Assessing the Usefulness of Type Inference Algorithms in Representing Java Control Flow to Support Software Maintenance Tasks

Alex Kinneer
NVIDIA Corporation
Riata Trace Parkway, Austin, TX
akinneer@nvidia.com

Gregg Rothermel
Dept. of Computer Science and Engineering
University of Nebraska - Lincoln
grother@cse.unl.edu

Abstract

A wide range of techniques for supporting software maintenance tasks rely on representations of program control flow. The accuracy of these representations can be important to the effectiveness and efficiency of these techniques. The Java programming language has introduced structured exception handling features that complicate the task of representing control flow. Previous work has attempted to address these complications by using type inference algorithms to analyze the control flow effects of exceptions, but to date, there has been no study of whether the use of these algorithms is justified. In this paper we report results of an empirical study addressing this issue. We find that type inference algorithms can lead to more accurate representations of control flow, but this improvement does not necessarily translate into benefits for maintenance techniques that use them. It follows that type inference algorithms should not just automatically be applied; rather, the tradeoffs of applying them must first be assessed with respect to particular maintenance techniques and workloads.

1. Introduction

Many of the techniques that have been proposed as aids to software maintenance tasks rely on control flow representations of software. For example, techniques for improving regression testing through regression test selection (e.g., [10, 21, 23]) and test case prioritization (e.g. [24, 27]) rely explicitly on control flow information, while techniques based on program slicing [11] (such as for determining dependence clusters [3] and analyzing change impact [4]) depend on control flow information to calculate dependencies.

The accuracy of control flow representations can impact the effectiveness and efficiency of such techniques. The extent to which a control flow graph accounts for possible paths of execution through a program can determine the soundness of techniques, and the extent to which a control flow graph distinguishes paths that differ can determine the precision and efficiency of techniques.

The Java programming 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 [26]. 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. These exception handling features introduce new challenges for the representation of program control flow.

Prior research on analysis of exception handling constructs in Java programs (see Section 6) has sought to model the effects of such constructs on control flow. Such research has often been more concerned with creating *safe* representations of exceptional control flow (in which no potential paths are missed) than with creating *precise* representations (in which paths that are distinct are properly distinguished, and paths that are not feasible are omitted). To address problems of precision, the application of type inference algorithms to exceptions in Java programs has been suggested [26]. To date, however, we can find no reports of empirical studies evaluating the effects of using type inference algorithms to improve control flow graph precision.

In addition to considering precision, we must also consider the *cost* of applying type inference algorithms. In general, more powerful type inference are expected to produce more precise results at a higher cost. Because type inference algorithms have not been evaluated specifically in the context of analysis for exceptional control flow, however, there is little data available on the costs associated with their use in that context, and the extent to which precision may be gained from more costly algorithms is not clear.

While understanding the precision and cost of techniques for analyzing control flow using type inference algorithms is important, of greater practical importance is the impact of these attributes on the client techniques that use the resulting control flow graphs. More powerful type inference

978-1-4244-2614-0/08/$25.00 © 2008 IEEE

algorithms are justified only if the additional precision they provide yields sufficient benefits to client techniques, relative to the additional costs incurred. To date, we can find no research addressing this issue.

We have thus performed an empirical study, evaluating the use of four type inference algorithms for analyzing exceptional control flow for Java programs [26], to assess the costs and benefits of using these algorithms to model control flow. To perform the assessment we rely on three methods. First, we assess the costs of applying the algorithms when computing exceptional control flow and producing control flow graphs. Second, we use a metric that is independent of client techniques to assess the relative precision of the control flow graphs produced. Third, we investigate the effects of using the improved control flow graphs to support a client maintenance technique, regression test selection.

The results of our study show that, while the type inference algorithms investigated do differ in terms of costs and precision measured independently of client applications, the benefits of these algorithms do not extend, for the programs, versions, and test suites studied, to the client analysis problem of regression test selection. More generally, these results illustrate that the usefulness of precision-enhancing analyses such as type inference algorithms should not automatically be assumed to carry over to specific applications of specific maintenance techniques. Rather, the tradeoffs of applying such analyses should be assessed with respect to particular analyses and workloads.

