, the total number of *try*s directly nested inside a *finally* be NT, and n, V, A, L and C be as defined in section 4.2.

Now consider a program point l. There could be at most V user defined variables of reference type visible at l. Each of them may point to L different objects. Each such points-to may be implied by at most V assumed-points-to's, this follows from lemma 6. This implies that if l is not contained in a *finally*, the solution set at l may contain up to $O(V^2L)$ conditional-points-to's, $O(XT\star V^2L)$ exceptional-conditional-points-to's, $O(V\star XO)$ points-to's for exception objects and $O(XT+1)$ data-flow elements for checking reachability. On the other hand, if l is contained in a *finally*, the solution set at l may contain up to $O(B\star V^2L)$ labelled-conditional-points-to's, $O(XT\star V^2L)$ exceptional-conditional-points-to's, $O(V\star XO)$ points-to's for exception objects and $O(XT+B)$ data-flow elements for checking reachability.

Suppose l is not an exit-node of a method or an exit-node of a *try-catch-finally* nested inside a *finally* or a call-node. For each data-flow element reaching l, *apply* takes constant time and a constant amount of work is done along an edge to a successor of l. Since there are $O(n)$ nodes and $O(n)$ intraprocedural edges, the total amount of work done for such nodes is $O(n($ maximum number of conditional-points-to's + maximum number of labelled-conditional-points-to's + maximum number of exceptional-conditional-points-to's + maximum number of data-flow elements for exception objects + maximum number of data-flow elements used for checking reachability)) i.e.

$$O(n((B+XT+1)V^2L+V\star XO+XT+B+1)).$$

The work done at call-nodes in passing data-flow elements to the entry-nodes of called methods is $O(C(A+1)(B+XT+1)V^2L)$ because each points-to at a call-node can imply at most $A+1$ points-to's at the entry-node of the called method. In addition $O(C(B+XT+1)V^2L)$ work may be done in passing points-to's representing values of local variables from call-nodes to their respective return-nodes (note that part of this work may be done at an exit-node).

Now consider an exit-node *ex-node*. Let *r-node* be one of the return-nodes corresponding to *ex-node*. If *r-node* is not contained in a *finally*, for each conditional-points-to or exceptional-conditional-points-to or data-flow element representing an exception object passed from *ex-node* to *r-node*, the assumed-points-to may expand into $O(V)$ assumed-points-to's at *r-node*, again we use lemma 6. If *r-node* is contained in a *finally*, for each conditional-points-to passed from *ex-node* to *r-node*, the assumed-points-to may expand into $O(V(XT+B+1))$ assumed-points-to's at *r-node*. Since exceptions generated by the call override previous exceptions and labels, the assumed-points-to of an exceptional-conditional-points-to or data-flow element representing an exception object may expand into at most V assumed-points-to's. Similarly, for an edge leaving a *try-catch-finally* nested inside a *finally* up to $O(V(XT+B))$ work may be done for each data-flow element. So the total amount of work done along an interprocedural edge from an exit-node to a return-node is $O(V(V^2L+XT\star V^2L+V\star XO)+V\star(XT+B)V^2L)$, while the total amount of work done along an edge leaving a nested *try-catch-finally* is $O(V(XT\star V^2L+V\star XO)+V(XT+B)(BV^2L+V^2L))$.

There are C interprocedural edges between exit-nodes and return-nodes, and NT edges leaving nested *try-catch-finally*. This means that the worst-case complexity is:

$$O((C(V+A)+n)((XT+B+1)\star V^2L+$$
$$V\star XO)+XTn^2+S)$$

Here, $O(XT\star n^2)$ is the cost of building the *catch* tables and

$$S=NT\star V((XT+(XT+B)B)\star V^2L+V\star XO).$$

Since each of C, V, A, L, XO, B (at most the maximum size of a method), NT and XT is at most $O(n)$, the the

[13] identified by their creation sites.

worst-case complexity is $O(n^7)$. Note that if we disallow *try*s nested inside a *finally*, S is 0 and hence the worst-case complexity is $O(n^6)$.

7 Safety of the extended algorithm in the general case

In Appendix D we show that the algorithm presented in 6.1 computes a safe solution for programs written in *JavaWoThreads*. Specifically, we present an outline of a proof for the following theorem:

Theorem 6 *The extended algorithm computes a safe solution for programs written in a subset of Java defined in section 2.10.*

8 Complexity of the extended algorithm in the general case

In this section, we show that the extended algorithm is polynomial-time in the general case. Let n, A, XO, XT, B, NT and L be as defined in section 6.4, the total number of entities which can be the first member of a points-to pair be V (we assume this to be at least 1), the total number of interprocedural edges (recall that due to dynamic dispatch, these are computed during analysis) be I, the maximum number of possible points-to's be D (VL) and the maximum number of field selectors appearing in an operand be k (e.g., number of field selectors in p.f1.f2 is 2).

