Complexity of Concrete Type-inference in the Presence of Exceptions*

Ramkrishna Chatterjee[1] Barbara G. Ryder[1] William A. Landi[2]

[1] Department of Computer Science, Rutgers University, Piscataway NJ 08855, USA
+1 908 445 2001 {ramkrish,ryder}@cs.rutgers.edu
[2] Siemens Corporate Research Inc, 755 College Rd. East, Princeton, NJ 08540 USA, wlandi@scr.siemens.com

DCS-TR-341

Abstract. Concrete type-inference for statically typed object-oriented programming languages (e.g., Java, C^{++}) determines at each program point, those objects to which a reference may refer or a pointer may point during execution. A *precise* compile-time solution for this problem requires a flow-sensitive analysis. Our new complexity results for concrete type-inference distinguish the difficulty of the intraprocedural and interprocedural problem for languages with combinations of single-level types[3], exceptions with or without subtyping, and dynamic dispatch. Our results include:

- The first polynomial-time algorithm for concrete type-inference in the presence of exceptions, which handles Java without threads, and C^{++};
- Proofs that the above algorithm is always safe and provably precise on programs with single-level types, exceptions without subtyping, and without dynamic dispatch;
- Proof that interprocedural concrete type-inference problem with single-level types and exceptions with subtyping, and without dynamic dispatch is **PSPACE-hard**, while the intraprocedural problem is **PSPACE-complete**.

Other complexity characterizations of concrete type-inference for programs without exceptions are also presented.

1 Introduction

Concrete type-inference (*CTI* from now on) for statically typed object-oriented programming languages (e.g., Java, C^{++}) determines at each program point, those objects to which a reference may refer or a pointer may point during execution. This information is crucial for static resolution of dynamically dispatched calls, side-effect analysis, testing, program slicing and aggressive compiler optimization.

* The research reported here was supported, in part, by NSF grant GER-9023628 and the Hewlett-Packard Corporation.

[3] These are types with data members only of primitive types.

The problem of *CTI* is both intraprocedurally and interprocedurally flow-sensitive. However, there are approaches with varying degrees of flow-sensitivity for this problem. Although some of these have been used for pointer analysis of C, they can be adapted for *CTI* of Java without exceptions and threads, or C^{++} without exceptions. At the one end of the spectrum are intraprocedurally and interprocedurally flow-insensitive approaches [Ste96, SH97, ZRL96, And94], which are the least expensive, but also the most imprecise. While at the other end are intraprocedurally and interprocedurally flow-sensitive approaches [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.

An intraprocedurally flow-insensitive algorithm does not distinguish between program points within a method; hence it reports the same solution for all program points within each method. In contrast, an intraprocedurally flow-sensitive algorithm tries to compute different solutions for distinct program points. The example in Figure 5 in Appendix C illustrates this difference.

An interprocedurally flow-sensitive (i.e. context-sensitive) algorithm considers (sometimes approximately) only interprocedurally *realizable paths* [RHS95, LR91]: paths along which calls and returns are properly matched, while an interprocedurally flow-insensitive (i.e. context-insensitive) algorithm does not make this distinction. The example in Figure 6 in Appendix C illustrates this difference. For the rest of this paper, we will use the term *flow-sensitive* to refer to an intra- and interprocedurally flow-sensitive analysis.

In this paper, we are interested in a flow-sensitive algorithm for *CTI* of a robust subset of Java with exceptions, but without threads (this subset is described in Section 2). The complexity of flow-sensitive *CTI* in the presence of exceptions has not been studied previously. None of the previous flow-sensitive pointer analysis algorithms [LR92, WL95, EGH94, PR96, Ruf95, CBC93, MLR+93] for C/C^{++} handle exceptions. However, unlike in C^{++}, exceptions are *frequently* used in Java programs, making it an *important* problem for Java.

The main contributions of this paper are:

- The first polynomial-time algorithm for *CTI* in the presence of exceptions that handles a robust subset of Java without threads, and C^{++}[4],
- 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 *CTI* 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 *CTI* in the absence of exceptions.

These results are summarized in Table 1, which also gives the sections of the paper containing these results.

[4] In this paper, we present our algorithm only for Java.

