

Assessing the Cost-Benefits of Using Type Inference Algorithms to Improve the Representation of Exceptional Control Flow in Java

Alex Kinner and Gregg Rothermel
Department of Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, Nebraska 68588-0115, USA
{akinneer, grother}@cse.unl.edu

May 15, 2005

Abstract

Accurate representations of program control flow are important to the soundness and efficiency of program analysis and testing techniques. The Java programming language has introduced structured exception handling features that complicate the task of constructing safe and precise representations of the possible control flow in Java programs. Prior work has considered applying various type inference algorithms to exceptions, but has not yet investigated whether the use of higher cost algorithms is necessarily justified. It is important to understand and assess the tradeoffs associated with the use of more powerful yet costly algorithms, thus we conducted an empirical study to evaluate the relative performance of several such algorithms. We find that applying type inference to exceptions may improve representations of control flow, but that these improvements do not necessarily translate into benefits for practical techniques that use them. Thus we argue that type inference should not be blindly applied, but rather the tradeoffs of applying them must be assessed with respect to particular program analyses and techniques.

Keywords: program analysis, experimentation

1 Introduction

Construction of accurate representations of control flow in Java software systems is important to a variety of program analysis and testing activities, such as construction of system dependence graphs [7, 11, 29], test case [5] and criteria [15] generation, regression test selection [21], checking of security properties [4], and even just accurate visualizations of control flow for program understanding. The safety of a control flow graph is the degree to which it properly accounts for all possible paths of execution through the program, a property that can determine the soundness of the results of many analysis techniques. A related notion is the precision of the graph, or how accurately it identifies paths as uniquely separable from each other, a property that often has a significant impact on the efficiency of analysis techniques that depend on the control flow representation.

The Java language includes facilities for structured exception handling, including mandatory checked exceptions (those for which the programmer must explicitly account). Such exception handling constructs are frequently present in Java programs [23]. In particular, exceptional control flow in Java introduces the potential for non-local transfers and type dependent transfers, including via dynamic polymorphic binding of exceptions to handlers by type. Therefore it is clear that the exception handling features of the language introduce new challenges in the construction of safe and precise representations of program control flow.

Prior work on analysis of exception handling constructs in Java programs [24] has focused on devising correct methods for modeling the effects of such constructs on the control flow of those programs. Such work has generally been more concerned with creating safe representations of exceptional control flow — in which no potential paths are missed — than with determining which paths are actually feasible during program execution (precise representations). The application of type inference to exceptions thrown in Java programs has been suggested in [24] as a means of addressing the problem of precision in control flow models of exceptional constructs. However, previous work has not measured or evaluated tradeoffs associated with type inference with respect to the resulting improvements in control flow graph precision.¹ In particular, the application of type inference has not been evaluated with respect to any consumers (e.g. client analysis techniques) of the control flow data obtained through static analysis.

In addition to issues of precision, there are questions about the *cost* of applying type inference algorithms. Because type inference has not been evaluated specifically in the context of exceptional control flow, there is little data available on the costs associated with the algorithms in that context. Most significantly, there is no data available regarding the cost tradeoffs attendant to using more powerful type inference techniques in conjunction with client analysis techniques that use the resulting control flow graphs.

In general, we expect that more powerful type inference techniques will produce more precise results at a higher cost. While data about the precision and costs of these techniques is important, we are also particularly interested in determining to what extent the differences in type inference techniques impact the client techniques they are intended to assist. More powerful type inference techniques are justified only if the additional precision yields benefits to client techniques that provide savings greater than the additional costs associated with the more powerful techniques, an issue that has not been previously studied.

Therefore in this study we empirically evaluate four type inference approaches (algorithms and techniques) for exceptions modeled on those described in [24], to assess the benefits of safer and more precise representations of exceptional control flow in Java programs against the costs of obtaining those representations. To assess the tradeoffs, we rely on two approaches. First, we define and collect a non-client-dependent metric for assessing the relative accuracy of the control flow graphs produced by the different approaches. Second, we investigate the impact of the improved control flow graphs when applied to the real world problem of regression test selection, compared with the costs associated with the increasingly more costly approaches for determining exceptional control flow.

The results of our study show that there are potential benefits to the application of type inference on exceptions, indicated by our metric, but these benefits do not extend to the client analysis problem of regression test selection.

2 Background and Related Work

2.1 Exceptions in Java

The Java language provides features for signaling exceptional conditions and implementing handlers to deal with those exceptions. All exceptions are first-class objects that inherit from a single base class (`java.lang.Throwable`). The Java Language Specification (JLS) [10] creates additional classifications for exceptions by identifying the subclasses `java.lang.Error` and `java.lang.RuntimeException` of `Throwable` as special. Exceptions that extend

¹Work has been done to investigate points-to analyses in Java, but these results have not been evaluated specifically with respect to exceptions, an issue discussed further in Section 2.

from these classes are *unchecked* exceptions, whereas all other exceptions are *checked* exceptions. An exception is raised to signal an exceptional state by throwing an instance of an exception object using the `throw` keyword.

Regions of code may be guarded by a `try` block that may transfer control to an exception *handler* when a particular class of exception is raised in that region (the exception is said to be *bound* to the handler). Multiple exception handlers may be associated with a single `try` block, represented as consecutive `catch` blocks, each of which declares the class of exception that is caught and handled. The matching of exceptions to handlers considers subclass relationships. If an exception is thrown, and there is no handler for the specific class of exception thrown, but there is a handler for a superclass of that exception, the exception binds to the handler for the superclass. Thus exception handlers in Java are said to *subsume* subtypes. Nesting of exception handlers is also permitted.

An exception that does not bind to any handler in the current method is said to escape the method. A handler may also re-throw an exception, in which case the exception may bind to any enclosing handlers or escape the method. The JLS requires that all checked exceptions that escape a method be reported by the method in the `throws` clause of that method's declaration. A caller in this case must provide a handler or handlers for the thrown exceptions, or itself declare the exceptions in its `throws` clause. Thus when an exception is thrown, it propagates up the call stack until it binds to a matching handler, or causes program termination.

A region of code may also be guarded by a `finally` block, which is code that must be executed regardless of whether or not execution in the guarded region raises an exception. If an exception is raised, control flows immediately to the matching handler, which in turn transfers control to the `finally` block. A `finally` block may determine control flow (such as by a `return`), in which case it supercedes any control flow induced by the handler, or it may return to the handler upon completion. In the absence of an exception, equivalent control flow occurs subsequent to execution of the last instruction in the guarded region.

2.2 Related Work

In this section we discuss prior work done in the areas of representation of exceptions in Java control flow and the broader analysis problem of points-to analyses.

2.2.1 Exceptional Control Flow

Beyond work reported in [24], there has been quite a bit of work related to analysis of exceptional control flow in Java.

Woo et al. [33] propose an algorithm for alias analysis in Java based on reference set computations that accounts for exceptional constructs. This contribution is presented in the context of the more general question of alias analysis.

Chang and Jo [6] present a set-based analysis to estimate the propagation of exceptions based on an operational semantics for Java. Their work is more concerned with determining interprocedural propagation of exceptions. It does not focus on the question of type inference specifically for the purpose of improving precision of control flow representation, and their proposed algorithm in fact concedes a dependency on such type inference techniques.

Choi et al. [8] present a control flow representation called the Factored Control Flow Graph (FCFG). This representation addresses the problem of frequently occurring potentially exception throwing instructions (PEIs) in Java, including unchecked exceptions such as divide-by-zero and other errors that may be raised implicitly by the virtual machine. While the approaches to modeling control flow for exceptions that we investigate account for

such exceptions in a conservative manner, we are more concerned with control flow related to checked exceptions – in particular, control flow explicitly created and handled by the programmer. The question Choi et al. addresses concerns efficiency more than precision – a related but different question.