There could be up to D points-to's valid at l. Each such points-to may be implied by up to D points-to's at the entry of the procedure containing l, this is due to general types and dynamically-dispatched calls as shown in section 3.4. This implies that the solution set at l may contain up to $O(D^2)$ conditional-points-to's. Similarly there could be up to $O(XT \star D^2)$ exceptional-conditional-points-to's and up to $O(D \star XO)$ data-flow facts representing exception objects. If l is contained in a *finally* statement, there could be up to BD^2 labelled conditional-points-to's. Suppose l is an assignment node. For each conditional-points-to/exceptional-conditional-points-to/labelled-conditional-points-to reaching l, up to $O((XT + B + 1)D^2)$ (due to multiple field selectors) conditional-points-to's/exceptional-conditional-points-to's/labelled-conditional-points-to's may be generated at l and up to $O(k^2 D^{2k} L^2)$ work may be done by *apply*. In contrast, for single-level types, only a single conditional-points-to may be generated, and a constant amount of work is done. But in this case, due to multiple field selectors, each operand (lhs and rhs) may generate up to $O(D^k L)$ data-flow elements, as each field selector increases the number of assumed-points-to's by 1. Finally one assumed-points-to is selected from $2k$ (at most) assumed-points-to's associated with a data-flow element if they are compatible. Note that $O(k^2)$ is the worst-case cost of checking pairwise compatibility of $O(k)$ assumed-points-to's as the compatibility relation is not transitive. Otherwise this data-flow element is discarded. Similarly, we can show that the amount of work done at any node which is neither a method exit-node nor a call-node is $O((XT+B+1)k^2 D^{2(k+1)} L^2)$. Note that any work done at a node along an edge leaving the node will be charged to the edge and not to the node.

Suppose l is not an exit-node of a method or an exit-node of a *try-catch-finally* nested inside a *finally* or a call-node. The amount of work done along an edge to a successor of l is at most $O((XT + B + 1)D^2)$.

The work done at call-nodes in passing data-flow elements to the entry-nodes of called methods is $O(I(A+1)(B + XT + 1)V^2 L)$ because each points-to at a call-node can imply at most $A + 1$ points-to's at the entry-node of the called method. In addition $O(I(B + XT + 1)V^2 L)$ work may be done in passing points-to's representing values of local variables from call-nodes to their respective return-nodes (note that part of this work may be done at an exit-node).

Now consider an exit-node *ex-node*. Let *r-node* be one of the return-nodes corresponding to *ex-node*. If *r-node* is not contained in a *finally*, for each conditional-points-to or exceptional-conditional-points-to or data-flow element representing an exception object passed from *ex-node* to *r-node*, the assumed-points-to may expand into $O(D)$ assumed-points-to's at *r-node*, again this is due general types and dynamically-dispatched calls. If *r-node* is contained in a *finally*, for each conditional-points-to passed from *ex-node* to *r-node*, the assumed-points-to may expand into $O(D(XT + B))$ assumed-points-to's at *r-node*. Since exceptions generated by the call override previous exceptions and labels, the assumed-points-to of an exceptional-conditional-points-to or data-flow element representing an exception object may expand into at most D assumed-points-to's. Similarly, for an edge leaving a *try-catch-finally* nested inside a *finally* up to $O(D(XT + B))$ work may be done for each data-flow element. So the total amount of work done along an interprocedural edge from an exit-node of a method to a return-node is $O(D(D^2+XT\star D^2+D\star XO)+D\star(XT+B)D^2)$, while the total amount of work done along an edge leaving a nested *try-catch-finally* is $O(D(XT \star D^2 + D \star XO) + D(XT + B)(BD^2 + D^2))$. There are I interprocedural edges and NT edges leaving nested *try-catch-finally*. Hence, we have the following lemma and theorem.

Lemma 7 *The worst-case complexity of the extended algorithm in the general case is* $O((I(D+V)+n)((XT+B+1)D^2+D\star XO)+XTn^2+n(XT+B+1)k^2 D^{2(k+1)} L^2+S)$.

Here $S = NT \star D((XT + (XT + B)B) \star D^2 + D \star XO)$, $O(I \star D((XT + B + 1)D^2 + D \star XO))$ is the worst-case cost of propagation along an interprocedural edge, $O(k^2 D^{2k} L^2)$ is the worst-case cost of *apply* and $O(S)$ is the worst-case cost of propagation along an edge leaving a *try-catch-finally* directly nested inside a *finally.*

Theorem 7 *The extended algorithm is polynomial-time for program written in JavaWoThreads.*

proof: Since V (at most O(n^2)), D (at most VL), I (at most O(n^2)), XT (at most n) , L (at most n), B (at most n), NT (at most n), A (at most n) and XO (at most n) are polynomial in n, and k can be considered constant, the theorem follows from lemma 7.

Note that we can always choose k to be a fixed constant, say c, (even 1) because using temporary variables, any statement with more field selectors can be broken into a set of statements each having at most c field selectors. This will increase the size of the program by at most a factor of k, and might degrade the precision a little without compromising safety.

Remark. The worst-case complexity of the *basic* algorithm can be obtained from lemma 7 by replacing B, XT, XO and NT by 0. This shows that the overhead due to exceptions is small compared to the complexity of the *basic* algorithm. This is important because this technique for handling exceptions may be combined with other flow-sensitive algorithms.

Complexity of Computing the Precise Solution in the General Case. Using Landi's result [Lan92b] about undecidability of may alias analysis, it can be shown that *PTA* in the general case is undecidable.

Optimizations. The estimate in lemma 7 is a worst-case bound; in practice we expect the performance of the extended algorithm to be much better. Moreover, various heuristics can be used for improving its precision and efficiency. For example, in [CR97], we discuss some techniques for improving the precision and efficiency of the *basic* algorithm. One obvious way to improve precision is to associate more context with data-flow elements. In [CR97], we show that remembering the receiver object along with an assumed-points-to can improve precision in many situations. Similarly, an assumed-points-to may be associated with *excp-type* in exceptional-conditional-points-to's. Precision can also be improved by using finite approximations of the call stack (along with creation sites) for naming dynamically created objects, instead of just creation sites.

9 C++ Related Issues

In this section we briefly discuss the changes needed in the *extended algorithm* to handle some C++specific issues: (1) pointers to stack locations, (2) arbitrary casting and (3) implicit calls to destructors.

Consider the C++program fragment in Figure 16 which illustrates pointers to out-of-scope stack locations.