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

Table 1. Complexity results for *CTI* summarized

The rest of this paper is organized as follows. First, we present a flow-sensitive algorithm, called the *basic* algorithm, for *CTI* in the absence of exceptions, and prove our results about complexity of *CTI* in the absence of exceptions. Next, we prove **PSPACE**-hardness results about *CTI* in the presence of exceptions. Finally, we present an extension of the *basic* algorithm for *CTI* 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. 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 3. We will denote the entry-node of *main* by *start-node* in the rest of this paper.

2.2 Representation of dynamically created objects

All run-time objects (or arrays) created at a program point n are represented symbolically by *object_n*. No distinction is made between different elements of an array. Thus, if an array is created at n, *object_n* represents all elements of the array.

2.3 Precise solution for *CTI*

A *reference variable* is one of the following:

- a static variable (class variable) of reference[5] type;
- a local variable of reference type;
- 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
- $V.s_1...s_k$, 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.

Using these definitions, the precise solution for *CTI* can be defined as follows: given a *reference variable RV* and an object *object_n*, ⟨*RV,object_n* ⟩ 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.*

2.4 Points-to

A points-to has the form ⟨*var,obj* ⟩; 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, an of object created at a program point n; and *obj* is *object_s* - an object created at a program point s.

2.5 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.

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

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 *CTI*, 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.

2.6 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.

2.7 Safe solution

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

2.8 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. Since finalize methods are called (non-deterministically) during garbage collection or unloading of classes, if a *finalize* method modifies a variable of reference type (extremely rare), it cannot be handled by our algorithm. Static initializations complicate analysis due to dynamic loading of classes. If static initializations can be done in program order, our algorithm can handle them. Otherwise, if they depend upon dynamic loading (extremely rare), our algorithm cannot handle them. Similarly, our algorithm cannot handle classes that are constructed on the fly and not known statically. We will refer to this subset as *JavaWoThreads*.

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 what is useful.

3 *CTI* in the absence of exceptions

Our *basic* algorithm for *CTI* 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-tos. 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-tos are computed conditioned on *assumed-points-tos* (akin to reaching alias in [LR92] [PR96]), which represent points-tos reaching the entry of a method, and approximate the calling context in which the method has been called (see the example in Appendix A). A points-to along with its *assumed-points-to* is called a *conditional-points-to.* A conditional-points-to has the form $\langle$*condition, points-to*$\rangle$, where *condition* is an assumed-points-to or *empty* (meaning this points-to is applicable to all contexts). 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-tos 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-tos and the data-flow element *reachable.*

3.2 Query

Using these conditional-points-tos, a query for *CTI* is answered as follows. Given a *reference variable* V and a program point l, the conditional-points-tos with *compatible* assumed-points-tos computed at l are combined to determine the possible values of V. Assumed-points-tos are *compatible* if and only if they do not imply different values for the same user defined variable. For example, if V is *p.f1*, and the solution computed at l contains $\langle$*empty*, $\langle$*p*, *obj1* $\rangle\rangle$, $\langle$z, $\langle$*obj1.f1*, *obj2* $\rangle\rangle$ and $\langle$u, $\langle$*obj1.f1*, *obj3* $\rangle\rangle$, then the possible values of V are *obj2* and *obj3.*

3.3 Algorithm description

Figure 1 contains a high-level description of the main loop of the *basic* algorithm. *apply* computes the effect of a statement on an incoming conditional-points-to. For example, suppose l labels the statement $p.f1 = q$, *ndf_elm* (i.e. the 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 at l so far. 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 condition for the generated data-flow element. For example, if u is chosen then $\langle u$, $\langle$*object_s.f1, object_n* $\rangle\rangle$ will be generated. When a conjunction of conditions is associated with a points-to, any fixed-size subset of these conditions may be stored without affecting safety. At a program point where this data-flow element is used, if all the conjuncts are true then any subset of the conjuncts is also true. This may cause overestimation of solution at program points where only a proper subset of the conjuncts is true. At present, we store only the first member of the list of conditions. *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 $\langle z,\ u \rangle$ holds 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 $\langle x,\ t \rangle$ is in the solution set at *C* and t implies z at the entry-node of *M*, $\langle x,u \rangle$ is propagated by *process_exit_node* to *R*. *process_exit_node* is defined in Figure 2.