Jorgenson [13] investigates improvements to the representation and handling of exceptional constructs in the Soot framework [28] for analysis and transformation of Java class files. This work is particular to the Soot framework and is principally concerned with preserving the correctness of potential transformations. The problem of type inference of thrown exceptions is noted, but no type inference is actually implemented.

Significantly, the foregoing work focuses primarily on presenting approaches, and includes no substantive empirical evaluation of techniques with respect to client analysis problems, and only limited evaluation of the implications of applying type inference.

2.2.2 Points-To Analysis

Considerable attention has been given to the question of points-to analysis in Java [3, 16, 17, 18, 22, 27, 30, 31, 32]. However, such work has not been limited to the question of impacts to the accurate representation of exceptional control flow. It is thus difficult to draw empirical conclusions from this work about the cost-benefits specific to the use of type inference in determining exceptional control flow.

Streckenbach and Snelling [27] implemented adaptations of Anderson’s [1] and Steensgaards’s [26] algorithms for points-to analysis of C to work on Java and evaluated their performance. Strictly speaking, points-to analysis on references is a superset of type inference analysis on exceptions. As a consequence, [27] investigates a more general question that involves extra complexity and considerations unimportant to type inference at the narrower scope of exceptions. Thus we cannot draw conclusions specific to type inference on exceptions from their results.

Wang and Smith [30] describe constraint based type inference techniques for Java, but these too are techniques aimed at addressing the much more general question of type inference on all object references, evidenced by the use of their techniques to check the safety of downcast operations in Java programs. The techniques are not discussed in the context of constructing control flow representations or related questions specific to exceptional constructs.

3 Exception Type Inference Techniques

We wish to evaluate the potential benefits of three approaches, of increasing cost, for performing type inference on exceptions. We expect that the additional work performed by approaches of greater cost will produce control flow representations that are more precise. The approaches we investigate replicate, as closely as possible, techniques described in [24]. Here, we describe the three approaches and a simple baseline technique. All approaches were implemented by the first author, with common functionality shared or implemented equivalently in code among the approaches whenever possible.

Simple Baseline (base). This technique uses simple contextual information to generate a conservative estimate of the possible types of exceptions at each throw point. The contextual information used is the catch types associated with enclosing exception handlers, types declared as thrown by the current method, and in the case of calls, types declared as thrown by the called method. This technique seeks only to preserve the property of safety with respect to proper computation of control flow; that is, no program behavior can cause control flow which cannot be correlated to paths in the control flow graphs constructed for that program.

Flow-sensitive Intraprocedural (FS-intra). This algorithm performs a flow-sensitive backwards search from the point of each throw to find all of the reaching exception object instantiations, if possible. The search stops when the beginning of the method is reached. FS-intra also terminates the search when a method call is reached, unless it is searching for the creation of an exception assigned to a local variable, as non-local variables may be assigned by the called method. The result of the search is classified as precise if object instantiations can be found on all reaching paths. FS-intra makes conservative estimates to determine possible exceptional control flow out of calls.

Flow-insensitive Interprocedural (FI-inter). This algorithm performs a flow-insensitive search for all exception objects that may be instantiated by the current call or its callees. For call instructions, the algorithm takes the union of the types reaching exceptional exit nodes in the graphs for all possible bindings of the method. The types inferred at a throw point are then the subset of the found types that are subclasses of those types declared as thrown by the method or caught by enclosing exception handlers. In the case of `throw` instructions, the inferred control flow varies only with enclosing exception handlers. Inferred control flow for calls varies depending both on the implementations that may be bound to the call and the enclosing handlers.

Combined Intraprocedural and Interprocedural (cmb). This technique applies the flow-sensitive algorithm first, then executes the flow-insensitive algorithm on blocks for which the results of flow-sensitive analysis were imprecise.

3.1 Baseline

The basic control flow computation, which represents the application of no special type inference algorithm, computes a general estimate of the possible type or types of exception using only that information that is immediately available at no cost. This provides a baseline against which the other algorithms can be compared, thus serving as the control in the experiment described subsequently. The simple algorithm works as follows to compute exceptional control flow:

1. Exceptions declared as thrown by the method under analysis are used as a conservative upper bound on the types of checked exceptions that may be thrown from any point in the method.
2. At each throw instruction, the algorithm attempts to determine if the exception object being thrown is created at the throw site. This can be accomplished at the bytecode level by checking whether the immediately preceding instruction is an invocation of a constructor, and captures a common usage of the type `throw new Exception()`.
3. If the exception is created at the throw site, the type of the object on which the constructor was invoked is captured as the precise type of the exception thrown at that location. Otherwise the algorithm identifies all enclosing exception handlers and takes the union of the types of the exceptions caught by those handlers and the types determined by Step 1. This combined set represents the conservative estimate of the types of exceptions that may be thrown.
4. At each call instruction, the algorithm takes the union of the types determined in Step 1, the types caught by all enclosing exception handlers, and the types declared as thrown by the called method. The resulting set represents the conservative estimate of the types of exceptions that may be thrown out of the called method.
5. Edges are created from each throw or call node to matching handler nodes or exceptional exits, labeled with the class of exception that causes the corresponding control flow.

A significant weakness of the simple algorithm is that it often will fail to account for unchecked exceptions. Since it uses contextual clues to provide estimates of the possible types of exceptions thrown, it can account for unchecked exceptions only if they are created at the throw site, or if they are explicitly handled or declared in the throws clauses of method signatures. The simple algorithm will also often provide only upper bounds on the types of exceptions, whereas we expect that more sophisticated analyses in many cases should be able to identify specific exception subclasses that may actually be thrown. However, a considerable proportion of exceptions are immediately created at the throw site, which makes evaluation of other techniques against this baseline particularly interesting.

3.2 Flow-Sensitive Intraprocedural

The second algorithm we implemented is a flow-sensitive intraprocedural analysis. In this analysis, we walk the control flow graph for the method in reverse from each point at which an exception is thrown and attempt to locate all of the points at which an exception object is created that could eventually reach the throw point. In the most trivial case the exception will be created at the throw point, in which case a single precise result is obtained as with the simple algorithm. Otherwise, the search is implemented as an exhaustive depth-first search on the reverse of the control flow graph starting from the node representing the exception throwing block.

A simulation of the Java bytecode execution stack is maintained that can abstractly compute the reverse result of executing an instruction. It accomplishes this by determining whether an instruction produces, consumes, or has no effect on the number of operands on the stack, and then adjusts the number of operands on the simulated stack by inverting the action (e.g. an instruction that produces an operand on the stack in normal forward execution causes an operand to be removed from the stack in the reverse execution simulation). Another value is also maintained that indicates the number of remaining operand producing instructions required to produce the object that will be at the top of the stack at the throw point on a given reverse flow path. This value is modified by, and required to deal with, certain not strictly stack instructions such as SWAP and DUP_Xn class instructions.

From an initial state in which an operand producing instruction creates the object at the top of the stack, which in a well-formed class file is the exception to be thrown, the flow-sensitive algorithm walks the control flow in reverse, feeding corresponding instructions into the reverse stack simulation. If the stack simulation reaches a point at which an operand producing instruction creates the object at the top of the stack at the throw point, the instruction is analyzed to determine if it can produce an exception of only one type. If this is the case, the result along that path is considered precise and the search on that path terminates. When a branch is reached in the reverse control flow (which corresponds to a control flow join in the normal forward execution), each path is searched using a separate copy of the simulated stack. Thus when a search terminates along a given path, the search backtracks to the most recent branch and resumes the search along another path. If no such branch exists during backtracking, the flow-sensitive search is complete for that throw point.

The flow sensitive search along any given path is also constrained by certain early stop conditions. If it reaches the beginning of an exception handler, the search terminates and returns the declared class of exception caught by the handler as the conservative and imprecise estimate on that path. This approach is taken because a precise inference would require a determination of all the actual classes of exceptions possibly thrown within the associated exception handler, minus those that may be caught by any more precisely binding exception handlers also active at the throw points. We also take the heuristic view that a majority of exception handlers are associated with

method calls, which means that a precise inference will likely not be possible in any case.² The search along a given path also terminates if the beginning of the method is reached (an error that in reality corresponds to an illegal class file), if the exception is produced as the return value of a method call, or if one of a certain set of instructions is encountered as described in the following.