The remainder of this paper is organized as follows. Section 2 provides necessary background information. Section 3 summarizes essential details of our type inference algorithm implementations. Section 4 presents our experiment design and results, and Section 5 discusses implications of our findings. Section 6 discusses related work, and Section 7 concludes and describes future work.

2. Background: Exceptions in Java

The Java language provides features for signaling exceptional conditions and implementing handlers to deal with exceptions. All exceptions are first-class objects that inherit from a single base class (`java.lang.Throwable`). The Java Language Specification (JLS) [9] creates additional classifications for exceptions by identifying the subclasses `java.lang.Error` and `java.lang.RuntimeException` of `Throwable` as special. Exceptions that extend from these classes are *unchecked* exceptions; they can occur anywhere in a program and need not be explicitly handled by the programmer. All other exceptions are *checked* exceptions; they occur at specific program locations and must be explicitly handled by the programmer. A checked 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 supersedes 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.

3. Exception Type Inference Techniques

The accuracy of control flow representations in the presence of exception features depends on the accuracy with which the possible binding handlers of exceptions can be determined, which is linked to the types of exceptions that can be raised at various program points. Type inference algorithms [6, 26] determine the possible types for expressions. *Safe* estimates of types ensure safe representations of control flow (in which no paths that might occur due to exceptions are missed), while more *precise* estimates, obtained at additional analysis *costs*, improve the precision of control flow representations, by more accurately distinguishing alternative paths and omitting infeasible paths.

In this work we evaluate the effects of four algorithms for performing type inference on exceptions. The algorithms we investigate replicate, as closely as possible, those pre-

sented in [26]. All four algorithms ensure the safety of the type estimate and thus of the resulting control flow. Three of the algorithms perform additional work, at additional cost, to produce type estimates that lead to control flow representations of greater precision. Here, we briefly describe the algorithms; for details see [16]. All algorithms were implemented by the first author, with common functionality shared or implemented equivalently in code.

Simple Baseline (base). This algorithm 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.

Flow-sensitive Intraprocedural (fsi). This algorithm performs a flow-sensitive backwards search from the point of each throw to find all reaching exception object instantiations. The search stops when the beginning of the method is reached. The algorithm 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 nonlocal variables may be assigned by the called method. The algorithm makes conservative estimates to determine possible exceptional control flow out of calls. The result of the search is classified as precise if object instantiations can be found on all reaching paths.

Flow-insensitive Interprocedural (fii). 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 call. 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 on the implementations that may be bound to the call and the enclosing handlers.

Combined Intraprocedural and Interprocedural (cmb). This algorithm applies the flow-sensitive algorithm first, then applies the flow-insensitive algorithm to blocks for which flow-sensitive analysis was imprecise.

4. Experiment Design and Results

Our goal is to empirically evaluate the use of type inference algorithms to improve representations of control flow, considering the cost of the algorithms, the precision of resulting representations, and the effects on a client analysis that uses the representations. This section describes our objects of analysis, variables and measures, experiment setup, threats to validity, and results.

4.1. Objects of Analysis

We used four Java programs as objects of analysis: `ant`, `xml-security`, `jmeter`, and `jtopas`, all drawn and available from the SIR repository [8]. `Ant` [1] 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 each of these programs. Each program is also equipped with JUnit test suites that were created by the developers of the programs.

Table 1 summarizes the characteristics of the most recent versions of each of these programs with respect to overall size, percentage of methods containing exception handling constructs (% MwH), and JUnit test suite sizes. Based on this information, and the data reported in [26], it can reasonably be argued that these programs are a representative sample of Java software being developed in practice in terms of size, use of exceptions, and 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.)

