논·문 17-1-6

Exception Propagation Analysis for Java*

Jang-Wu Jo[1] and Byeong-Mo Chang[2]

[1]Division of Computer Engineering, Pusan Univ. of Foreign Studies, jjw@pufs.ac.kr
[2]Dept. of Computer Science, Sookmyung Women's Univ. chang@sookmyung.ac.kr

Abstract

Exception analyses so far cannot provide information on the propagation of thrown exceptions, which is necessary to construct interprocedural control flow graph, visualize exception propagation, and slice exception-related parts of programs. In this paper, we propose a set-based analysis, which can estimate exception propagation paths of Java programs finitely. Our analysis is unique to the other exception analyses, in that it can show exception propagation paths. We show the soundness of the analysis and also provide some applications.

1. Introduction

Exception facilities in Java allow the programmer to define, throw and catch exceptional conditions. Exceptional conditions are brought to the attention of another expression where the thrown exceptions may be handled. Because unhandled exceptions will abort the program's execution, it is important to make sure at compile-time that the input program will have no uncaught exceptions at run-time.

There have been several uncaught

* This work was supported by grant No. (R01-2002-000-00363-0) from the Basic Research Program of the Korea Science & Engineering Foundation.

exception analyses, that estimate uncaught exceptions [1,9,15,20]. The current JDK Java compiler also provides a type-based exception analysis which relies on programmer's specification for checking against uncaught exceptions[9]. Several interprocedural exception analyses were proposed for Java in [1,15,20] that estimate uncaught exceptions independently of the programmer's specifications.

However, they estimate uncaught exceptions only by their names, so that they cannot provide information on *propagation paths* of thrown exceptions, which is necessary to construct interprocedural control flow graph [17], visualize exception propagation, and slice exception-related parts of programs.

In this paper, we propose a static analysis based on set-based framework, which estimates exception propagation paths of Java programs. Our analysis is unique among the other exception analyses, in that it can show exception propagation paths.

Based on the operational semantics of Java with exception propagation taken into consideration [7], we first design set-constraint construction rules to estimate exception propagation paths. We then design naive constraint solving rules S. We can compute the possibly infinite solution $lm_S(C)$ of the constraints C by applying the naive solving rules S. This solution can be infinite due to recursive calls in the input program. So, we design the new solving rules S' by slightly modifying the naive rules for finite solution. The main idea is to represent an exception propagation path with the edges constituting the path and the unique identifier of the thrown exception. We can compute the finite solution $lm_{S'}(C)$ of the constraints C by applying the new solving rules S'.

We sketch the soundness of the naive solution with respect to the collecting semantics which can be defined by lifting the standard semantics so as to consider sets of concrete traces. We then show the soundness of the new finite solution with respect to the possibly infinite solution by induction on the length of traces.

We also show how analysis information can be applied to constructing interprocedural control flow graph, visualizing exception propagation, and slicing exception-related parts of programs.

The next section describes the core of Java, on which our presentation is based. Section 3 describes a static analysis to estimate exception propagation paths. Section 4 describes constraint solving and its correctness. Section 5 presents some applications of this analysis. Section 6 discusses related works and Section 7 concludes this paper.

2. Source Language

For presentation brevity we consider an imaginary core of Java with its exception constructs [20]. Its abstract syntax is in Figure 1.

P	::= C*	program
C	::= class *c* ext *c* {var *x** *M**}	class definition
M	::= *m*(*x*)[throws *c**] = e	method definition
e	::= *id*	variable
	| *id* := *e*	assignment
	| new *c*	new object
	| this	self object
	| *e* ; *e*	sequence
	| if *e* then *e* else *e*	branch
	| throw *e*	exception raise
	| try *e* catch (*c x e*)	exception handle
	| *e.m*(*e*)	method call
id	::= *x*	method parameter
	| *id.x*	field variable
c		class name
m		method name
x		variable name

Figure 1 Abstract Syntax of a Core of Java

A program is a sequence of class definitions. Class bodies consist of field variable declarations and method definitions. A method definition consists of the method name, its parameter, and its body expression. Every expression's result is an object. Assignment expression returns the object of its righthand side expression. Sequence expression returns the object of the last expression in the sequence. A method call returns the object from the method body. The `try` expression `try` e_0 `catch` (c x e_1) evaluates e_0 first. If the expression returns a normal object then this object is the result of the `try` expression. If an exception is thrown from e_0 and its class is covered by c then the handler expression e_1 is evaluated with the exception object bound to x. If the thrown exception is not covered by class c then the thrown exception continues to propagate back along the evaluation chain until it meets another handler. Note that a nested `try` expression can express multiple handlers for a single expression e_0 : `try` (`try` e_0 `catch` (c_1 x_1 e_1)) `catch` (c_2 x_2 e_2). The exception object e_0 is thrown by `throw` e_0. The programmers have to declare in a method definition any exception class whose exceptions may escape from its body.