Some instructions may produce the exception being thrown, but do not themselves precisely constrain the possible types of the thrown exception. The following is a list of these types of instructions, and the action taken by the flow sensitive search along a path when it reaches such an instruction of that type:

- **GETFIELD** (Get object reference from instance variable). The search switches into a mode in which it looks for corresponding **PUTFIELD** instructions. For each such **PUTFIELD** instruction, it then searches for the instruction producing the stored value. Note that if a method call is encountered while performing this search, the search terminates along that path, as it is no longer possible to determine the possible assignments to that field without an interprocedural analysis.
- **GETSTATIC** (Get object reference from class variable). The search behaves in the same manner as described for **GETFIELD**, where it searches for corresponding **PUTSTATIC** instructions instead.
- **ALOAD** (Get object reference from local variable). The search behaves in the same manner as described for **GETFIELD**, where it searches for corresponding **ASTORE** instructions instead, with the exception that method calls do not cause the search to terminate.³ Note that search mode switches may occur multiple times along a given search path, as a result of this or any of the object reference retrieval instructions so far addressed, as long as no early stop conditions are encountered.
- **AALOAD** (Get object reference from array). The search terminates and returns the declared type of the array as the conservative and imprecise estimate along that path. A more precise result would require a determination of all of the possible array indices that may be valid at the instruction, and corresponding searches for all assignments to those elements of the array, an analysis that is far too costly.

When an early stop condition or an instruction for which a precise estimate is not obtainable is reached, the type inference for the throwing instruction is considered incomplete. A conservative estimate is reported instead, calculated as the first common superclass of the types inferred or estimated along all control flow paths that reach the throw point.

There are two of additional special conditions that potentially modify the results reported by the flow-sensitive search. First, if the search along a path determines that it is possible that a null reference may be thrown, the `java.lang.NullPointerException` class is reported as the precise result for that search path. This behavior is justified by the observation that the declared behavior of the Java language is to throw that class of exception under that circumstance, and that the externally visible behavior is indistinguishable regardless of whether the `NullPointerException` is created explicitly in the user code or implicitly as a result of the (almost certainly unintentional) throwing of a null reference. Second, if the search along a given path determines that a cast operation post-dominates all

²An explicit throw to a handler in most cases represents the use of an exception for non-exceptional control flow, a bad practice. Thus in practice we expect to encounter this idiom rarely and correspondingly do not expect to sacrifice much in the way of significant gains in precision as a result of avoiding the extra analysis.

³Local variables cannot be altered outside the scope of the current method, so precision cannot be lost under this circumstance. If the local variable is found to be assigned from a non-local variable, stop conditions will still apply to the type inference applied to determine the possible types resulting from the assignment. (This behavior is required to achieve the results described in [24]).

possible productions of the thrown exception on that path, the conservative type estimate on that path is reduced to the class of the cast. Note however that this narrowing of the conservative estimate is only relevant if the search along that path ultimately proves to be imprecise.

In the intraprocedural control flow graphs we construct, method calls are also treated as potential exception throwing points. Because our flow sensitive analysis is constrained to operate intraprocedurally, it is limited to producing conservative estimates of the possible thrown exceptions at method calls. Thus at method calls, the flow-sensitive analysis produces an estimate in the same manner as the simple type inference described previously and classifies the result as imprecise. This may be reasonably viewed as a significant weakness of the flow-sensitive algorithm, however we postulate that it may perform excellently as a complement to the interprocedural analysis when the two are combined, an aspect that is tested by our fourth technique, which will be described later.

A final crucial aspect of the flow sensitive analysis that must be considered is the possibility that new paths introduced in the graph as the result of type inference on one exception throwing instruction may extend the flow-sensitive search paths for other throwing instructions and thus may introduce new results at other nodes in the graph. Such a situation can even potentially create a feedback cycle wherein new edges introduced at another node as a result of the type inference at the current node can then introduce additional new paths at the current node. To account for this scenario, our flow-sensitive algorithm will actually repeatedly iterate over all of the throwing nodes in the graph until the type inference results for every node are reported as precise, or the structure of the graph reaches a fixed point where no new edges are added.

3.3 Flow-Insensitive Interprocedural

The next algorithm we implemented is a flow-insensitive interprocedural analysis. To implement an interprocedural analysis, we first require a representation of the class dependencies in the system to enable the algorithm to determine the possible bindings for virtual method calls, or a safe superset thereof. For this purpose, we chose to use an Interclass Relation Graph, as described in [19], which will now be described in additional detail including some minor modifications that were necessary.

An Interclass Relation Graph (IRG) is a simple undirected graph to model the class inheritance and interface implementation hierarchy of a set of classes comprising the program under analysis. Nodes represent particular classes or interfaces in the system, and edges represent the inheritance or implementation relationships between those classes and interfaces (edges are not explicitly represented in the data structure). Given a list of Java classes and interfaces constituting the program to be analyzed, the IRG is constructed as shown by Algorithm 1.

Analysis of a method proceeds by first determining all of the classes of exceptions instantiated by that method and recursively through any methods it calls. The process of recursively exploring all called methods causes the full type inference process to also be triggered for those methods, a process with some implications that will be described shortly. Whenever a new class of exception is added to the set of inferred exception types for a bound implementation, it is automatically reported to all of the methods in its list of callers, so that the result propagates back up the call chain. This propagation stops whenever a caller is already aware of the class of exception in its set of inferred types, thus preventing infinite propagation in recursive call chains. This automatic propagation of new results inferred in called methods is sufficient to deal with recursion when collecting information about instantiations of exception objects.

After determining all of the exception objects that can be created as a result of the call, the analysis then iterates

over the blocks corresponding to explicitly thrown exceptions. For each such throw block, a much abbreviated flow-sensitive search is performed to determine a conservative upper bound on the types of exception that can be thrown at that point. From the set of instantiated exceptions determined previously, the subset corresponding to subclasses of the conservative estimate is reported as the set of possible classes of exceptions thrown from that block. Note that any exception instantiations located by the previously described process are implicitly assumed to be escaping. This is a conservative approximation, and yields a performance tradeoff, but in practice the approximation appears to perform well.

The analysis next iterates over the blocks corresponding to calls to other methods to determine estimates of the

Algorithm 1 Construction of Interclass Relation Graph (IRG)

Require: List L of classes and interfaces in the program

```

1: Create empty set  $N$  for nodes
2: for all  $o \in L$  do
3:   if  $o$  is a class,  $C$  then
4:     if  $\exists n \in N$  for  $C$  then
5:        $n_c = n$ 
6:     else
7:       Create  $n_c$  for  $C$  and add to  $N$ 
8:     end if
9:     Assign  $superclass(C)$  to  $n_c$ 
10:    if  $\exists n \in N$  for  $superclass(C)$  then
11:       $n_s = n$ 
12:    else
13:      Create  $n_s$  for  $superclass(C)$ 
14:    end if
15:    Assign  $C$  to subclasses list of  $n_s$ 
16:    for all  $I \in interfaces(C)$  do
17:      if  $\exists n \in N$  for  $I$  then
18:         $n_i = n$ 
19:      else
20:        Create  $n_i$  for  $I$  and add to  $N$ 
21:      end if
22:      Assign  $C$  to implementors list of  $n_i$ 
23:    end for
24:  else if  $o$  is an interface,  $I$  then
25:    if  $\exists n \in N$  for  $I$  then
26:       $n_i = n$ 
27:    else
28:      Create  $n_i$  for  $I$  and add to  $N$ 
29:    end if
30:    for all  $I_s \in interfaces(I)$  do
31:      if  $\exists n \in N$  for  $I_s$  then
32:         $n_i = n$ 
33:      else
34:        Create  $n_i$  for  $I_s$  and add to  $N$ 
35:      end if
36:      Assign  $I$  to implementors list of  $n_i$ 
37:    end for
38:  end if
39: end for

```

types of exceptions that may be thrown from those calls. It first uses the IRG to determine the possible method implementations that may be bound to the call. For each of these methods, it retrieves a data structure that is used to record the types inferred as thrown by that method implementation, whether the inference for that method is yet considered complete, and a list of the methods whose calls may potentially bind to that implementation. If such a data structure does not yet exist for that method implementation, it is created. The algorithm then checks whether the results for each bound implementation are complete, and if so unifies the results for that implementation with the set of types inferred for the current method under analysis (the caller). Otherwise, the partial results are unified with the set of inferred types for the current method, the current method is added to the list of callers for that implementation, and analysis of the called method begins.