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 $\langle x,\ t \rangle$ holds at a call site *C* which has a method *M* in its set of callable methods computed so far. If t implies a points-to z at the entry-node of *M* (e.g., through an actual to formal binding), $\langle z,z \rangle$ is forwarded to the entry-node of *M*. *process_call_node* also remembers the association between x and z at *C* because this is used by *process_exit_node* as described above. *process_call_node* is defined in Figures 3 and 4.

Other functions used by the above routines are defined in Appendix B. Appendix A contains an example which illustrates the *basic* algorithm.

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

In this section, we show [6] 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.

[6] Our proof is strongly influenced by a similar proof for aliases in [LR91].

```
// 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
```

Fig. 1. High-level description of the *basic* algorithm

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 $r = t + p$* [7] *from the start-node to l, such that p is a balanced* [8] *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. □

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 $\langle z,\ u \rangle$*[9] *is computed by the algorithm at l, then*
   – *there exists a rp(r,t,p,l,M) such that u holds at l, and*

[7] + indicates concatenation.

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

[9] z could be empty.

```
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
```

Fig. 2. Code for processing an exit-node

– *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 $\langle \textit{reachable} \rangle$. □

Remark: For single-level types, a conditional-points-to at a program point can never require the simultaneous occurrence of multiple conditional-points-tos 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. *$\langle \textit{reachable} \rangle$ is computed by the algorithm at l.*

```
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 4
  }

  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 );
}
```

Fig. 3. Code for processing a call-node

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 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:

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.*

```
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
}
```

Fig. 4. Code for interprocedurally_propagate

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-tos 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-tos[10]. 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-tos at the entry-node of the called method. In addition $O(CV^2L)$ work may be done in passing points-tos 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-tos 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-tos 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 *CTI* for programs with only single-level types and without dynamic dispatch, exceptions or threads

Theorem 2 *Intraprocedural CTI 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-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. $\square$

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

5 Complexity of *CTI* in the presence of exceptions

Semantics of finally: Before describing our results about *CTI* 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.

Now, in the rest of this section, we show that intraprocedural *CTI* for programs with only single-level types and exceptions with subtyping is **PSPACE**-complete, while the interprocedural case without dynamic dispatch is **PSPACE**-hard. In [Lan92a], Bill Landi showed that intraprocedural may-alias analysis for a subset of C in the presence of pointers with at most 4 levels of indirection[11] 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 may-points-to analysis[12] in the presence of pointers with at most 4 levels of indirection and without dynamic allocation is **PSPACE**-complete. Moreover, it is easy to modify this proof to show that intraprocedural may-points-to analysis in the presence of pointers with at most 2 levels of indirection and without dynamic allocation (PTA for short) is **PSPACE**-complete. We will reduce PTA in polynomial time to the problem of *CTI* 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 (TIA for short). Consider an instance of PTA:

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

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

[12] 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.

```
  ...
   // pointer assignments, ifs and whiles

}
```

The corresponding instance of TIA 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 TIA {
  static void proc2( )
  {
    int_type r1, ..., rm;
    int_type q1_var, ..., qn_var;
    q1_type q1_addr;
    ...
    qn_type qn_addr;

    multi_level_pointer p1, ..., ps;

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

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

    ...
  }
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ifs and whiles are translated verbatim, except the test expressions, which we assume (without loss of generality) to be side-effect free and hence ignore in our analysis.

Other statements that do not modify any pointer variable are translated into empty statements. This does not preserve program semantics, but it does not affect the mapping between the the points-to solution and the solution for CTI.

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 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 TIA, constructed above, has $O(n^3)$ statements, as each statement in PTA is replaced by at most $O(n^2)$ statements. As a result, we have a polynomial-time reduction from PTA to TIA. Note that if we start with an instance of 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 TIA of size at most $O(n^7)$. In general, if we start with at most k levels of indirection, we will get an equivalent instance of TIA of size at most $O(n^{2(k-1)+1})$. As a result, even if we use Bill Landi's proof for 4 levels of indirection, the above construction shows that intraprocedural TIA is **PSPACE**-hard.