Note that exceptions are first-class objects in Java. Like normal objects, they can be defined by classes, instantiated, assigned to variables, passed as parameters, etc. Exception facilities in Java allow the programmer to define, throw and catch exceptional conditions.

The semantics of Java was proposed in [7] with exception throwing, propagation and handling taken into consideration.

Let's consider a simple example in Figure 2 which shows exception propagation. The thrown exception `E1` from the method `m2` is propagated through `m2` and `m1`, and caught by the `try-catch` in

the main method. The exception E2 may

```
class Demo {
    public static void main(){
        try {
          m1();
        } catch (Exception x) {;}
        ...
        m3();
    }
    void m1() {
        m2();
    }
    void m2() {
        throw new E1;
    }
    void m3() {
        if(...) throw new E2;
        if(...) m3();
    }
}
```

Fig. 2. An Example Program for Exception Propagation

be thrown from the method m3. If it is thrown, then it is propagated until the main method and not caught. The method m3 also has a recursive call to itself, so that the thrown exception E2 may propagated back through the recursive calls.

3. Set-constraint construction

Our analysis is designed based on the set-constraint framework [11]. We assume class information *class(e)* is already available for every expression e in the analysis, which can be obtained by type inference [6,7,13], or class analysis [5]. Note that exception classes are normal classes in Java.

Each set constraint is of the form $X \supseteq se$ where X is a set variable and se is a set expression. The meaning of a set constraint $X \supseteq se$ is intuitive: set X contains the set represented by set expression se. Multiple constraints are conjunctions. We write C for such conjunctive set of constraints.

In case of our analysis, the set expression is of this form:

$se \rightarrow \langle c^l, l \rangle$ thrown exception from l
| X set varaible
| $se \cup se$ union
| $se - \{c_1, .., c_n\}$ catching exceptions
| $se \cdot \lambda$ exception propagation

The thrown exception from a throw expression labelled with l is represented by $\langle c^l, l \rangle$ where c is the name or class of the exception and l is the location or label of the throw expression. We will take c^l as the *unique identifier* of the thrown exception in this paper. The set expression $se - \{c_1, ..., c_n\}$ is for catching exceptions. The set expression $se \cdot l$ records exception propagation paths by appending a label l to se.

Semantics of set expressions naturally follows from their corresponding language constructs. The formal semantics of set expressions is defined by an interpretation

I that maps from set expressions to sets of values in V = *Exception* × *Trace*, where *Exception* = *ExnName* × *Label*, where *ExnName* is the set of exception names, and *Trace* = *Label**. A trace $\tau \in$ *Trace* is a sequence of labels in *Label*, which is an exception propagation path. For example, $I(se \cdot l') = I(se) \cdot l'$ where $I(se) \cdot l' = \{< c^i, l_1 \ldots l_n l' > | < c^i, l_1 \ldots l_n > \in I(se)\}$. We call an interpretation I a *model* (a solution) of a conjunction C of constraints if, for each constraint $X \supseteq se$ in C, $I(X) \supseteq I(se)$.

Collected constraints for a program guarantee the existence of its least solution (model) because every operator is monotonic (in terms of set-inclusion) and each constraint's left-hand-side is a single variable [11]. We write $lm(C)$ for the least model of a collection C of constraints.