To provide complete handling of recursion, the flow insensitive analysis employs a special mechanism for remembering the state of the analysis on a method. Upon beginning analysis of a method, a data structure is created that records information about the progress of analysis on that method, and is placed in a global set that indicates all the methods for which analysis is still in progress. Whenever the analysis of the current method reaches a call to a method that is present in this global set, it retrieves the analysis progress data for the called method from the set, registers itself as a caller of that method, and then resumes analysis on the recursively called method. Finally, when the analysis of the current method is about to finish, it requests resumption of analysis on all methods in the global set to ensure that results from methods called subsequent to the call inducing the recursive cycle and results from multiple polymorphic bindings at a recursive call site are properly included. Combined with the automatic propagation of new inference results to callers, this provides complete handling of recursion in the call structure of the program under analysis.

3.4 Combined Intraprocedural and Interprocedural

This technique combines the flow-sensitive intraprocedural analysis with the flow-insensitive interprocedural analysis to attempt to leverage the strengths of both algorithms. The flow-sensitive analysis is applied first to each of the blocks corresponding to `throw` instructions. Then the flow-insensitive analysis is applied to the call blocks and those throw blocks for which the results of the flow-sensitive analysis were imprecise. Precise type inference results obtained by the flow-sensitive analysis are provided to the flow-insensitive analysis along with the complete set of blocks for which precise results were not obtained; this ensures that the interprocedural analysis can utilize those results as necessary (such as when propagating information about thrown exceptions to callers). The final type inference results thus unify precise types determined by flow-sensitive analysis with the most precise types inferable by the interprocedural analysis on references to non-locally assigned exceptions and exceptions thrown from calls.

4 Experiment Design and Results

To investigate the cost-benefit tradeoffs of applying various type inference techniques to the computation of exceptional control flow in Java software, we applied the techniques described in Section 3 to several Java programs and analyzed the results in two ways:

1. by considering a client-use-independent metric computed comparatively over the graphs generated by each algorithm over multiple versions of the programs;

Object	Versions	KLOC	No. of Classes	Tests	% MwH
ant	11	80.4	789	878	8.2
xml-security	9	16.3	207	84	18.7
jmeter	7	43.4	486	98	6.9
jtopas	4	5.4	63	209	9.8

Table 1: Experiment Objects

2. by evaluating the performance of the algorithms as they impacted regression test selection on JUnit test suites across the same versions of those programs.

In the remainder of this section we describe our objects of analysis, variables and measures, experiment setups, threats to validity, and results and analysis.

4.1 Objects of Analysis

We used four Java programs as objects of analysis: `ant`, `xml-security`, `jmeter`, and `jtopas`.⁴ `Ant` [2] is a build tool similar to `make`. `Xml-security` is a library that implements security standards defined for XML [34]. `Jmeter` is a desktop application for load testing and measuring performance of other Java software [12]. `Jtopas` is a simple library for text parsing [14]. A sequence of released versions are available for these programs.

Table 1 summarizes the characteristics of the most recent versions of each of these programs with respect to their overall size, percentage of methods with exception handling constructs (% MwH), and JUnit test suite sizes. Based on this information, and the data reported in [23], we argue that these programs are a reasonably representative sample of Java software being developed in practice in terms of size, use of exceptions, and construction of test suites. (As a point of reference, we collected similar data on the 27 most frequently downloaded Apache Jakarta and Sourceforge projects. We found a range of program sizes from 2.9 to 157.7 KLOC, with an average of 41.3 KLOC.)

Our experiment uses the four JUnit test suites that were supplied by the developers with the object programs. These test suites are thus real examples of how JUnit testing is being applied in practice, and presumably provide coverage of program behaviors that developers consider important based on internal knowledge of the object programs.

4.2 Variables and Measures

4.2.1 Independent Variables

Our independent variable is the type inference algorithm applied during the computation of exceptional control flow. The choice of type inference algorithm should influence the cost of analysis, and may or may not have an impact on the benefits derived from the analyses based upon the results.

The baseline technique, as described in Section 3.1, served as the control for our experiment. This lets us compare the relative benefits of the more costly type inference algorithms based on differences in results across all measures against a baseline, low-cost conservative technique.

⁴These objects are drawn from the SIR repository [9], supplemented with additional program versions available from the authors.

4.2.2 Dependent Variables and Measures

We chose two classes of dependent variables and measures to evaluate. The first class includes client-independent measures related strictly to technique performance and the quality of the resulting exceptional control flow. The second class includes measures related to our client consumer of the control flow graphs, regression test selection. The first class of measures helps us understand the general performance of the algorithms, in a manner that provides initial guidance when deciding whether to apply type inference on exceptions to a given program analysis problem; however, it cannot provide any assessment of whether the use of an algorithm is justified in a particular practical application. The second class of measures does involve a client application: the application targeted in [24] where the techniques we consider are described.

Client-Independent Measures

Analysis Time. The first dependent variable we measure is the time required to construct control flow graphs for all of the methods in the object program, using each algorithm. This simple measure provides an understanding of the relative performance of the algorithms, and is useful for assessing time cost tradeoffs.

Graph Comparison Metric. To obtain a client-independent notion of the quality of the control flow graphs computed using a type inference algorithm (and thus of that algorithm) we required a client-independent metric. In this context, a useful metric must take into account both the feasibility of exceptional paths in a graph and the

Algorithm 2 CFG-assess

Require: Set E of edges computed only by algorithm A , Set E' of edges computed only by algorithm A'

```
1: return Metric score indicating improvement yielded by  $A'$  over  $A$ 
2: if  $e$  in  $E$  then
3:   if not superclass of any in  $E'$  then
4:     if maximal class in  $E$  then
5:       if subclass of any in  $E'$  then
6:         {anti-refinement; ignore}
7:       else if checked exception then
8:         score += 1 {Eliminated imprecise/infeasible edge}
9:       end if
10:    else if subclass of other edge  $e_2 \in E$  then
11:      if  $e_2$  scored point then
12:        score += 1 {Eliminated infeasible edge}
13:      else
14:        {transitive anti-refinement; ignore}
15:      end if
16:    end if
17:  else
18:    score += 1 {Eliminated imprecise/infeasible edge}
19:  end if
20: else
21:   if subclass of  $e_2$  in  $E$  then
22:     score += 1 {Refinement}
23:   else
24:     {anti-refinement; ignore.}
25:   end if
26: end if
```

type precision along those paths. A metric that simply counts edges will not suffice in this context, because a graph can have a larger or smaller number of more accurate edges than another graph and be of higher quality. A metric relating a graph to a baseline “optimal” graph would be ideal, however, it is not possible to compute an optimal graph for non-trivial systems such as those we consider. We thus instead constructed a metric allowing arbitrary pairs of graphs to be compared in a meaningful way. The metric is given in algorithmic form by Algorithm 2. First we provide the intuition for the metric, and then we present it in additional detail.