Table 1. Experiment Objects

Object	Versions	KLOC	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

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, and we use the four algorithms described in Section 3. The baseline algorithm *base* serves as the control for our experiment. This lets us compare more complex type inference algorithms against a low-cost conservative algorithm.

4.2.2 Dependent Variables and Measures

We chose three dependent variables and measures. The first two variables involve client-independent measures related strictly to type inference algorithm performance and the quality of the resulting exceptional control flow. These measures help us understand the general performance of the algorithms, in a manner that provides initial guidance on their relative strengths and weaknesses. The third measure is related to the support for a client consumer of the control flow graphs: the regression test selection technique described in [26]. We chose this client technique in particular because in [26], that technique is the technique that the type inference algorithms we study here were meant to assist.

Analysis Cost. The first dependent variable we measure is analysis cost; we measure this in terms of the time required to perform type inference and construct control flow graphs for all of the methods in the object program. This simple measure provides an understanding of the relative performance of the algorithms.

Analysis Precision. A client-independent metric of the relative precision of two type inference algorithms can be obtained by considering the control flow graphs produced by those algorithms. In particular, a type inference algorithm A' is more precise than another algorithm A on program P if, by eliminating infeasible edges, refining the precision of types on edges, and introducing new more precise feasible edges, A' produces a control flow graph on P that is more precise than the graph produced by A on P.

One way to evaluate a type inference algorithm A' for precision, following this definition, would be to compare the control flow graph computed using A' to a baseline "optimal" graph; however, it is not possible to compute an optimal graph for non-trivial systems such as those we consider. Further, when comparing control flow graphs, a metric that simply counts edges will not suffice, because one graph can have a larger or smaller number of edges that are more accurate in terms of type information than another graph and be of higher precision than that graph. Thus, we have devised a metric allowing pairs of graphs to be compared for precision in a manner that accounts for the various types of precision improvements that are possible. We provide the metric in algorithmic form here, with discussion of the intuition behind it; additional details are available in [16].

Our metric is designed to *award higher scores to control flow graphs that exhibit greater precision*; that is, 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 outgoing edges of these nodes. This makes it possible to compare outgoing edges from corresponding nodes in two graphs.

Algorithm 1 (CFG-assess) performs this comparison. The algorithm takes as inputs the sets of edges E and E' computed by two algorithms A and A', respectively, but with common edges (any edge in E that represents the same exceptional edge as an edge in E', and vice-versa) removed. For each edge e in E, CFG-assess first compares the exception associated with e against the exceptions associated with edges in E' to determine whether it is a superclass of any exceptions inferred by A'. If so, a point is awarded since this indicates that A' eliminated that class of exception as a possibility, either because more precise subclasses could be determined, or because it was infeasible.