At call site *2*, a pointer to *local* is passed argument to *proc1*. Since storage for *local* is allocated on stack (as

```
void proc1(int **param) {
  1:  *param = new int;
}
void proc2(void) {
  int *local;
  2:  proc1(&local);
}
```

Figure 16: Treatment of out-of-scope locations

```
void proc1( void ) {
  A local;
  excp_t1 *p;
  1:  p = new excp_t1();
  2:  throw p;
  3:
}
```

Figure 17: Exceptions and implicit destructor calls

opposed to heap), a pointer to stack is created at program point *2*. Further, although *local* is not directly visible at program point *1*, *local* is updated at program point *1* as **param* is an alias for *local*. Such situations can be easily handled using the techniques of [LR92] and [EGH94] for dealing with *nonvisibles*. Intuitively, the idea is to represent all the out-of-scope locations to which *param* can point to at the entry of *proc1* using a special *nonvisible* location.

Arbitrary casting (and union types) can be easily handled using a low-level memory representation as discussed in [WL95].

Consider the example in Figure 17 which illustrates the interaction between exceptions and implicit calls to destructors. Suppose, class A has a used-defined destructor method. Statement *2* throws an exception which is propagated to program point *3*, the exit-node of *proc1*. Before the exception is propagated from program point *3* to a call site of *proc1*, the destructor of class A is implicitly invoked at program point *3* for the object allocated for *local*. This results in a call in the presence of an active exception. Thus, this situation is similar to a call contained in a finally statement and hence it can be handled using the technique described in this paper for dealing with calls nested inside a finally statement.

10 Related work

Points-to analysis. Our algorithm differs from other context-sensitive pointer analysis algorithms [EGH94, GH98, WL95, Ruf95, CBC93, MLR[+]93] in the way it maintains context-sensitivity by associating assumed-points-to's with each data-flow element, rather than using some truncation of the call history. This manner of capturing calling context using the data-flow solution itself, enables us to obtain precise solutions for polynomial-time solvable cases and to handle exceptions. Our approximation is similar to the use of reaching aliases by Landi, Ryder and Pande [LR92, PR96]; our algorithm uses points-to's rather than aliases. The algorithm presented in this paper is a whole-program points-to analysis. [CRL99, Cha99] extended these ideas into a modular points-to analysis.

Concrete type inference and call graph construction for object-oriented languages are subsumed by our analysis because knowing the set of objects which can be referenced at run-time is tantamount to knowing their types. There are many whole-program-analysis approaches for these two problems. Most use constraint-based analysis [Suz81, PS91, PC94, Age95, GDDC97, DGC98], but a few [CHS95, PR96, DMM96] are data-flow-based. Both [CHS95, PR96] are flow- and context-sensitive algorithms for C++. [DMM96] presents a flow- and context-insensitive analysis of Modula-3. Among constraint-based approaches, [Suz81, DGC98] are flow- and context-insensitive, while [PS91, PC94, Age95, GDDC97] are flow-insensitive, but context-sensitive. None of the above approaches handle exceptions. [BS96] is an extension of hierarchy analysis for C++ and thus is flow- and context-insensitive.

Complexity. [GI98] characterizes the complexity of concrete type inference in the presence of dynamic dispatch, recursion and multi-level pointers, but does not consider flow due to exceptions. [GI98] shows that for non-recursive programs with single-level types and dynamic dispatch, concrete type inference is **PSPACE**-complete, and for recursive programs with single-level types and dynamic dispatch, concrete type inference is **EXP-TIME**-complete. Our common result is that for programs with single-level types and dynamic-dispatch, *PTA* is **PSPACE**-hard. The **PSPACE**-completeness proof we presented for intraprocedural *PTA* 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, *PTA* 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 using polynomial amount of space.

11 Conclusion

Points-to analysis is crucial for building useful software tools for object-oriented systems, because it allows for good compile-time estimates of possible interprocedural flow of control in the presence of dynamic binding, exceptions and subtyping. Such information is necessary to obtain appropriately precise side-effects, to perform data-flow-based testing, and to enable practical interprocedural program slicing.

The complexity results in this paper argue the difficulty of precise points-to analysis for programming languages which include dynamic dispatch, exceptions and subtyping (e.g., Java (or C++)). These are the first complexity results are for *PTA* aimed at a subset of Java (or C++) that includes exceptions.

The following are our main contributions:

1. The first polynomial-time algorithm for *PTA* in the presence of exceptions that handles a robust subset of Java without threads (and C++).
2. A proof that *PTA* for programs with only single-level types and exceptions without subtyping, but without dynamic dispatch, is in **P** and can be solved in $O(n^7)$ worst-case time.
3. A proof that interprocedural *PTA* for programs with only single-level types and exceptions with subtyping, but without dynamic dispatch, is **PSPACE**-hard, and the intraprocedural case is **PSPACE**-complete.
4. A proof that *PTA* for programs with only single-level types and dynamic dispatch, but without exceptions, is **PSPACE**-hard.
5. A proof that *PTA* for programs with only single-level types but without exceptions or dynamic dispatch can be solved in $O(n^5)$ worst-case time. This an improvement over the $O(n^7)$ worst-case bound achievable by applying previous approaches of [RHS95] and [LR91] to this case.
6. A proof that intraprocedural *PTA* for programs with only single-level types but without exceptions or dynamic dispatch is in non-deterministic log-space and hence *NC*. Thus, this case is efficiently parallelizable.

References

[AG96] D. Atkinson and W. Griswold. The design of whole-program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering*, pages 16–27, 1996.

[AG98] D. Atkinson and W. Griswold. Effective whole-program analysis in the presence of pointers. In *Proceedings of the ACM SIGSOFT '98 Symposium on the Foundations of Software Engineering*, pages 46–55, November 1998.