One type inference algorithm A can be thought of as better than another algorithm A' if, by eliminating infeasible edges and refining the precision of types on other edges or introducing new more precise feasible edges, the accuracy of the control flow graph A produces for a method is greater than that for the graph produced by A' . Additionally, the metric should not assume a priori knowledge about whether A or A' should perform better in a pairwise comparison (such as assuming the first algorithm in the comparison should be better), as this introduces a bias.

Consistent with the notion of graph quality just described, our metric is designed to award higher scores to graphs in which fewer exceptional edges represent infeasible paths and a greater number of exceptional edges encode exact or more precise types of exceptions associated with the control flow. In the control flow graphs that we compare, there is a direct correspondence between exception throwing nodes in each graph, and the only possible variance is in the number of outgoing edges. This makes it possible to compare the outgoing edges from corresponding nodes in two graphs. To do this we first remove from each set of outgoing edges any that represent the same exceptional control flow as an edge in the other set. We then apply Algorithm 2 (CFG-assess) to the remaining edges in the sets.

For each unique edge e in the set E produced by A , the metric applies the decisions described when the condition at line 2 is true. The exception associated with e is first compared against the edges in E' to determine if it is a superclass of any exceptions inferred by the A' . If that is the case, then CFG-assess awards a point because it indicates that A' eliminated that class of exception as a possibility, either because more precise subclasses could be determined, or it was infeasible entirely.

If e is not a superclass of any type inferred by A' , we now have reason to suspect that it is an infeasible edge that was eliminated or for which strictly more precise types were inferred, which would suggest A' performed better. Thus CFG-assess next tests whether the exception holds a maximal superclass relationship relative to any other exceptions inferred by the first algorithm. If this is also true, and the exception is not a subclass of any type inferred by the second algorithm, CFG-assess awards a point since we can conclude that A' eliminated the exception as infeasible. A point is not scored if the exception is a subclass of an exception inferred by the second algorithm, as this indicates a loss of precision.

In the case that the exception does not hold a maximal superclass relationship relative to the other exceptions, we test whether it is a subclass of any other exception inferred by A . If it is, CFG-assess awards a point based on whether the inference of the superclass was awarded a point. The reasoning here is that the judgment of the metric with respect to the superclass transitively applies to any subclasses. In other words, if the elimination of the superclass from the inferred types was a loss of precision, then the same applies to any subclasses.

After CFG-assess computes the score for the unique edges generated by A , it moves on to the unique edges produced by A' and applies to each such edge the decisions described if the condition at line 2 is false. This portion of the metric computation is simpler and should be intuitive to understand. CFG-assess simply determines whether

an exception inferred by A' is a subclass of any exception inferred by A . If this is the case, it awards a point as this is a simple refinement of precision. Otherwise, the type inferred by A' cannot be considered a superior result, so no score is awarded.

There are two important observations to be made about the computation of this metric. First, this metric is not commutative. No attempt is made to penalize an algorithm in the comparison, therefore reversing the order of comparison will not result in a negation of the metric value. The metric only assesses the improvement yielded by one type inference technique with respect to another. If the score awarded by the metric is zero, this implies only the absence of conclusive evidence that one of the algorithms performs better. This situation arises primarily from the difficulty that would be associated with attempting to assign an appropriate and meaningful penalty score where it is suspected that precision has been lost or infeasible edges introduced, an issue that is also compounded by issues related to unchecked exceptions.

The metric described here can only partially assess the impact of unchecked exceptions on the goodness of the resulting control flow graphs. This limitation derives from the fact that contextual information is typically used by the type inference algorithms to generate conservative estimates when more rigorous results cannot be achieved – contextual information which is frequently not available for unchecked exceptions. Because an unchecked declaration is by definition not required to be declared or handled explicitly in Java, constructs such as enclosing exception handlers may not be available for conservative estimates. As a consequence, many unchecked exceptions may be entirely invisible to less rigorous type inference algorithms, but may be revealed by more advanced algorithms. We do not expect this to be a significant concern, however, because we expect that this limitation will tend to cause the metric computation to underestimate the value added by a better performing algorithm. For example, a new unchecked exception inferred by algorithm A' may fail the test for a subclass relationship with an exception inferred by algorithm A where that would not be possible with checked exceptions. In this case, it is not possible to determine whether the inference of the exception class by algorithm A' is favorable without a priori knowledge about which algorithm is expected to be better.⁵

This metric should be considered only a heuristic for assessing the performance of type inference algorithms independent of specific target applications. However, the metric captures the notion of accuracy of exceptional edges in a reasonable way, allowing a client-independent assessment of algorithm (and graph) quality.

Client-Dependent Measures

Regression test selection is the problem of choosing which tests in a test suite should be re-run when a new version of a software system is to be verified for release [20]. One approach for selecting tests is to select those that exercise code changed since the software was last tested, which is the approach used by `DejaVu` [21]. `DejaVu` performs a simultaneous depth-first search on the control flow graphs for the old and new versions of a method to find blocks that have been changed. Traces of the execution of the test suite on the old version of the program are then used to select tests based on whether they traverse edges which reach changed blocks. Thus test selection based on this criteria requires control flow graphs to instrument and trace the program code, and to locate the differences between the versions.

The safety property of the underlying control flow graphs is crucial to the soundness of `DejaVu` with respect

⁵If A' is in fact better, then this inference is a favorable result. If, however, A is in fact the better algorithm, the appearance of the class of exception in the inferred set for A' would be the re-introduction of a type deemed infeasible by A , which of course is not beneficial.

to fault detection, and the precision property can be important to its efficiency. This is particularly relevant for exceptional control flow, as conservative estimates are safe but often highly imprecise. Therefore, to assess the impact of type inference in this client analysis, we implemented a version of `DeJaVu` and measured a dependent variable specific to this problem. The variable we chose is the number of tests selected using the graphs generated by each of the algorithms. This measure indicates whether the differences in exceptional control flow resulting from the various algorithms affect the test selection results on the given test suites.⁶

4.3 Experiment Setup

All of our implemented techniques were executed using v1.4.2 of the Java Runtime Environment (JRE) in a Linux environment. For timing consistency, all measurements for each particular object program were collected on the same system, though different objects may have been evaluated on different machines. Experimentation on `jtopas` was performed on a Pentium-III 800 Mhz system with 512 Mb RAM running SuSE Linux 9.1.⁷ Experimentation on the larger subjects was performed on a Pentium-M 1.6 Ghz machine with 1 Gb RAM running SuSE Linux 9.1 connected to an external power supply.⁸

We implemented our techniques within the Sofya analysis system [25], which provides utilities for instrumentation and control flow graph construction. We used shell scripts to automatically execute the control flow graph builder with the various type inference algorithms enabled across the versions of each program, and modified the main method of the CFG builder to report the total execution time, giving us accurate measurements of the time required for graph construction with each respective type inference algorithm or technique active.

4.4 Threats to Validity

Internal Validity. The principle threat to the internal validity of this experiment is possible faults in the implementation of the algorithms, and tools that we use to perform evaluation. We control for this potential threat through the use of extensive functional tests on our tools and verification against smaller Java programs and code fragments for which we can manually determine correct results.

Inconsistent decisions and practices in the implementation of the algorithms and techniques pose another threat to internal validity. For example, variation in the efficiency of implementations of common functionality between techniques could bias timing assessments. As noted in Section 3, we reduce this threat by having all our techniques implemented by the same developer, utilizing consistent implementation decisions and shared code.

External Validity. A threat to the generalizability of our results is the representativeness of our object programs. Other systems, including larger and more complex systems developed in industrial practice may exhibit different behaviors and cost-benefit tradeoffs. On the other hand, as noted in Section 4.1, the programs we investigate do reflect several characteristics of a popular set of open-source Java programs. A related concern involves sample size. Our experiment considers four programs, a number that may limit the validity of our conclusions. Investigation on additional programs will grant extra external validity to our results.

