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, Fax: 732 445 0537, {ramkrish,ryder}@cs.rutgers.edu

[2] Siemens Corporate Research Inc, 755 College Rd. East, Princeton, NJ 08540 USA, wlandi@scr.siemens.com

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 intraprocedural concrete type-inference problem with single-level types and exceptions with subtyping is **PSPACE-complete**, while the interprocedural problem without dynamic dispatch is **PSPACE-hard.**

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.

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. 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.

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 discuss our results about complexity of *CTI* in the absence of exceptions. Next,

[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 4	x	x		
intraprocedural CTI **PSPACE**-complete	sec 4	x		x	
interprocedural CTI **PSPACE**-hard	sec 4	x		x	
interprocedural CTI **PSPACE**-hard	sec 3	x			x
interprocedural CTI in **P**, $O(n^5)$	sec 3	x			
intraprocedural CTI in **NC**	sec 3	x			

Table 1. Complexity results for *CTI* summarized

we extend the *basic* algorithm for *CTI* in the presence of exceptions, and discuss the complexity and correctness of the extended algorithm. Finally, we present **PSPACE**-hardness results about *CTI* in the presence of exceptions. Due to lack of space, we have omitted all proofs. These proofs and further details about the results in this paper are given in [CRL97][5].

2 Basic definitions

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.

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.

Precise solution for *CTI.* A *reference variable* is one of the following:

– a static variable (class variable) of reference[6] type;

[5] available at http://www.prolangs.rutgers.edu/refs/docs/tr341.ps.
[6] may refer to an instance of a class or an array.

- 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 execution 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 paths in a program are not necessarily executable and determining which are executable is undecidable. Barth[Bar78] defined *precise up to symbolic execution* to be the precise solution under the assumption that all 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.*

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, of an object created at a program point *n*; and *obj* is *object_s* - an object created at a program point *s*.

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.

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.

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.

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 4, we will extend this algorithm to handle exceptions.

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 ⟨*condition, points-to*⟩, where *condition* is an assumed-points-to or *empty* (meaning this points-to is applicable to all contexts). For simplicity, we will write ⟨*empty,points-to*⟩ 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.*

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 ⟨*empty*, ⟨*p, obj1*⟩⟩, ⟨z, ⟨*obj1.f1, obj2*⟩⟩ and ⟨u, ⟨*obj1.f1, obj3*⟩⟩, then the possible values of V are *obj2* and *obj3*.

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 ⟨*z*, ⟨*p, object_s*⟩⟩ and ⟨*u*, ⟨*q, object_n*⟩⟩ is present in the solution computed at l so far. Assuming z and u are compatible, *apply* generates ⟨*object_s.f1, object_n*⟩ 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 ⟨*u*, ⟨*object_s.f1, object_n*⟩⟩ 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 ⟨*z, u*⟩ 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 ⟨*x, t*⟩ is in the solution set at C and t implies z at the entry-node of M, ⟨*x, u*⟩ 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 called from this site. Due to dynamic dispatch, the set of methods invoked from a call site is iteratively computed during the data-flow analysis as in [EGH94]. Suppose ⟨*x, t*⟩ holds at a call site C which has a method M in its set of invocable 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), ⟨*z,z*⟩ 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.

Precision of the *basic* algorithm. By induction on the number of iterations needed to compute a data-flow element and the length of a path associated with a data-flow element, in [CRL97], we prove that the *basic* algorithm computes the

```
// 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

precise solution for programs with only single-level types and without dynamic dispatch, exceptions or threads. For programs of this form, *CTI* is distributive and a conditional-points-to at a program point can never require the simultaneous occurrence of multiple conditional-points-tos at (any of) its predecessors. Intuitively this is why the above proof works. The presence of general types, dynamically dispatched calls or exceptions with subtyping violate this condition and hence *CTI* is not polynomial-time solvable in the presence of these constructs. We also prove that the *basic* algorithm computes a safe solution for programs written in *JavaWoThreads*, but without exceptions.

Complexity of the *basic* algorithm. The complexity of the *basic* algorithm for programs with only single-level types and without dynamic dispatch, exceptions or threads is $O(n^5)$, where n is approximately the number of statements in the input program. This an improvement over the $O(n^7)$ worst-case bound achievable by applying previous approaches of [RHS95] and [LR91] to this case. Note that $O(n^3)$ is a trivial worst-case lower bound for obtaining a precise solution for this case. For programs written in *JavaWoThreads*, but without exceptions, the *basic* algorithm is polynomial-time.

Other results on the complexity of *CTI* in the absence of exceptions. In [CRL97], we prove the following two theorems:

Theorem 1 *Intraprocedural CTI for programs with only single-level types is in non-deterministic log-space and hence* **NC**.