[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.

[Bar78] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, 1978.

[BCCH94] Michael Burke, Paul Carini, Jong-Doek Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, pages 234–250. Springer-Verlag, August 1994.

[BH93] S. Bates and S. Horwitz. Incremental program testing using dependence graphs. In *Conference Record of the Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 384–396, January 1993.

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

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

[Cha99] R. Chatterjee. *Modular Data-flow Analysis of Statically Typed Object-oriented Programming Languages.* PhD thesis, Department of Computer Science, Rutgers University, October 1999.

[CHS95] Paul R. Carini, Michael Hind, and Harini Srinivasan. Flow-sensitive type analysis for c++. Research Report RC 20267, IBM T. J. Watson Research Center, 1995.

[Coo89] B. G. Cooper. *Ambitious Data Flow Analysis of Procedural Programs.* Master's thesis, University of Minnesota, May 1989.

[Cou86] D. S. Coutant. Retargetable high-level alias analysis. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 110–118, January 1986.

[CR97] Ramkrishna Chatterjee and Barbara Ryder. Scalable, flow-sensitive type-inference for statically typed o-o languages. Technical Report DCS-TR-326, Dept of CS, Rutgers University, 1997.

[CR99] Ramkrishna Chatterjee and Barbara G Ryder. Data-flow-based testing of object-oriented libraries. Department of Computer Science Technical Report DCS-TR-382, Rutgers University, March 1999.

[CRL98] Ramkrishna Chatterjee, Barbara Ryder, and William Landi. Complexity of concrete type-inference in the presence of excep tions. In *LNCS 1381, Proceedings of European Symposium on Programming*, April 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*, 1999.

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

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

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

[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.

[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 constrction in object-oriented languages. In *Proceedings of ACM SIGPLAN Conference on Object-oriented Programing Systems, Languages and Applications (OOPSLA'97)*, pages 108–124, October 1997.

[GH98] R. Ghiya and L. Hendren. Putting pointer analysis to work. In *Conference Record of the Twenty-fifth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 121–133, 1998.

[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.

[GL91] K. Gallagher and J Lyle. Using program slices in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[GS96] R. Gupta and M. L. Soffa. Hybrid slicing: An approach for refining static slices using dynamic information. In *Proceedings of Third ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 29–40, 1996.

[Gua88] C. A. Guarna. A technique for analyzing pointer and structure references in parallel restructuring compilers. In *Proceedings of the International Conference on Parallel Processing*, pages 212–220, 1988.

[HA90] W. L. Harrison III and Z. Ammarguellat. Parcel and Miprac: parallelizers for symbolic and numeric programs. In *Proceedings of International Workshop on Compilers for Parallel Computers*, pages 329–346. Ecole des Mines de Paris - CAI, UPMC - Laboratoire MASI, December 1990. Paris, France.

[HC98] M. J. Harrold and N. Ci. Reuse-driven interprocedural slicing. In *Proceedings of the 20th International Conference on Software Engineering*, pages 74–83, April 1998.

[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.

[HP98] M. Hind and A. Pioli. Assessing the effects of flow sensitivity on pointer alias analysis. In *Proceedings of International Static Analysis Symposiusm (SAS'98)*, pages 57–81. Springer-Verlag, September 1998.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), January 1990.

[HS91] M. J. Harrold and M. L. Soffa. Selecting and using data for integration testing. *IEEE Software*, 8(2):58–65, March 1991.

[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.

[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.

[LH96] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *Proceedings of the 18th International Conference on Software Engineering*, pages 495–505, March 1996.

[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*, pages 93–103, January 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*, pages 235–248, June 1992.

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

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

[OO84] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, May 1984.

[Ost90] Thomas J. Ostrand. Data-flow testing with pointers and function calls. In *Proceedings of the Pacific Northwest Software Quality Conference*, October 1990.

[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.

[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.

[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.

[RM99] M.P. Robillard and G.C. Murphy. Analyzing exception flow in java programs. In *Proceedings of Seventh European Software Engineering Conference and Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 322–337, September 1999. Also available as LNCS 1687.

[RR95] T. Reps and G. Rosay. Precise interprocedural chopping. In *Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineeering*, pages 41–52, 1995.

[Ruf95] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, June 1995.

[SFRW90] S. Sagiv, N. Francez, M. Rodeh, and R. Wilhelm. A logic-based approach to data flow analysis. In *Proceedings of the Second International Workshop in Programming Language Implementation and Logic Programming*, pages 277–292, August 1990. Volume 456 of Lecture Notes in Computer Science.

[SH97a] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Proceedings of the Fourth International Symposium on Static Analysis (SAS'97)*, pages 16–34, September 1997.

[SH97b] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of the Twenty-fourth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 1–14, January 1997.

[SH98] S. Sinha and M.J. Harrold. Analysis of programs that contain exception-handling constructs. In *Proceedings of International Conference on Software Maintenance*, pages 348–357, November 1998.

[SHR99] S. Sinha, M. J. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 21st International Conference on Software Engineering*, pages 432–441, May 1999.

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

[Ste96a] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference*

Record of the Twenty-third Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, pages 32–41, 1996.

[Ste96b] 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.

[TAFM97] P. Tonella, G. Antoniol, R. Fiutern, and E. Merlo. Flow-insensitive c^{++} pointers and polymorphism analysis and its application to slicing. In *Proceedings of the 19th International Conference on Software Engineering (ICSE97)*, pages 433–443, 1997.

[TCFR96] 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 1996.

[Tip96] Frank Tip. A survey of program slicing technques. *Journal of Programming Languages*, 3(3):121–189, 1996.

[Ven91] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 107–119, June 1991.