Algorithm 1 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: for each e in E do
 3:   if class(e) is a superclass of any class(e') (e' ∈ E') then
 4:     score += 1 {A' eliminated an imprecise/infeasible edge}
 5:   else
 6:     if class(e) is a maximal class in E then
 7:       if class(e) is a subclass of any class(e') (e' ∈ E') then
 8:         {anti-refi nement; ignore}
 9:       else if class(e) is a checked exception then
10:         score += 1 {A' eliminated an imprecise/infeasible edge}
11:       end if
12:     else if class(e) is a subclass of class(e2) (e2 ∈ E) then
13:       if e2 was awarded a point then
14:         score += 1 {A' eliminated an infeasible edge}
15:       else
16:         {transitive anti-refi nement; ignore}
17:       end if
18:     end if
19:   end if
20: end for
21: for each e' in E' do
22:   if class(e') is a subclass of any class(e) (e ∈ E) then
23:     score += 1 {A' led to a refi nement}
24:   else
25:     {anti-refi nement; ignore.}
26:   end if
27: end for
```

If e is not a superclass of any type inferred by A', we 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 that A' performed better. Thus CFG-assess next tests whether the exception associated with e holds a maximal superclass relationship relative to other exceptions inferred by A. If this is true, and if the exception is a checked exception and is not a subclass of any type inferred by A', CFG-assess awards a point since A' eliminated the exception as infeasible. A point is not awarded, however, if the exception is a subclass of an exception inferred by A', as this indicates a loss of precision.

In the case where the exception associated with e does not hold a maximal superclass relationship relative to other exceptions inferred by A, CFG-assess tests whether it is a subclass of any other exceptions e_2 inferred by A. If it is, CFG-assess awards a point based on whether the inference of the superclass e_2 was awarded a point. The reasoning here is that the judgment of the metric with respect to the superclass transitively applies to any subclasses.

Finally, CFG-assess considers the unique edges produced by A' and 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 point is awarded.

Note that our metric can only partially assess the impact of unchecked exceptions, a limitation resulting from the fact that contextual information is used when conservative estimates are required, and this information is often not available for unchecked exceptions. However, we believe 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.

Support for Regression Test Selection. Regression test selection is the problem of choosing which test cases in a test suite should be re-run when a new version of a software system is to be regression tested [22]. One approach for selecting regression tests is to select those that exercise code changed since the software was last tested. This is the approach used by `DejaVu` [23] and its object-oriented counterpart, `DejaVoo` [10]. `DejaVu` performs simultaneous depth-first traversals on the control flow graphs for the old and new versions of a method to find places where code has been changed. Traces of the execution of test cases on the old version of the method are then used to select test cases that traverse edges that reach changed blocks in the graphs. `DejaVu` relies on control flow graphs to facilitate code instrumentation and to perform its graph traversals.

The safe representation of control flow is crucial to `DejaVu`'s ability to preserve fault detection with respect to selected test cases, and the degree of precision with which control flow is represented can be important to `Dejavu's` efficiency. This is particularly relevant for exceptional control flow, as conservative estimates of control flow are safe but often highly imprecise. Therefore, to assess the impact of type inference on 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 given the graphs generated using each of the type inference algorithms. This measure indicates whether the differences in exceptional control flow resulting from the various algorithms affect the test selection results on the given programs, versions, and test suites.

4.3. Experiment Setup

All of the algorithms we implemented 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. Experimentation on the larger subjects was performed on a Pentium-M 1.6 Ghz machine with 1 Gb RAM running SuSE Linux 9.1. (The use of multiple systems enabled faster data collection, and does not bias our results since we focus only on relative performances on particular objects.)

We implemented our algorithms within the Sofya analysis system [15, 17], which provides utilities for instrumentation and control flow graph construction. We used shell scripts to 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 type inference algorithm active.

4.4. Threats to Validity

External Validity. The primary threat to external validity for this study involves the representativeness of our object programs, versions, and associated test suites. Other systems and system releases, including larger and more complex industrial systems, may exhibit different behaviors and cost-benefit tradeoffs, as may other forms of test suites. However, as noted in Section 4.1, the programs we investigate do reflect several characteristics of a popular set of open-source Java programs, the versions we utilize are actual released versions of those programs, and the test suites we utilize are the actual suites provided with those programs by their developers.