⁶Analogous to our first client-independent measure, we could also measure the total time required to apply the test selection technique on our object programs. If the use of a type inference algorithm causes `DeJaVu` to select fewer tests, the application of that algorithm is justified if the time to execute the larger test set minus the time to execute the smaller test set is greater than the time required to perform selection. We could then compare the resulting savings. In our study, however, as we shall show, the use of this metric would have added no value.

⁷The use of multiple systems enabled faster data collection, and does not bias our results since we do focus only on relative performances on particular objects.

⁸This guarantees that the CPU clock cycle will not be stepped down for power conservation.

Construct Validity. The client-independent metric we calculate may not be the only such metric suitable to evaluate the performance of the type inference algorithms. As we note, this metric may not account for the influence of unchecked exceptions in all circumstances. However we believe this threat is controlled for, for reasons outlined in Section 4.2.2. We also contend that the use of unchecked exceptions in a manner that is invisible to the metric is limited in practice. Specific handling of unchecked exceptions is visible to the metric, and corresponds to most interesting control flow related to such exceptions in practice.⁹

An additional construct threat is the decision to use real test suites provided with our object programs. Because we use these test suites to support the test selection evaluation of the performance of the algorithms, the quality and characteristics of the test suites can potentially influence the validity of the conclusions drawn from the test selection evaluation. The use of these test suites grants us considerable external validity but may bias our results. These concerns are further addressed in Section 5.

⁹Absence of explicit handling usually indicates a condition that will lead to the immediate termination of the program.

	base	FS-intra	FI-inter	cmb
xmlsecurity				
v0	561.8	580.3	768.9	821.7
v1	560.2	578.0	770.2	822.6
v2	569.2	587.1	785.2	836.1
v3	642.9	663.1	875.3	929.4
v4	653.5	673.7	890.1	946.8
v5	664.1	686.2	905.6	965.6
v6	669.8	691.0	914.7	974.5
v7	442.7	457.8	565.1	608.7
v8	440.1	456.0	563.0	604.8
jtopas				
v0	44.5	45.6	51.7	54.0
v1	48.3	49.4	55.4	58.0
v2	52.2	53.3	60.1	62.8
v3	186.4	190.3	222.0	229.9
ant				
v0	514.2	517.7	841.1	868.4
v1	834.2	842.6	1428.4	1533.5
v2	1700.3	1741.7	2957.5	3026.3
v3	1677.7	1760.8	2980.1	3164.9
v4	4371.9	4477.1	7842.4	8001.1
v5	4415.8	4452.3	7857.4	7951.2
v6	4521.4	4565.3	7900.0	8138.0
v7	4400.5	4428.6	8074.2	8200.7
v8	4471.2	4498.9	7995.4	8206.4
v9	7009.1	7065.6	12564.2	12842.5
v10	7015.1	7073.7	12770.6	12835.2
jmeter				
v0	1304.0	1299.7	1784.3	1843.3
v1	1260.5	1276.4	1749.7	1816.9
v2	1198.6	1200.1	1719.9	1781.5
v3	1820.1	1801.3	2659.6	2742.9
v4	1824.6	1836.0	2720.6	2792.6
v5	1945.2	1985.3	2865.7	2921.4
v6	1893.6	2047.1	2741.7	2824.7

Table 2: CFG construction times (seconds)

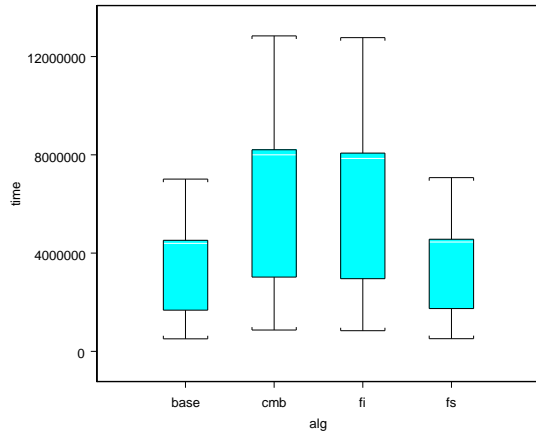
4.5 Results and Analysis

We now describe the results of our experiment; further discussion and interpretation of results occurs in Section 5.

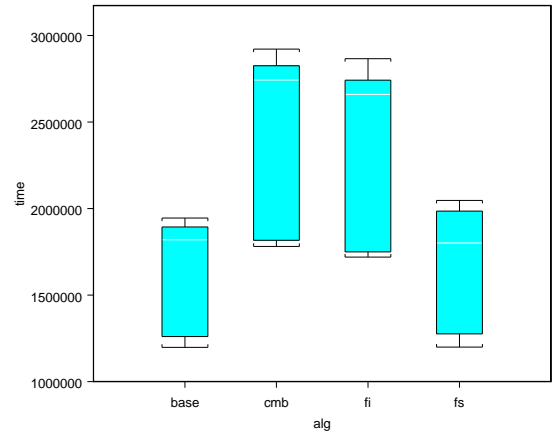
4.5.1 Client-Independent Measures

Analysis Time. Table 2 displays CFG construction times for the four algorithms, for each version of each object program. The data suggests that the algorithms have similar comparative performances across objects and versions. The basic algorithm is least expensive, but FS-intra is only slightly (typically only a few seconds) more expensive. The interprocedural algorithms are more expensive than basic and FS-intra, but their run times never exceed 1.85 times those of the simpler techniques.

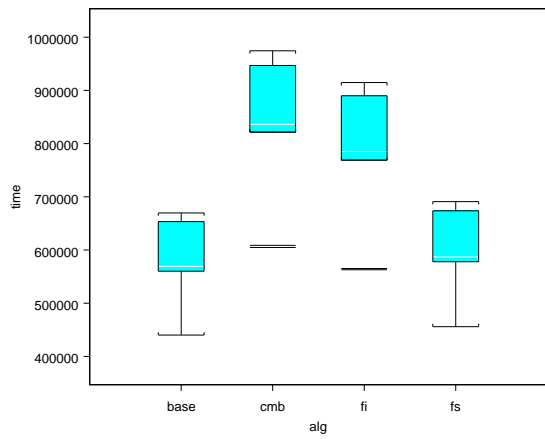
Boxplots of the timing results (Figure 1) show similar variance among techniques for each program. We performed per program ANOVAs on the CFG construction times, and, in cases where differences were observed, used the Bonferroni method for multiple comparisons between techniques to further assess the differences. Table 3 reports the results, with statistically significant pairings are indicated by “*****”. For `xml-security` and `jmeter`, the analysis reveals statistically significant differences between the performance of the different techniques. The