[Wei80] W. E. Weihl. *Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables and Label Variables*. Master's thesis, M.I.T., June 1980.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[Wey94] E. Weyuker. More experience with data flow testing. *IEEE Transactions on Software Engineering*, 19(9):912–919, September 1994.

[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] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step towards practical analyses. In *Proceedings of the 4th Annual ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 81–92, October 1996.

[ZRL98] Sean Zhang, Barbara G. Ryder, and William A. Landi. Experiments with combined analysis for pointer aliasing. In *Proceedings of ACM SIGPLAN Workshop on Program Analysis and Software Tools for Engineering*, pages 11–18, June 1998.

```
void process_exit_node( exit_node, ndf_elm ) {
  // Let M be the method containing the exit_node.
  if ( ndf_elm represents the value of a local variable )
    // it need not be forwarded to the successors (return-nodes of call sites for
    // this method) because the local variable is not visible outside this method.
    return;

  if ( ndf_elm is reachable ) {
    for ( each call site C in the current set of call sites of M ){
      if ( solution at C contains reachable ) {
        add_to_solution_and_worklist_if_needed( ndf_elm, R );
        // R is the return-node for C.
        for ( each s in C.waiting_local_points_to_table ) {
          // conditional-points-tos representing values of local variables reaching
          // C are not forwarded to R until it is found reachable.
          // C.waiting_local_points_to_table contains such conditional-points-tos.

          // Since R has been found to be reachable
          delete s from C.waiting_local_poinst_to_table;
          add_to_solution_and_worklist_if_needed( s, R );
        }
      }
    }
    return;
  }

  add_to_table_of_conditions( ndf_elm, exit_node );
  // This table is accessed from the call sites of M for expanding assumed-points-tos.

  for ( each call site C in the current set of call sites of M ) {
    S = get_assumed_points_tos( C, ndf_elm.assumed_points_to, M );
    for ( each assumed_points_to Apt in S ) {
      CPT = new conditional-points-to( Apt, ndf_elm.points_to );
      add_to_solution_and_worklist_if_needed( CPT, R );
      // R is the return-node for C.
    }
  }
} // end of process_exit_node
```

Figure 18: Code for processing an exit-node

A Pseudocode for *process_exit_node* and *process_call_node* of the *basic* algorithm

Auxiliary functions used by the routines given in this appendix are defined in Appendix B.

```
void process_call_node( C, ndf_elm ){
// R is the return-node for call_node C.
  if ( ndf_elm implies an increase in the set CM of methods invoked
      from this site ) {
    // Recall that due to dynamic dispatch, the interprocedural
    // edges are constructed on the fly, as in [EGH94].
    add this new method nM to CM;
    for ( each dfelm in the solution set of C )
      interprocedurally_propagate( dfelm, C, nM ); // defined in Figure 20
  }

  if ( ndf_elm represents value of a local variable ) {
    if ( solution set for R contains reachable )
      // Forward ndf_elm to the return-node because (unlike C++)
      // a local variable cannot be modified by a call in Java.
      add_to_solution_and_worklist_if_needed( ndf_elm, R );
    else
      // Cannot forward till R is found to be reachable.
      add ndf_elm to waiting_local_points_to_table;
  }

  for ( each method M in CM )
    interprocedurally_propagate( ndf_elm, C, M );
}
```

Figure 19: Code for processing a call-node

```
interprocedurally_propagate( ndf_elm, C, M) {
  // C is a call_node, R is the return-node of C and M is a method called from C.
  if ( ndf_elm == reachable ) {
    add_to_solution_and_worklist_if_needed(ndf_elm, M.entry_node);
    if ( M.exit_node has reachable ) {
      add_to_solution_and_worklist_if_needed(ndf_elm, R);
      for ( each s in C.waiting_local_points_to_table ) {
        // Since R has been found to be reachable
        delete s from C.waiting_local_poinst_to_table;
        add_to_solution_and_worklist_if_needed( s, R );
      }
    }
    propagate_conditional_points_tos_with_empty_condition(C,M);
    return;
  }

  // get the points-tos implied by ndf_elm at the entry-node of M
  S = get_implied_conditional_points_tos(ndf_elm,M,C);

  for ( each s in S ) {
    add_to_solution_and_worklist_if_needed( s, M.entry_node );
    add_to_table_of_assumed_points_tos( s.assumed_points_to,
      ndf_elm.assumed_points_to, C );
    // This table is accessed from exit-nodes of methods called from C
    // for expanding assumed-points-tos.

    if ( ndf_elem.apt is a new apt for s.apt ) {
      // apt stands for assumed-points-to
      Pts = lookup_table_of_conditions( s.assumed_points_to, M.exit_node );
      // ndf_elm.assumed_points_to is an assumed-points-to for each element of Pts

      for ( each pts in Pts ) {
        cpt = new conditional_points_to( ndf_elm.assumed_points_to, pts );
        add_to_solution_and_worklist_if_needed( cpt, R );
      }
    }
  } // end of each s in S
}
```

Figure 20: Code for interprocedurally_propagate

```
// CPT stands for a conditional-points-to. DFE stands for a
// data-flow-element which could be a CPT or 'reachable'.
--------------------------------------------------------------------
set of data-flow-elements apply( node, rDFE ) {
// this function computes the effect of the statement (if any) associated
// with node on rDFE, i.e. the resulting data-flow-elements at the bottom
// of node.
  set of data-flow-elements retVal = empty set;
  if ( node is a not a pointer assignment node ) {
    add rDFE to retVal;
    return retVal;
  }

  if ( rDFE == reachable ) {
    add rDFE to retVal;
    if (node unconditionally generates a conditional-points-to) {
      // e.g. l: p = new A; unconditionally generates
      // ⟨empty,⟨p,object_l⟩⟩.
      add this conditional-points-to to retVal;
    }
  }
  else {
    // rDFE is a conditional-points-to
    lhs_set = combine compatible CPTs in the solution set computed
      so far (including rDFE) to generate the set of locations
      represented by the left-hand-side.
    // Note: each element in lhs_set has a set of
    // assumed-points-to's associated with it

    similarly compute rhs_set for the right-hand-side.

    retVal = combine compatible elements from lhs_set and rhs_set.
    // only one of the assumed-points-to's associated with a
    // resulting points-to is chosen as its assumed-points-to.

    if ( rDFE is not killed by this node )
      add rDFE to retVal;
  }
  return retVal;
} // end of apply
```