Internal Validity. The primary threat to the internal validity of this experiment is possible faults in the implementation of the algorithms, and in the tools that we use to perform evaluation. We controlled for this 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. A second threat involves inconsistent decisions and practices in the implementation of the algorithms studied; for example, variation in the efficiency of implementations of common functionality between algorithms could bias timing assessments. We controlled for this threat by having all of our algorithms implemented by the same developer, utilizing consistent implementation decisions and shared code.

Construct Validity. The client-independent metric we calculate is not the only such metric that might be devised for evaluating the performance of type inference algorithms. As we note, this metric may not account for the influence of unchecked exceptions in all circumstances. Similarly, the client-dependent measure we utilize is not the only possible such metric. Rather than measuring numbers of tests selected, we could measure the time required to apply the regression test selection technique on our object programs and execute the selected tests, and compare the resulting savings. In our study, however, as we shall see, the use of this alternative metric would have added no value.

Table 2. Analysis Costs (seconds)

	base	fsi	fi i	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 3. ANOVA on Analysis Costs

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 i	-2.04e5	55400	-3.6e5	-48000	****
base-fsi	-18800	55400	-1.75e5	1.37e5	
cmb-fi i	52500	55400	-1.03e5	2.08e5	
cmb-fsi	2.37e5	55400	81700	3.93e5	****
fi i-fsi	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 i	-2.93e6	1.42e6	-6.88e6	1.01e6	
base-fsi	-4.48e4	1.42e6	-3.99e6	3.9e6	
cmb-fi i	1.42e5	1.42e6	-3.8e6	4.08e6	
cmb-fsi	3.03e6	1.42e6	-9.09e5	6.97e6	
fi i-fsi	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 i	-7.14e5	2.42e5	-1.41e6	-16500	****
base-fsi	-28400	2.42e5	-7.26e5	6.69e5	
cmb-fi i	68800	2.42e5	-6.28e5	7.66e5	
cmb-fsi	7.54e5	2.42e5	56800	1.45e6	****
fi i-fsi	68.5e5	2.42e5	-12000	1.38e6	

4.5. Results and Analysis

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

4.5.1 Analysis Cost

Table 2 displays analysis costs (CFG construction times) for the four algorithms, for each version of each object program. The data suggests that the algorithms compare similarly across objects and versions. The *base* algorithm is least expensive, but *fsi* is only slightly (at most 5%) more expensive. The interprocedural algorithms are more expensive than *base* and *fsi*, but their costs never exceed 1.85 times those of the base algorithm.

Boxplots of the analysis cost results (available in [16], omitted here for space) show similar variance among algorithms for each program. We performed per program ANOVAs on the data, for a significance level of 0.05, and in cases where differences were observed (on all programs but `jtopas`), used the Bonferroni method for multiple comparisons between algorithms to further assess the differences. Table 3 reports the results, with cases in which differences between algorithms were statistically significantly indicated by "****". For `xml-security` and `jmeter`, the analysis reveals statistically significant differences between the costs of the different algorithms. The p-value of 0.04 for `ant` suggests that there are statistically significant differences between algorithms for that object as well, but the (less powerful) Bonferroni comparisons do not identify differences between any specific pairs of algorithms. The results for `jtopas` did not indicate significance.

4.5.2 Analysis Precision

Table 4 reports the results obtained by applying our graph comparison metric to the pairs of graphs constructed by

Table 4. Analysis Precision Values

	base/ fsi	base/ fi i	base/ cmb	fsi/ fi i	fi i/ 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

the four algorithms, for each version of each object program. The columns correspond to comparisons between pairs of algorithms (as indicated in the header in format "first-algorithm/second-algorithm"). The values shown indicate the improvement gained by the second algorithm with respect to the first in each comparison, in terms of our metric. The magnitudes of the values should not be compared across object programs or program versions, but rather to the values reported (by other algorithm comparison combinations) within the same version.

The data clearly indicates the extent to which the more advanced algorithms can improve the accuracy of the reported control flow information. As expected, there is a progression in benefits from *base* to *fsi* to *fii*, with (not surprisingly) the results of the flow sensitive intraprocedural algorithm falling between those of the *base* and flow insensitive interprocedural algorithms. The flow insensitive interprocedural algorithm produces the greatest gains. On most versions, however, there is no gain associated with the combined algorithm with respect to the interprocedural algorithm.

Table 5. Numbers of Selected Tests

	no. of tests	base	fsi	fi i	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

4.5.3 Support for Regression Test Selection

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 algorithms, 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 regression test selection. In nearly all cases the test selection results are identical to those obtained when using control flow graphs constructed using the *base* algorithm. In only four cases — versions 5, 6, and 7 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 (significance level .05, p-value $= 1$, results not shown).