```
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

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.

Theorem 2
CTI for programs with only single-level types and dynamic dispatch is **PSPACE**-*hard.*

4 Algorithm for *CTI* in the presence of exceptions

In this section we extend the *basic* algorithm for *CTI* of *JavaWoThreads*, and discuss the complexity and precision of this extended algorithm.

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

```
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

1. ⟨*reachable*⟩,
2. ⟨*label, reachable*⟩,
3. ⟨*excp-type, reachable*⟩,
4. ⟨*z, u*⟩,
5. ⟨*label, z, u*⟩,
6. ⟨*excp-type, z, u*⟩,
7. ⟨*excp, z, obj*⟩.

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. Further details are given in [CRL97]. 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-tos*, 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 ⟨⟨*q, obj1*⟩, ⟨*p, obj2*⟩⟩ 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

```
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

propagated to the exit-node of the corresponding *try* statement, if there is one. A precalculated *catch-table* 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 so, 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 itself. 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) 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.

finally statement: 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. In order to correctly handle this involved semantics, 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 (enumerated above), 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*. Thus 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. 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).

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 mechanism similar to the one used for handling a call is used for handling a *try* statement nested inside a *finally* because it can cause labels and exceptions to stack up. Details of this are given in [CRL97].

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 the *finally* is associated with the data-flow element.

Example. The example in Figure 5 illustrates the above algorithm.

Precision of the extended algorithm. In [CRL97], we prove that the extended algorithm described in Section 4 computes the precise solution for programs with only single-level types, exceptions without subtyping, and without dynamic dispatch. We also prove that this algorithm computes a safe solution for programs written in *JavaWoThreads*.

Complexity of the extended algorithm. The the worst-case complexity of the extended algorithm for programs with only single-level types, exceptions without subtyping, and without dynamic dispatch is $O(n^7)$. Since we have proved that the algorithm is precise for this case, this shows that this case is in *P*. If we disallow *try*s nested inside a *finally*, the worst-case complexity is $O(n^6)$. For general programs written in *JavaWoThreads*, the extended algorithm is polynomial-time.

Complexity due to exceptions with subtyping. In [CRL97], we prove 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 (even) without dynamic dispatch is* **PSPACE**-*hard.*

```
// 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:
  }
};
```

Fig. 5. *CTI* in the presence of exceptions

Theorems 2 and 3 show that in the presence of exceptions, among all the reasonable special cases that we have considered, programs with only single-level types, exceptions without subtyping, and without dynamic dispatch comprise the only natural special case that is in **P**. Note that just adding subtyping for exception types and allowing overloaded *catch* clauses increase complexity from *P* to **PSPACE**-hard.

5 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.

6 Conclusion

In this paper, we have studied the complexity *CTI* for a subset of Java, which includes exceptions. To the best of our knowledge, the complexity of *CTI* in the presence of exceptions has not been studied before. The following are the main contributions of this work (proofs are not presented in this paper, but appear in [CRL97]):

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

Additional contributions are:

1. A proof that *CTI* for programs with only single-level types, dynamic dispatch, and without exceptions is **PSPACE**-hard.
2. A proof that *CTI* for programs with only single-level types can be done in $O(n^5)$ time. This is an improvement over the $O(n^7)$ worst-case bound achievable by applying previous approaches of [RHS95] and [LR91] to this case.
3. A proof that intraprocedural *CTI* for programs with only single-level types 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.

[CRL97] Ramkrishna Chatterjee, Barbara Ryder, and William Landi. Complexity of concrete type-inference in the presence of exceptions. Technical Report DCS-TR-341, Dept of CS, Rutgers University, September 1997.

[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.

[KU76] J.B. Kam and J.D. Ullman. Global data flow analysis and iterative algorithms. *Journal of ACM*, 23(1):158–171, 1976.

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

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

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

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

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

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

[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
    }
  }
}
```