Figure 21: Pseudocode for apply

B Auxiliary functions

The pseudocode for the auxiliary functions used by the *basic* algorithm is given in Figures 21 and 22.

```
void add_to_table_of_conditions( rCPT, exit-node ) {
  // each exit-node has a table condTable associated with it,
  // which stores for each assumed-points-to, the points-to's
  // which hold at the exit-node with this assumed-points-to.
  // This function stores rCPT in this table.   }
```

```
void add_to_table_of_assumed_points_tos(s, condition, C) {
// C is a call-node, condition is an assumed-points-to and s is a
// points-to passed to the entry-node of a method invoked from C.
// Each call-node has a table asPtTtable associated with it, which
// stores for each points-to that is passed to the entry-node of a
// method invoked from this site, the assumed-points-to's which
// imply this point-to at the call site. This function stores
// condition with s in this table.   }
```

```
set of points-to's get_assumed_points_tos( C, s, M ) {
  if ( s is empty ) {
    if ( the solution set at C does not contain reachable )
      return empty set;
    else {
      if (C is a dynamically-dispatched-call site)
        return the set of assumed-points-to's for the values
        of receiver, which result in a call to method M;
      else
        return a set containing empty;
    }
  }
  else {
    return the assumed-points-to's stored with s in C.asPtTtable;
    // asPtTtable is defined in add_to_table_of_assumed_points_tos.
  }
}
```

```
set of points-to's lookup_table_of_conditions( condition, exit-node ) {
  // It returns the points-to's stored with condition in
  // exit-node.condTable, which is defined in add_to_table_of_conditions.
}
```

```
set of conditional-points-to's get_implied_conditional_points_tos( rCPT, M, C ) {
  // It returns conditional-points-to's implied by rCPT.points-to
  // at the entry-node of method M at call-node C. This means it also
  // performs actual to formal binding. Note that it may return
   // an empty set.   }
```

```
void propagate_conditional_points_tos_with_empty_condition( C, M) {
  if ( C is not a dynamically_dispatched_call site )
    S = { empty };
  else
    S = set of assumed-points-to's for the values
      of receiver, which result in a call to method M;

  Pts = lookup_table_of_conditions( empty, M.exit_node );
  for ( each s in S ) {
    for ( each pts in Pts ) {
      cpt = new conditional_points_to( s, pts );
      add_to_solution_and_worklist_if_needed( cpt, R );
      // R is the return-node of C
    }
  }
}
```

Figure 22: Pseudocode for auxiliary functions

C Proof of precision of the *extended* algorithm

In this section, we show that the algorithm described in section 6.1 computes the precise solution for programs with only single-level types and exceptions without subtyping (and without dynamically-dispatched calls).

definition 2 *An exception block is a body of try, catch, finally or method.*

definition 3 *Let l be a program point in a method M. l is said to be directly contained in a finally if and only if the innermost exception block containing it is a finally statement.*

definition 4 *Let l be a program point in a method M. l is said to be contained in a nested try-catch-finally if and only if it is contained in a try-catch-finally which is nested inside a finally.*

definition 5 *Let l be a program point in a method M. $\bar{rp}$(r,t,p,l,M) denotes a realizable path $r = t+p$ from the start-node to l, such that*

- *either l is not contained in a nested try-catch-finally and p is a balanced path from the entry-node of M to l and t is a realizable path from the start-node to the entry-node of M, or*
- *l is contained in a nested try-catch-finally S (innermost) and p is a balanced path from the entry-node of S to l and t is a realizable path from the start-node to the entry-node of S.*

definition 6 *Let l be a program point in a method M. rpwoe(r,t,p,l,M) denotes a $\bar{rp}$(r,t,p,l,M) such that when r is followed, at l, there is no uncaught exception generated*[14] *in p.*

definition 7 *Let l be a program point in a method M. rpwe(r,t,p,l,M,excp-type) denotes a $\bar{rp}$(r,t,p,l,M) such that when r is followed, at l, there is an uncaught exception generated in p of type excp-type.*

Lemma 8 *Let l be a program point in a method M and r be a realizable path from the start-node to l. Then exactly one of the following hold:*

- *l is not contained in a nested try-catch-finally and r can be written as $r = t+p$, where t is a realizable path from the start-node to the entry-node of M and p is a balanced path from the entry-node of M to l.*
- *l is contained in a nested try-catch-finally S (innermost) and r can be written as $r = t + p$, where t is a realizable path from the start-node to the entry-node of S and p is a balanced path from the entry-node of S to l.*