Now we will show that intraprocedural TIA is in **PSPACE**, and hence **PSPACE**-complete. Savitch's theorem [HU79] implies that **PSPACE= NPSPACE**[13]. Hence, it is enough to show that intraprocedural TIA is in **NPSPACE**. Our proof is similar to Bill Landi's proof [Lan92a] that PTA is in **NPSPACE**. We present a non-deterministic algorithm for TIA, which uses polynomial amount

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

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*. *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 old *S*, and then non-deterministically (except for a *throw* or exit-node of a *try*, *catch* or *finally*) 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.

The innermost construct containing a *throw* is one of the following: *try*, *catch*, *finally* or a method body. In the following, we consider only the first case; others are handled similarly. For each *try* statement, the algorithm maintains an array (*catch-table*) indexed by types, which stores a pointer to the entry of a *catch* statement [14] applicable when the exit-node of the *try* is reached with an exception of this type. These tables can be easily built in polynomial time by making a prepass through the method. At a *throw* statement, the algorithm instantiates *excp-var* with the exception object, and instead of non-deterministically choosing a successor, uses the type of the exception object to lookup the array associated with the innermost *try* statement containing the *throw* statement to find the entry-node of the appropriate *catch* statement (or entry-node of a *finally* statement or exit-node of *M* or exit-node another enclosing *try* statement). This node is then chosen as the successor to be considered next. On entering a *catch*, the value of *excp-var* is used to instantiate the parameter of the *catch*, after which *excp-var* is reinitialized to null.

At the exit-node of a *finally*, the value of *excp-var* or *label-var* is used for deciding the successor. When a *try-catch-finally* nested inside another *finally* is

[14] or entry-node of a *finally* statement or exit-node of method *M* or exit-node of another enclosing *try* statement, if there is no appropriate *catch* statement.

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. When a *try-catch-finally* nested inside another *finally*, *f*, is exited, if *label-var* represents a statement in *f*, the top element of *St* is deleted and used for instantiating *label-var* or *excp-var* as the case may be. If the exit is due to an exception or jump to a statement not in *f*, *St* is popped but the current reason for exit overrides the previous reason.

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

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

Theorem 3 *Intraprocedural CTI for programs with only single-level types and exceptions with subtyping is* **PSPACE***-complete, while the interprocedural case without dynamic dispatch is* **PSPACE***-hard.*

Using a construction similar to the one given above, we can show the following[15]:

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

In this construction, 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 *CTI* 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 *CTI* 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 *CTI* 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.

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

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 *CTI* in the presence of exceptions

This algorithm is an extension of the *basic* algorithm described in section 3.

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

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

Here z and u are points-tos. 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 they are propagated. 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.

throw **statement:** In addition to the conditional-points-tos described previously, this algorithm uses another kind of conditional-points-tos, called **exceptional-conditional-points-to**s, which capture propagation due to exceptions. The conditional part of these points-tos consists of an exception type 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) 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 $\langle excp,\ \langle p,\ obj1\ \rangle,\ obj\ \rangle$ representing the exception is generated. Such data-flow elements are handled like exceptional-conditional-points-tos, 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.

In addition to propagating *reachable* (defined in section 3), this algorithm also propagates data-flow elements of the form ⟨*excp-type, reachable* ⟩. When ⟨*reachable* ⟩ reaches a *throw* statement, it is transformed into ⟨*excp-type, reachable* ⟩, where *excp-type* is a run-time type of the exception thrown, which is then propagated like other exceptional-conditional-points-tos.

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.

***exit-node* of a method:** At the exit-node of a method, a data-flow element of type 4,6 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-tos (type 4) are handled as before. However, a data-flow element of type 6 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 ⟨*reachable* ⟩ or ⟨*excp-type,reachable* ⟩ reaches the exit-node of one of the called methods. ⟨*excp-type,reachable* ⟩ at the exit-node of a method causes the conditional-points-tos representing values of local variables at a call site of this method to change to exceptional-conditional-points-tos with *excp-type* at the return-node of the call site. ⟨*reachable* ⟩ is propagated across a call (not contained in a *finally*) if and only if ⟨*reachable* ⟩ reaches the exit-node of one of the called methods.