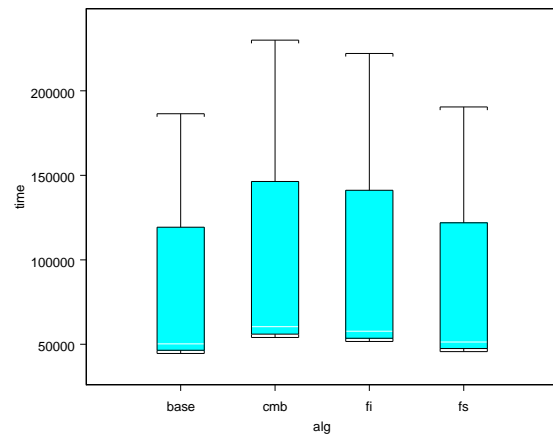
(a) ant



(b) jmeter



(c) xml-security



(d) jtopas

Figure 1: CFG Construction Time Boxplots

xml-security	Df	SS	MS	F-value	p-value
Technique	3	4.5e11	1.5e11	10.9136	0.00004
Residuals	32	4.4e11	1.4e10		
multiple comparison by Bonferroni method					
critical point: 2.8123					
	Estimate	Std. Err	Lower Bound	Upper Bound	
base-cmb	-2.56e5	55400	-4.12e5	-1.0e5	****
base-fi	-2.04e5	55400	-3.6e5	-48000	****
base-fs	-18800	55400	-1.75e5	1.37e5	
cmb-fi	52500	55400	-1.03e5	2.08e5	
cmb-fs	2.37e5	55400	81700	3.93e5	****
fi-fs	1.85e5	55400	29200	3.41e5	****
jtopas	Df	SS	MS	F-value	p-value
Technique	3	9.95e8	3.32e8	0.0552	0.9821
Residuals	12	7.22e10	6.01e9		
ant	Df	SS	MS	F-value	p-value
Technique	3	9.8e13	3.3e13	2.95	0.044
Residuals	40	4.43e14	1.11e13		
multiple comparison by Bonferroni method					
critical point: 2.7759					
	Estimate	Std. Err	Lower Bound	Upper Bound	
base-cmb	-3.08e6	1.42e6	-7.02e6	8.65e5	
base-fi	-2.93e6	1.42e6	-6.88e6	1.01e6	
base-fs	-4.48e4	1.42e6	-3.99e6	3.9e6	
cmb-fi	1.42e5	1.42e6	-3.8e6	4.08e6	
cmb-fs	3.03e6	1.42e6	-9.09e5	6.97e6	
fi-fs	2.89e6	1.42e6	-1.05e6	6.83e6	
jmeter	Df	SS	MS	F-value	p-value
Technique	3	3.79e12	1.26e12	6.1369	0.003
Residuals	24	4.93e12	2.06e11		
multiple comparison by Bonferroni method					
critical point: 2.8751					
	Estimate	Std. Err	Lower Bound	Upper Bound	
base-cmb	-7.82e5	2.42e5	-1.48e6	-85300	****
base-fi	-7.14e5	2.42e5	-1.41e6	-16500	****
base-fs	-28400	2.42e5	-7.26e5	6.69e5	
cmb-fi	68800	2.42e5	-6.28e5	7.66e5	
cmb-fs	7.54e5	2.42e5	56800	1.45e6	****
fi-fs	68.5e5	2.42e5	-12000	1.38e6	

Table 3: ANOVA on CFG Construction Times

p-value of 0.04 for `ant` suggests that there are statistically significant differences between techniques for that object as well, but the Bonferroni comparisons do not identify any specific differences between pairs of techniques. The results for `jtopas` did not indicate significance, we suspect due to lack of data.

Graph Comparison Metric. Table 4 reports the results obtained by applying our graph comparison metric to the pairs of graphs constructed by the various approaches, for each version of each object program. The values shown indicate the improvement yielded by the second approach with respect to the first in each comparison. The magnitudes of the values should not be compared between different object programs, or between different versions of a program, but rather to the values reported by other technique comparison combinations within the same version.

The data clearly indicates the extent to which the more advanced techniques can improve the accuracy of the reported control flow information. As expected, there is a progression in benefits from base to FS-intra to FI-inter, with (not surprisingly) the results from flow sensitive analysis falling between those of the basic and interprocedural

	base/ FS-intra	base/ FI-inter	base/ cmb	FS-intra/ FI-inter	FI-inter/ cmb
xmlsecurity					
v0	36	998	1001	960	0
v1	36	998	1001	960	0
v2	36	1074	1077	1036	0
v3	46	1887	1880	1854	34
v4	48	1978	1987	1944	36
v5	52	2047	2056	2015	0
v6	52	2047	2056	2015	0
v7	52	460	460	429	0
v8	50	458	458	429	0
jtopas					
v0	4	15	15	14	0
v1	4	15	15	14	0
v2	4	19	19	18	0
v3	90	241	241	167	0
ant					
v0	151	360	360	219	0
v1	218	499	499	293	0
v2	312	819	819	454	0
v3	310	805	805	450	0
v4	586	1400	1401	752	0
v5	592	1406	1407	754	0
v6	600	1410	1411	755	0
v7	600	1412	1413	755	0
v8	608	1478	1479	801	0
v9	722	2250	2257	917	3
v10	715	2255	2262	916	3
jmeter					
v0	110	560	560	469	0
v1	111	530	530	438	0
v2	117	546	546	448	0
v3	132	633	633	532	0
v4	129	640	640	508	0
v5	140	639	639	497	0
v6	140	633	632	491	0

Table 4: Graph comparison metric values

techniques. The interprocedural technique produces the greatest gains, even outperforming FS-intra by margins of a factor of 3 times or more. On most versions, however, there is no gain associated with the combined technique.

4.5.2 Client-dependent measures

Table 5 shows the number of tests selected by `DejaVu` when that technique is applied to the control flow graphs constructed using the four different type inference approaches, per version, per object program. For example, the entry for row `v1` for `jtopas` indicates that the changes in `jtopas` between versions `v0` and `v1` caused `DejaVu` to select 87 tests from the test suite, for each of the four type inference algorithms.

The data indicates that the application of type inference algorithms during construction of control flow graphs yielded little benefit, on our experiment objects, for test selection. Indeed, in nearly all cases the test selection results are identical to those obtained when using control flow graphs constructed using the basic technique. In only four cases — versions 4, 5, and 6 of `ant` and version 6 of `jmeter` — did we see changes in test selection results, and in these cases the difference involved only a single selected test. An ANOVA on the regression test selection results confirms this observation (p-value = 1, results not shown).

	no. of tests	base	FS-intra	FI-inter	cmb
xmlsecurity					
v1	104	0	0	0	0
v2	106	48	48	48	48
v3	92	88	88	88	88
v4	92	0	0	0	0
v5	94	55	55	55	55
v6	94	0	0	0	0
v7	84	78	78	78	78
v8	84	0	0	0	0
jtopas					
v1	126	87	87	87	87
v2	128	15	15	15	15
v3	209	44	44	44	44
ant					
v1	137	103	103	103	103
v2	219	121	121	121	121
v3	219	43	43	43	43
v4	521	189	189	189	189
v5	534	199	200	199	199
v6	557	410	410	411	411
v7	559	42	43	42	42
v8	559	518	518	518	518
v9	877	504	504	504	504
v10	878	349	349	349	349
jmeter					
v1	78	48	48	48	48
v2	81	73	73	73	73
v3	79	78	78	78	78
v4	79	49	49	49	49
v5	98	47	47	47	47
v6	98	0	0	1	0

Table 5: No. of Selected Tests

5 Discussion

Keeping in mind the threats to validity for this study, we now comment on the implications of our results.

Based on the minimal differences between the basic and FS-intra techniques in the timing data reported in Table 2, there is strong evidence that the application of FS-intra should be preferred over the basic technique, either by itself or in combination with other approaches. There is a greater cost associated with the interprocedural techniques, which suggests a more careful evaluation of the tradeoffs in using it is needed.

The results reported by the graph comparison metric serve as our first means of evaluating the tradeoffs. The graph comparison metric further confirms that the FS-intra technique is superior to the basic technique. Given the similar performance of the techniques, there simply is no significant penalty associated with the application of FS-intra, and thus there should be no reason to prefer the basic technique.

Further, as noted in the results, there is a considerable improvement associated with the application of FI-inter, which should not be surprising since the interprocedural algorithm benefits from the significant advantage of being able to refine the exceptional flow paths associated with calls. Given that exceptions are typically used to signal unexpected conditions to callers, there are considerably more opportunities for the interprocedural algorithm to improve the correctness of the control flow graphs in our aggregate intraprocedural graph model.

The metric results provide evidence that the application of more advanced type inference algorithms has the potential to yield considerable benefits to consumers of the resulting control flow graphs. The question of whether

a more costly algorithm is justified then may need to be evaluated on an individual basis against the cost savings obtained through the consumer of the improved control flow graphs. Thus we next look at the client analysis that we considered, regression test selection.

As the results in Table 5 show, the improvements reported by the metric did not translate into meaningful gains during the regression test selection process. Given such results, there clearly would be no advantage to incurring the additional cost associated with the more precise type inference algorithms, when performing this process. However, further analysis of the results also suggests some additional and complicating factors worth considering.