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

Lemma 9 *Let l be a program point in a method M.*

1. *If ⟨reachable ⟩ is computed by the algorithm at l, then*
   - *l is not directly contained in a finally statement, and*
   - *there exists an rpwoe(r,t,p,l,M).*
2. *If ⟨excp-type, reachable ⟩ is computed by the algorithm at l, there exists an rpwe(r,t,p,l,M,excp-type).*
3. *If a conditional-points-to ⟨z, u ⟩*[15] *is computed by the algorithm at l, then*
   - *l is not directly contained in a finally statement, and*
   - *there exists an rpwoe(r,t,p,l,M) such that u holds at l, and*
   - *for any $\bar{rp}$(q,s,p,l,M), u holds at l if and only if z holds at the first node of p.*
4. *If an exceptional-conditional-points-to ⟨excp-type, z, u ⟩ is computed by the algorithm at l, then*
   - *there exists an rpwe(r,t,p,l,M,excp-type) such that u holds at l, and*
   - *for any $\bar{rp}$(q,s,p,l,M), u holds at l if and only if z holds at the first node of p.*
5. *If ⟨exp, z, obj ⟩ is computed by the algorithm at l, then*
   - *there exists an $\bar{rp}$(r,t,p,l,M) such that when r is followed, there exists an uncaught exception-object obj at l, which is generated by a throw in p, and*
   - *for any $\bar{rp}$(q,s,p,l,M), at l, there exists an uncaught exception-object obj generated by a throw in p if and only if z holds at the first node of p.*
6. *If a labelled-conditional-points-to ⟨label, z, u ⟩ is computed by the algorithm at l, then*
   - *l is directly contained in a finally statement, and*
   - *there exists an rpwoe(r,t,p,l,M) such that u holds at l and the reason for entering the finally statement (entry corresponding to the last node of p i.e. l) is an exit to the statement numbered label from the try statement or one of the catch statements associated with the finally statement, and*
   - *for any $\bar{rp}$(q,s,p,l,M), u holds at l if and only if z holds at the first node of p.*
7. *If ⟨label, reachable⟩ is computed by the algorithm at l, then*
   - *l is directly contained in a finally statement, and*
   - *there exists an rpwoe(r,t,p,l,M) such that the reason for entering the finally statement is an exit to the statement numbered label from the try statement or one of the catch statements associated with the finally statement.*

[14] Due to a *finally* clause, an uncaught exception generated in t may be active when control reaches a point in p.

[15] z could be empty.

proof The 7 claims can be proved simultaneously using induction on the number of iterations needed to compute ⟨*reachable*⟩, ⟨*excp-type, reachable* ⟩, ⟨z,u⟩, ⟨*excp-type,z,u* ⟩, ⟨exp, z, obj⟩, ⟨$label, z, u$⟩ or ⟨$label, reachable$⟩. □

Lemma 10 *Suppose there exists an $\bar{rp}$(r,t,p,l,M). The following hold with respect to it.*

1. *If l is not directly contained a finally statement and $\bar{rp}$(r,t,p,l,M) is an rpwoe(r,t,p,l,M), ⟨reachable ⟩ is computed by the algorithm at l.*
2. *If l is directly contained in a finally statement and $\bar{rp}$(r,t,p,l,M) is an rpwoe(r,t,p,l,M), ⟨label, reachable ⟩ is computed by the algorithm at l for some label such that the reason for entering the finally statement is an exit to the statement numbered label from the try statement or one of the catch statements associated with the finally statement.*
3. *If $\bar{rp}$(r,t,p,l,M) is an rpwe(r,t,p,l,M,excp-type), ⟨excp-type, reachable ⟩ is computed by the algorithm at l.*
4. *If l is not directly contained in a finally statement and $\bar{rp}$(r,t,p,l,M) is an rpwoe(r,t,p,l,M) and a points-to u holds at l when r is followed, then*
   - *there exists a points-to z such that ⟨z, u ⟩ is computed by the algorithm at l, and*
   - *for any $\bar{rp}$(q,s,p,l,M), u holds at l if and only if z holds at the first node of p.*
5. *If l is directly contained in a finally statement and $\bar{rp}$(r,t,p,l,M) is an rpwoe(r,t,p,l,M) and a points-to u holds at l when r is followed, then*
   - *there exists a points-to z and a label v such that ⟨v, z, u ⟩ is computed by the algorithm at l, and*
   - *the reason for entering the finally statement is an exit to the statement numbered v from the try statement or one of the catch statements associated with the finally statement, and*
   - *for any $\bar{rp}$(q,s,p,l,M), u holds at l if and only if z holds at the first node of p.*
6. *If $\bar{rp}$(r,t,p,l,M) is an rpwe(r,t,p,l,M,excp-type) and a points-to u holds at l when r is followed, then*
   - *there exists a points-to z such that ⟨excp-type, z, u ⟩ is computed by the algorithm at l, and*
   - *for any $\bar{rp}$(q,s,p,l,M), u holds at l if and only if z holds at the first node of p.*
7. *When r is followed, if there exists an uncaught exception-object obj at l, which is generated by a throw in p, then*
   - *there exists a points-to z such that ⟨exp, z, obj ⟩ is computed by the algorithm at l, and*
   - *for any $\bar{rp}$(q,s,p,l,M), at l, there exists an uncaught exception-object obj generated by a throw in p if and only if z holds at the first node of p.*

proof The 7 claims can be proved simultaneously by induction on the length of r. □

Theorem 8 *The algorithm described in section 6.1 computes the precise solution for programs with only single-level types and exceptions without subtyping, and without dynamically-dispatched calls.*

proof: Lemma 9 (claims 3, 4 and 6) implies that the solution computed by the algorithm is a subset of the precise solution. While lemma 8 and lemma 10 (claims 4, 5 and 6) imply that the precise solution is a subset of the solution computed by the algorithm. Hence, we have the theorem. □

D Safety of the *extended* algorithm in the general case

In this section, we prove that the algorithm presented in 6.1 computes a safe solution for programs written in *JavaWoThreads*. Lemma 11 is a weakened form of lemma 10. The main difference is that in lemma 11 the assumed-points-to z at the first node of p is no longer necessary and sufficient for u to hold. Instead, when r is followed, z is just one the points-to's in the set of points-to's that hold at the first node of p and together imply u.