***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 3, 6 or 7, the associated exception already represents this reason. A *label* is associated with data-flow elements of type 1 or 4, 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. ⟨*label, reachable* ⟩,
2. ⟨*excp-type, reachable* ⟩,
3. ⟨*label, z, u* ⟩,
4. ⟨*excp-type, z, u* ⟩,
5. ⟨*excp, z, obj* ⟩.

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 3.1. 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 3.1).

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-tos are expanded at an exit-node of a method. For exceptional-conditional-points-tos 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-tos 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. 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 a call) 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-tos, *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 Example

The following two examples illustrate the above 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 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 param ) {
      // ⟨empty, ⟨param, 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 method( A param ) throws excp_t1 {
    excp_t1 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 ⟩ ⟩
  }
};

class test {
  public static void test_method( ) throws excp_t1 {
    A local;

    l2: local = new A();

    try {
      base.method( 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 ⟩.
      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 ⟩.

        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( ) {
    test_method();
  }
};
```

6.3 Proof of precision

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 *A construct is called alpha-construct if and only if it is a method body, try, catch or finally.*

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 alpha-construct 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* [16] *in p.*

[16] Due to a *finally* clause, an uncaught exception generated in t may be active when control reaches a point 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 6. *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 7. *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 ⟩*[17] *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*

[17] z could be empty.

  - *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.*

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

Lemma 8. *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 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.*

proof: Lemma 7 (claims 3, 4 and 6) implies that the solution computed by the algorithm is a subset of the precise solution. While lemma 6 and lemma 8 (claims 4, 5 and 6) imply that the precise solution is a subset of the solution computed by the algorithm. Hence the theorem. □

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 $\mathrm{O}(n^6)$. We will need the following lemma:

Lemma 9. *Suppose ⟨⟨var1,obj1 ⟩, ⟨var2,obj2 ⟩⟩, ⟨label, ⟨var1,obj1 ⟩, ⟨var2,obj2 ⟩⟩, ⟨excp-type, ⟨var1,obj1 ⟩, ⟨var2,obj2 ⟩⟩ or* $\langle excp, \langle var1, obj1\rangle, obj2\rangle$ *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[18], 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-tos, this follows from lemma 9. This implies that if l is not contained in a *finally*, the solution set at l may contain up to $\mathrm{O}(V^2L)$ conditional-points-tos, $\mathrm{O}(XT \star V^2L)$ exceptional-conditional-points-tos, $\mathrm{O}(V \star XO)$ points-tos for exception objects and $\mathrm{O}(XT+1)$ 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 $\mathrm{O}(B \star V^2L)$ labelled-conditional-points-tos, $\mathrm{O}(XT \star V^2L)$ exceptional-conditional-points-tos,

[18] identified by their creation sites.

O($V \star XO$) points-tos 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-tos + maximum number of labelled-conditional-points-tos + maximum number of exceptional-conditional-points-tos + 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-tos at the entry-node of the called method. In addition O($C(B + XT + 1)V^2L$) work may be done in passing points-tos 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-tos at *r-node*, again we use lemma 9. 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-tos 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-tos. 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 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 this section, we prove that the algorithm presented in 6.1 computes a safe solution for programs written in *JavaWoThreads*. Lemma 10 is a weakened form of lemma 8. The main difference is that in lemma 10 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-tos in the set of points-tos that hold at the first node of p and together imply u.

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 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 6 *The extended algorithm computes a safe solution for programs written in a subset of Java defined in section 2.8.*

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 2.3). Further, when r is followed, suppose *VR* points-to *obj* at l. Lemma 6 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-tos, each of these points-tos holds at l when r is followed. Using lemma 6 and lemma 10, each of these points-tos is computed by the algorithm as exceptional-conditional-points-to with *excp-type* and some assumed-points-to, and each of these assumed-points-tos holds at the first node of p when r is followed. Thus, the conditional parts of these exceptional-conditional-points-tos 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-tos which hold at l when r is followed. Suppose l is not contained in a *finally*. Using lemma 6 and lemma 10, each of these points-tos 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-tos are compatible and this relationship will be reported by the algorithm. Next, suppose l is contained in a *finally*. Again, using lemma 6 and lemma 10, each of these points-tos 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-tos hold at the first node of p when r is followed. Thus these are compatible. Hence this relationship will be reported by the algorithm.