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 *base* and flow-sensitive intraprocedural (*fsi*) algorithms in the timing data reported in Table 2, there is strong evidence that the *fsi* algorithm should be preferred over the *base* algorithm. There is a greater cost associated with the interprocedural algorithms, which suggests that a more careful evaluation of the tradeoffs in using them is needed.

The results reported by the graph comparison metric serve as our first means of evaluating the tradeoffs. The metric further confirms that the *fsi* algorithm has precision superior to that of the *base* algorithm. Given that the algorithms have similar costs, there simply is no significant penalty associated with the application of *fsi* and thus there should be no reason to prefer the *base* algorithm.

Further, as already noted, there is a considerable precision improvement associated with the use of the flow-insensitive interprocedural algorithm (*fii*), which should not be surprising since the interprocedural algorithm benefits from the significant advantage of being able to refine exceptional flow paths associated with calls. Given that exceptions are often used to signal unexpected conditions to callers, there are more opportunities for the interprocedural algorithm to improve the precision of the control flow graphs subsequently created.

The graph comparison metric results provide evidence that the application of more advanced type inference algorithms has the potential to yield 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 benefits obtained by 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 regression test selection on the programs, versions, and test suites considered. This is significant, given that a primary motivation for introducing these algorithms in the first place was to improve the regression test selection process [26]. However, further analysis also suggests some additional factors worth considering when attempting to explain or generalize these results.

First, programming practices when dealing with exceptions could be partly responsible for results such as this. In particular, the practice of creating exceptions immediately at the throw point may lead to exceptional control flow that can be trivially inferred correctly at that point by any type inference algorithm, rendering test selection results on the resulting control flow graphs identical with respect to the corresponding portions of the graphs. It is also plausible that code for handling exceptions is more stable than other code, and thus less likely to be considered by difference-based test selection algorithms such as `DejaVu`. Propagation of exceptions to high level handlers is an instantiation of this design strategy, and seems to be used to some extent in our object programs. 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 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 also be partly responsible for the results. Because `DejaVu's` test selection is based on differences between program versions, increased precision in exceptional control flow representation will translate into selected test cases only if changes occur on exceptional control flow paths. If exception handling code is more stable than other code as considered above, then additional precision in the representation may not yield notable improvement for this particular control flow dependent application. In general, this situation may be further complicated if changes on exceptional flow paths are masked by earlier differences on non-exceptional paths. `DejaVu` selects test cases based on the first dangerous edge found on paths; it then does not require further information about changes occuring further along the path. We did not observe this situation to occur with frequency, however, on our particular object programs and versions.

One concern that arises from the use of programs obtained from the field involves particular characteristics of their test suites. Test suites provided with the objects we studied do not focus specifically on boundary and exceptional use cases within the object programs. That said, our results do pertain to the actual test suites provided with the programs, thus representing results practitioners would expect in practice with such test suites.

Finally, it may also be the case that typical JUnit test cases 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 testing effects involving interactions likely to lead to exceptions. For this reason, other forms of tests, such as functional tests, may yield different results.

6. Related Work

Beyond the work reported in [26] (described in Section 1), 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 [5] present a set-based analysis to estimate the propagation of exceptions based on an operational semantics for Java. Their work is primarily 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 dependence on such type inference techniques.

Choi et al. [7] 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 algorithms for 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. address 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 [29] 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 algorithms, and includes no substantive empirical evaluation of techniques with respect to client analyses, and only limited evaluation of the implications of applying type inference.

Beyond the analysis of exceptions, considerable attention has been given to the question of points-to analysis in Java [2, 18, 19, 20, 25, 28, 30, 31, 32]. However, such work has not been limited to the question of impact on 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.