First, typical programming practices when dealing with exceptions are likely responsible in part for the observed results. In particular, the frequent practice of creating exceptions immediately at the throw site leads to uninteresting exceptional control flow that is unlikely to be impacted across progressive versions of a program. It is also plausible that code for handling exceptions is more stable, and thus less likely to be considered during difference-based test selection. Propagation of exceptions to high level handlers is an instantiation of this design strategy, and seems to be used relatively frequently in the object programs that we considered. Finally, the use of wrapped exceptions, especially subsequent to release 1.4 of the JDK when it was incorporated into the language design, adds complexity to the analysis of exceptional control flow. In particular, it seems entirely reasonable to surmise that the use of wrapped exceptions may result in considerably less local handling of exceptions, with a corresponding reduction in the number of code regions subject to change and test selection.

The nature of the `DejaVu` algorithm itself may be responsible for the results. Because test selection is based on differences between program versions, increased precision in exceptional control flow representation will translate into selected tests only if changes occur on exceptional flow paths. If exception handling code is more stable as considered above, then additional precision in the representation will not yield notable improvement for this particular control flow dependent application. The situation may be further complicated if changes on exceptional flow paths are masked by earlier differences on non-exceptional paths. `DejaVu` selects tests based on the first dangerous edge found on paths; it then does not require further information about changes occurring further along the path.

One concern that arises from the use of programs obtained from the field is that the test suites provided with those programs may not exhibit good coverage. An observation that might be drawn from the current experiment is that the test suites provided with the objects offer poor coverage of boundary and exceptional use cases within a program, which in turn may suggest a deficiency in test suite design. That said, our results do pertain to the actual test suites provided with the program, indicating results practitioners might expect in practice with such test suites.

Finally, it may also be the case that typical JUnit tests lack the scope to effectively probe exceptional flow paths within a program. Since individual JUnit test cases typically exercise a small region of code, the interactions most likely to result in exceptions may arise infrequently. Exceptional conditions may also be considered uninteresting to many JUnit test designers, since it is often the case that the response of other components to exceptions is the more interesting behavior, and unit testing is not as conducive to interaction testing. For this reason, functional tests may yield different results, an avenue for future investigation.

6 Conclusions and Future Work

We have performed a study of the tradeoffs related to the application of type inference algorithms in constructing representations of control flow in Java programs. This study found evidence that the use of type inference can create different control flow graphs and thus may potentially yield benefits for some program analysis and testing activities. However, this evidence comes from the computation of a metric assessing the overall quality of the control flow graphs constructed. Our study did not demonstrate a worthwhile cost-benefit tradeoff for the particular problem of regression test selection. It would seem that the cost associated with the type inference techniques is not justified for that particular client analysis.

More broadly, we conclude that the question of whether to utilize type inference to improve the representation of exceptional control flow must in future be more rigorously considered, by investigators developing analysis algorithms, with respect to particular analyses of interest. Additional experience and studies are required to measure the extent to which apparent benefits reported by the metric translate into positive tradeoffs for consumers of the control flow representations resulting from type inference on exceptions.

There are several possibilities for future work. First, one issue we would like to investigate is the impact of the type inference on regression test selection when applied to other types of test suites. For example, functional test suites may exhibit different characteristics than the JUnit test suites studied here, particularly with respect to program integration and the associated exception handling between components. Second, we wish to extend the study to include a broader range of object programs, to assess the extent to which results generalize. Third, since this study did not discover a positive cost-benefit tradeoff for the test selection problem, we would like to evaluate the tradeoffs associated with other consumers of the control flow representations produced by our tools; such evaluations can provide a more comprehensive understanding of the value of type inference on exceptions.

From the results presented in this paper and from future work, we hope to provide empirically grounded guidance to software practitioners in deciding when to make use of exceptional type inference to construct the representations of control flow used in testing and analysis activities.

Acknowledgements

This work has been supported by NSF awards CCR-0080898 and CCR-0347518 to the University of Nebraska - Lincoln, and by NSF awards CCR-9703108, CCR-9707792, CCR-0080900, and CCR-0306023 to Oregon State University.

References

- [1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).
- [2] <http://ant.apache.org>.
- [3] M. Berndt, O. Lhotak, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proc. Prog. Lang. Des. Impl.*, 2003.

- [4] Frédéric Besson, Thomas Jensen, Daniel Le Métayer, and Tommy Thorn. Model checking security properties of control flow graphs. *J. Comput. Secur.*, 9(3):217–250, 2001.
- [5] S. Beydeda and V. Gruhn. An integrated testing technique for component-based software. In *Proc. Int'l. Conf. Comp. Sys. Appl.*, June 2001.
- [6] Byeong-Mo Chang and Jang-Wu Jo. Estimating exception induced control flow for java. In *Wshop. Prog. Lang. Sys.*, 2001.
- [7] Zhenqiang Chen, Baowen Xu, and Jianjun Zhao. An overview of methods for dependence analysis of concurrent programs. *SIGPLAN Not.*, 37(8):45–52, 2002.
- [8] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Wshop. Prog. Anal. Softw. Tools Eng.*, pages 21–31, 1999.
- [9] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *Proc. Int'l. Symp. Emp. Softw. Eng.*, pages 60–70, 2004.
- [10] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
- [11] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Notices*, 23(7):35–46, June 1988.
- [12] <http://jakarta.apache.org/jmeter>.
- [13] John Jorgenson. Improving the precision and correctness of exception analysis in Soot. Technical report, McGill University, September 2003.
- [14] <http://jtopas.sourceforge.net/jtopas>.
- [15] Kalpesh Kapoor and Jonathan P. Bowen. Experimental evaluation of the tolerance for control-flow test criteria. *2nd U.K. Wshop. Softw. Test. Res.*, 14(3):167–187, 2003.
- [16] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.
- [17] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Wshop. Prog. Anal. Softw. Tools Eng.*, pages 73–79, 2001.
- [18] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *Proc. Int'l. Symp. Softw. Test. Anal.*, pages 1–11, 2002.
- [19] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 241–251, New York, NY, USA, 2004. ACM Press.

- [20] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22(8):529–551, 1996.
- [21] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, 1997.
- [22] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for java using annotated constraints. In *Proc. Conf. O.O. Lang. Sys.*, pages 43–55, 2001.
- [23] S. Sinha and M. J. Harrold. Analysis of programs with exception-handling constructs. In *Proc. Int’l. Conf. Softw. Maint.*, page 348, 1998.
- [24] Saurabh Sinha and Mary Jean Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Trans. Softw. Eng.*, 26(9):849–871, 2000.
- [25] <http://csce.unl.edu/~galileo/pub/sofya>.
- [26] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symp. Princ. Prog. Lang.*, pages 32–41, 1996.
- [27] M. Streckenbach and G. Snelting. Points-to for Java: A general framework and an empirical comparison. Technical report, University Passau, nov 2000.
- [28] Raja Vallé-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java Optimization Framework. In *Proc. CASCON*, pages 125–135, 1999.
- [29] Neil Walkinshaw, Marc Roper, and Murray Wood. The Java system dependence graph. In *Proc. Wshp. Source Code Anal. Manip.*, 2003.
- [30] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for java. In *Proc. Eur. Conf. O.O. Prog.*, pages 99–117, 2001.
- [31] John Whaley and Monica S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proc. Int’l. Static Anal. Symp.*, September 2002.
- [32] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. *ACM SIGPLAN Notices*, 34(10):187–206, 1999.
- [33] Jongwook Woo, Jehak Woo, Isabelle Attali, Denis Caromel, Jean-Luc Gaudiot, and Andrew L Wendelborn. Alias analysis for exceptions in Java. *Aust. Comput. Sci. Commun.*, 24(1):321–329, 2002.
- [34] <http://xml.apache.org/security>.