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-tos 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-tos valid at l. Each such points-to may be implied by up to D points-tos at the entry of the procedure containing l, this is due to general types and dynamically-dispatched calls as shown in appendix A. This implies that the solution set at l may contain up to $\mathrm{O}(D^2)$ conditional-points-tos. Similarly there could be up to $\mathrm{O}(XT{\star}D^2)$ exceptional-conditional-points-tos and up to $\mathrm{O}(D{\star}XO)$ data-flow facts representing exception objects. If l is contained in a *finally* statement, there could be up to BD^2 labelled conditional-points-tos. Suppose l is an assignment node. For each conditional-points-to/exceptional-conditional-points-to/labelled-conditional-points-to reaching l, up to $\mathrm{O}((XT + B + 1)D^2)$ (due to multiple field selectors) conditional-points-tos/exceptional-conditional-points-to/labelled-conditional-points-to may be generated at l and up to $\mathrm{O}(k^2 D^{2k} L^2)$ 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 $\mathrm{O}(D^k L)$ data-flow elements, as each field selector increases the number of assumed-points-tos by 1. Finally one assumed-points-to is selected from $2k$ (at most) assumed-points-tos associated with a data-flow element if they are compatible. Note that $\mathrm{O}(k^2)$ is the worst-case cost of checking pairwise compatibility of $\mathrm{O}(k)$ assumed-points-tos 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 $\mathrm{O}((XT + B + 1)k^2 D^{2(k+1)} L^2)$. 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 $\mathrm{O}((XT + B + 1)D^2)$.

The work done at call-nodes in passing data-flow elements to the entry-nodes of called methods is $\mathrm{O}(I(A+1)(B+XT+1)V^2 L)$ because each points-to at a call-node can imply at most $A + 1$ points-tos at the entry-node of the called method. In addition $\mathrm{O}(I(B + XT + 1)V^2 L)$ work may be done in passing points-tos 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 $\mathrm{O}(D)$ assumed-points-tos 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 $\mathrm{O}(D(XT + B))$ assumed-points-tos 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-tos. Similarly, for an edge leaving a *try-catch-finally* nested inside a *finally* up to $\mathrm{O}(D(XT + B))$

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 11. *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 11.

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 11 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.

8.1 Complexity of computing the precise solution in the general case

Using Bill Landi's result [Lan92b] about undecidability of may alias analysis, it can be shown that *CTI* in the general case is undecidable.

8.2 Optimizations

The estimate in lemma 11 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 a assumed-points-to can improve precision in many situations. Similarly, an assumed-points-to may be associated with *excp-type* in exceptional-conditional-points-tos. 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 Related work

As mentioned in the introduction, no previous algorithm for pointer analysis or *CTI* handles exceptions. This work takes state-of-the-art in pointer analysis one step further by handling exceptions. Our algorithm differs from other pointer analysis and *CTI* algorithms [EGH94, WL95, Ruf95, PC94, PS91, CBC93, MLR$^+$93] in the way it maintains context-sensitivity by associating assumed-points-tos with each data-flow element, rather than using some approximation of the call stack. This way of handling context-sensitivity enables us to obtain precise solution for polynomial-time solvable cases, and handle exceptions. This way of maintaining context is similar to Landi-Ryder's[LR92] method of storing context using reaching aliases, except that our algorithm uses points-tos rather than aliases. Our algorithm also differs from approaches like [PS91, Age95] in being intraprocedurally flow-sensitive.

10 Conclusion

In this paper, we have studied the complexity of *CTI* for a subset of Java, which includes exceptions. The complexity of *CTI* in the presence of exceptions has not been studied before. The following are the main contributions of this paper:

1. The first polynomial-time algorithm for *CTI* in the presence of exceptions that handles a robust subset of Java without threads and C^{++}.
2. A proof that *CTI* for programs with only single-level types and exceptions without subtyping, and without dynamic dispatch, is in **P** and can be solved in O(n^7) time.
3. A proof that interprocedural *CTI* for programs with only single-level types and exceptions with subtyping, and without dynamic dispatch, is **PSPACE**-hard, and the intraprocedural case is **PSPACE**-complete.
4. A proof that *CTI* for programs with only single-level types and dynamic dispatch, and without exceptions, is **PSPACE**-hard.
5. A proof that *CTI* for programs with only single-level types and without exceptions or dynamic dispatch can be solved in O(n^5) 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 *CTI* for programs with only single-level types and without exceptions or dynamic dispatch is in non-deterministic log-space and hence *NC*.

References

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

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

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

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

[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.

[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.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[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.

[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.

[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.

[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 Program-*

ming Systems, Languages and Applications (OOPSLA '94), pages 324–340, October 1994.

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

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

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

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

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

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

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

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

A Example for the *basic* algorithm

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