7. Conclusions and Future Work

We have performed a study of the costs and benefits 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 maintenance activities that depend on control flow representations. However, this evidence comes from the computation of a metric assessing the overall precision of the control flow graphs constructed. Our study did not demonstrate worthwhile benefits for the particular maintenance task of regression test selection. For the programs, versions, and test suites considered, the cost associated with performing type inference is not justified for that particular client technique.

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 client maintenance techniques of interest, and particular workloads to which those techniques are to be applied. Additional experience and studies are required to measure the extent to which apparent benefits reported by the client-independent 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 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 benefits for regression test selection, we would like to evaluate the tradeoffs associated with other consumers of control flow representations; such evaluations can provide a more comprehensive understanding of the value of using 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 by software maintenance techniques.

Acknowledgements

This work was supported in part by NSF under Awards CNS-0454203, CCR-0080898, and CCR-0347518 to the University of Nebraska - Lincoln.

References

[1] http://ant.apache.org.

[2] M. Berndl, O. Lhotak, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 103–114, June 2003.

[3] D. Binkley and M. Harman. Locating dependence clusters and dependence pollution. In *Proceedings of the International Conference on Software Maintenance*, pages 177–186, Sept. 2005.

[4] S. Bohner and R. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, 1996.

[5] B.-M. Chang and J.-W. Jo. Estimating exception induced control flow for Java. In *The Second Asian Workshop on Programming Languages and Systems,*, pages 377–387, Dec. 2001.

[6] R. Chatterjee, B. G. Ryder, and W. A. Landi. Complexity of concrete type-inference in the presence of exceptions. *Lecture Notes in Computer Science*, 1381:57–74, 1998.

[7] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Workshop on Program Analysis and Software Tools and Engineering*, pages 21–31, Sept. 1999.

[8] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[10] M. J. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. Spoon, and A. Gujarathi. Regression test selection for Java software. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 312–326, Oct. 2001.

[11] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Notices*, 23(7):35–46, June 1988.

[12] http://jakarta.apache.org/jmeter.

[13] J. Jorgenson. Improving the precision and correctness of exception analysis in Soot. Technical report, McGill University, Sept. 2003.

[14] http://jtopas.sourceforge.net/jtopas.

[15] A. Kinneer, M. Dwyer, and G. Rothermel. Sofya: A flexible framework for development of dynamic program analysis for Java software. Technical Report TR-UNL-CSE-2006-0006, University of Nebraska - Lincoln, Apr. 2006.

[16] A. Kinneer and G. Rothermel. Assessing the cost-benefits of using type inference algorithms to improve the representation of exceptional control flow in Java. Technical Report TR-UNL-CSE-2005-0002, University of Nebraska – Lincoln, May 2005.

[17] A. J. Kinneer, M. B. Dwyer, and G. Rothermel. Sofya: Supporting rapid development of dynamic program analyses for Java. In *Proceedings of the International Conference on Software Engineering*, pages 51–52, May 2007.

[18] O. Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[19] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 73–79, June 2001.

[20] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 1–11, July 2002.

[21] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 241–251, Nov. 2004.

[22] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[23] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, 1997.

[24] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *Proceedings of the International Conference on Software Maintenance*, pages 179–188, Aug. 1999.

[25] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Proceedings of the Conference on Object-Oriented Languages and Systems*, pages 43–55, Oct. 2001.

[26] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering*, 26(9):849–871, 2000.

[27] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the International Symposium on Software Testing and Analysis*, July 2002.

[28] M. Streckenbach and G. Snelting. Points-to for Java: A general framework and an empirical comparison. Technical report, University Passau, Nov. 2000.

[29] R. Vallé-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research*, pages 125–135, 1999.

[30] T. Wang and S. F. Smith. Precise constraint-based type inference for Java. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 99–117, 2001.

[31] J. Whaley and M. S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the International Static Analysis Symposium*, pages 180–195, Sept. 2002.

[32] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. *ACM SIGPLAN Notices*, 34(10):187–206, 1999.

[33] J. Woo, J. Woo, I. Attali, D. Caromel, J.-L. Gaudiot, and A. L. Wendelborn. Alias analysis for exceptions in Java. *Australian Computer Science Communications*, 24(1):321–329, 2002.

[34] http://xml.apache.org/security.