Lemma 11 *Suppose there exists an $\bar{rp}$(r,t,p,l,M). The following hold with respect to it.*

1. *If l is not directly contained in any finally statement and $\bar{rp}$(r,t,p,l,M) is an rpwoe(r,t,p,l,M), ⟨reachable ⟩ is computed by the algorithm at l.*
2. *If l is directly contained in a finally statement and $\bar{rp}$(r,t,p,l,M) is an rpwoe(r,t,p,l,M), ⟨label, reachable ⟩ is computed by the algorithm at l for some label such that the reason for entering the finally statement is an exit to the statement numbered label from the try statement or one of the catch statements associated with the finally statement.*
3. *If $\bar{rp}$(r,t,p,l,M) is an rpwe(r,t,p,l,M,excp-type), ⟨excp-type, reachable ⟩ is computed by the algorithm at l.*
4. *If l is not directly contained in any finally statement and $\bar{rp}$(r,t,p,l,M) is an rpwoe(r,t,p,l,M) and a points-to u holds at l when r is followed, then*
    - *there exists a points-to z such that ⟨z, u ⟩ is computed by the algorithm at l, and*
    - *when r is followed, z holds at the first node of p.*
5. *If l is directly contained in a finally statement and $\bar{rp}$(r,t,p,l,M) is an rpwoe(r,t,p,l,M) and a points-to u holds at l when r is followed, then*
    - *there exists a points-to z and label v such that ⟨v, z, u ⟩ is computed by the algorithm at l, and*
    - *the reason for entering the finally statement is an exit to the statement numbered v from the try statement or one of the catch statements associated with the finally statement, and*
    - *when r is followed, z holds at the first node of p.*
6. *If $\bar{rp}$(r,t,p,l,M) is an rpwe(r,t,p,l,M,excp-type) and a points-to u holds at l when r is followed, then*
    - *there exists a points-to z such that ⟨excp-type, z, u ⟩ is computed by the algorithm at l, and*
    - *when r is followed, z holds at the first node of p.*
7. *When r is followed, if there exists an uncaught exception-object obj at l, which is generated by a throw in p, then*
    - *there exists a points-to z such that ⟨exp, z, obj ⟩ is computed by the algorithm at l, and*
    - *when r is followed, z holds at the first node of p.*

proof The 7 claims can be proved simultaneously by induction on the length of r.

Theorem 9 *The extended algorithm computes a safe solution for programs written in a subset of Java defined in section 2.10.*

proof Let l be a program point in a method M, r be a realizable path from the start-node to l and VR be a reference variable (defined in section 2.4). Further, when r is followed, suppose VR points-to *obj* at l. Lemma 8 implies that r is an $\bar{rp}$*(r,t,p,l,M)*. Now there are two cases. First (case 1), suppose that when r is followed, at l, there is an uncaught exception generated in p of type *excp-type*. Since the relationship between VR and *obj* is due to a set of points-to's, each of these points-to's holds at l when r is followed. Using lemma 8 and lemma 11, each of these points-to's is computed by the algorithm as exceptional-conditional-points-to with *excp-type* and some assumed-points-to, and each of these assumed-points-to's holds at the first node of p when r is followed. Thus, the conditional parts of these exceptional-conditional-points-to's are compatible and hence this relationship will be reported by the algorithm. Now (case 2) suppose that when r is followed, at l, there is no uncaught exception generated in p. Again, the relationship between VR and *obj* is due to a set of points-to's which hold at l when r is followed. Suppose l is not contained in a *finally*. Using lemma 8 and lemma 11, each of these points-to's is computed by the algorithm with some assumed-points-to, and each of these assumed-points-to holds at the first node of p when r is followed. Thus, these assumed-points-to's are compatible and this relationship will be reported by the algorithm. Next, suppose l is contained in a *finally*. Again, using lemma 8 and lemma 11, each of these points-to's is computed by the algorithm with some assumed-points-to and label. These labels represent the reason for entering the the *finally* when r is followed. Hence they are same. Moreover, the assumed-points-to's hold at the first node of p when r is followed. Thus these are compatible. Hence this relationship will be reported by the algorithm.

About the authors:

Ramkrishna Chatterjee received a Master of Engineering degree in Computer Science and Engineering from Indian Institute of Science, Bangalore, India in 1993. From August 1993 to July 1994 he worked as a software engineer for TATA-IBM in Bangalore, India. He received a Ph.D. in Computer Science from Rutgers University, USA, in 1999. His current research interests include software engineering tools, data-flow analysis of object-oriented languages, multi-version databases and long transaction management. He currently works for Oracle Corporation in Nashua, New Hampshire, USA.

Barbara G. Ryder (http://www.cs.rutgers.edu/ ryder) is a Professor of Computer Science at Rutgers University, New Brunswick, NJ. Dr. Ryder is a Fellow of the ACM and serves on the Board of Directors of the Computer Research Association (CRA). Dr. Ryder was General Chair of the 1999 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'99). She served on the ACM SIGPLAN Executive Committee from 1989-1999, as Vice Chair for Conferences (1993-1995) and as Chair (1995-1997). She was a recipient of an NSF Faculty Award for Women Scientists and Engineers (1991-1996). Her current research is focused on static analysis and compilation of object-oriented languages, especially Java.

William A. Landi received his Ph.D. in Computer Science from Rutgers University in 1992. His research interests include pointer alias analysis and practical applications for static analysis. He is currently employed by Siemens Corporate Research where he is working on simple analysis for large software systems, up to 20 million lines of code, as well as software issues for manufacturing applications.