```
class A {};  class B {                  class C {
                public B field1;           public B field1;
             };                         };

class base {                       class derived extends base {
 public static A a;                 public void method( ) {
 public void method( ){                // overrides base::method
   l1: a = new A();                    l2: a = new A();

   // ⟨empty, ⟨a, object_l1 ⟩ ⟩         // ⟨empty, ⟨a, object_l2 ⟩ ⟩
   exit_node:                          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 ) {
    // 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( ) {                public void test2( ) {
     base p;                              derived p;
     l4: p = new base();                  l5: p = new derived();

     // ⟨empty, ⟨p, object_l4 ⟩ ⟩           l10: caller.call(p);
     l9: caller.call(p);

     // ⟨empty, ⟨p, object_l4 ⟩ ⟩ .         // ⟨empty, ⟨p, object_l5 ⟩ ⟩ .
     // At l9 ⟨p, object_l4 ⟩               // At l10 ⟨p, object_l5 ⟩ =>
     // => ⟨empty, ⟨a, object_l1 ⟩ ⟩.       // ⟨empty, ⟨a, object_l2 ⟩ ⟩.
     exit_node:                           exit_node:
  };                                    };

  public void test3( ) {                public void test4( ) {
     C q;                                 C q;
     l6: q = new C();                     l7: q = new C();
     potpouri.example( q );               potpouri.example( q );
  };                                    };
};
```

B Auxiliary functions

// 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 an 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-tos 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-tos 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
```

```
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-tos
  // 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-tos which
// imply this point-to at the call site. This function stores
// condition with s in this table. }
```

```
set of points-tos 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-tos for the values
        of receiver, which result in a call to method M;
      else
        return a set containing empty;
    }
  }
  else {
    return the assumed-points-tos stored with s in C.asPtTtable;
    // asPtTtable is defined in add_to_table_of_assumed_points_tos.
  }
}
```

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

```
set of conditional-points-tos get_implied_conditional_points_tos( rCPT, M, C ) {
  // It returns conditional-points-tos 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-tos 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
    }
  }
}
```

C Examples for intra- and interprocedural flow-sensitivity

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

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

      ...

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

Fig. 5. Intraprocedural flow-sensitivity

This article was processed using the LaTeX macro package with LLNCS style

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

    static void method2( ) {
      B p,q;

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

      // solution computed by an interprocedurally flow-sensitive algorithm:
      // ⟨q, object_l1 ⟩. object_l1 represents objects created at program point l1.

      // solution computed by an interprocedurally flow-insensitive algorithm:
      // ⟨q, object_l1 ⟩, ⟨q, object_l2 ⟩

      // Note: ⟨q, object_l2 ⟩results from the unrealizable path:
      // method3 $\xrightarrow{calls}$ method1 $\xrightarrow{returns-to}$ method2
    }

    static void method3( ) {
      B p,q;

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

      // solution computed by an interprocedurally flow-sensitive algorithm:
      // ⟨q, object_l2 ⟩

      // solution computed by an interprocedurally flow-insensitive algorithm:
      // ⟨q, object_l1 ⟩, ⟨q, object_l2 ⟩
    }
}
```

Fig. 6. Interprocedural flow-sensitivity or context-sensitivity