Set-based analysis consists of two phases [11]: collecting set constraints and solving them. The first phase constructs set-constraints by the construction rules, that describe the data flows between the expressions of the analyzed program. The second phase finds the sets of values that satisfy the constraints. A solution is a table or mapping from set variables in the constraints to the finite descriptions of

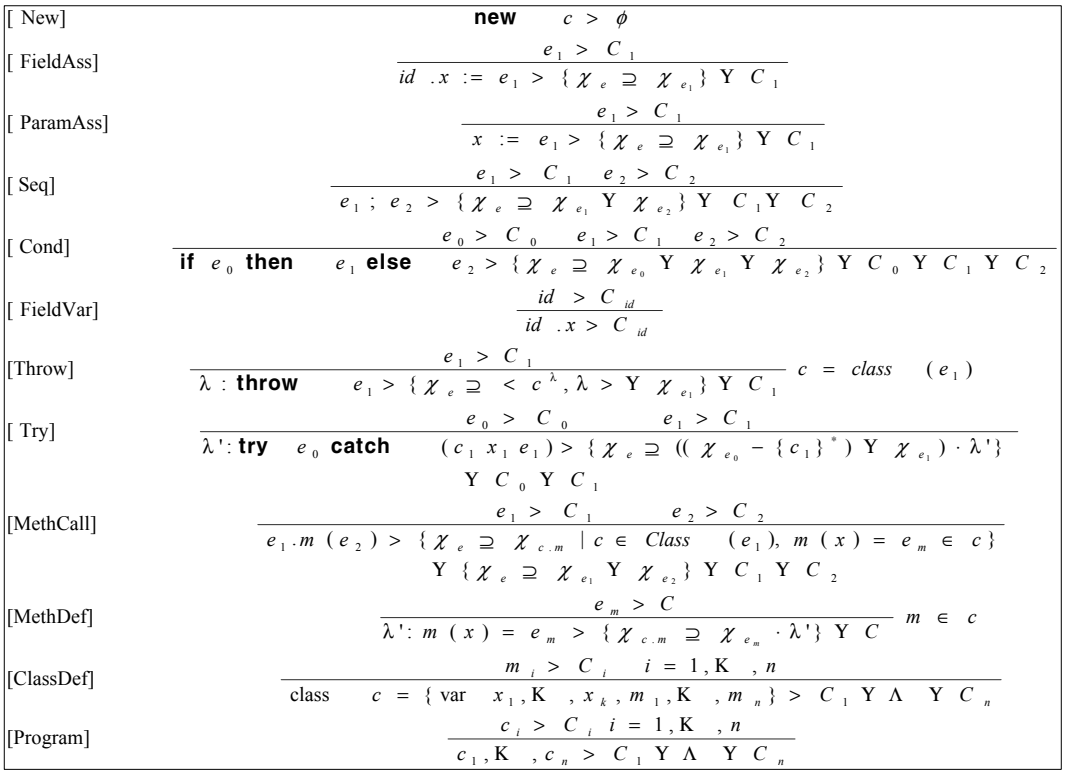

Fig. 3. Set-constraints construction rules

such sets of values.

We first present a constraint system that estimates traces of thrown exceptions from every expression of the input program. For simple presentation, our analysis traces exception propagation paths by recording the labels of just exception-related constructs such as `throw`, `try-catch`, and method declarations. We assume this kind of expressions e has a label l, which is denoted by $l : e$. If more detailed trace information is necessary, it is possible to record other expressions such as method call and `try`-block.

Figure 3 has the rules to generate set-constraints for every expression. For our analysis, every expression e of the program has a constraint: $X_e \supseteq se$. The X_e is a set-variable for collecting the propagation paths of the thrown exceptions inside e. The subscript e of X_e denotes the current expression to which the rule applies. The relation "$e \triangleright C$" is read "constraints C are generated from expression e".

Consider the rule for `throw` expression with a label l:

$$[\text{Throw}] \quad \frac{e_1 > C_1}{\lambda : \textbf{throw}\, e_1 > \{\chi_e \supseteq < c^{\lambda}, \lambda > \mathrm{Y}\, \chi_{e_1}\} \mathrm{Y}\, C_1} \; c = class(e_1)$$

It throws an exception e_1, which is represented as $\langle c^{l}, l \rangle$ where $c = class(e_1)$ is the name or class of the exception and l is the label of the `throw` statement, which is an origin of the exception. In the following, c^{l} is used as the *unique identifier* of the thrown exception. Prior to the throwing, it can have uncaught exceptions from inside e_1 too.

Consider the rule for `try` expression with a label l' :

$$[\text{Try}] \quad \frac{e_0 > C_0 \qquad e_1 > C_1}{\lambda' : \textbf{try}\, e_0\, \textbf{catch}(c_1\, x_1\, e_1) > \{\chi_e \supseteq ((\chi_{e_0} - \{c_1\}^*) \mathrm{Y}\, \chi_{e_1}) \cdot \lambda'\} \; \mathrm{Y}\, C_0 \mathrm{Y}\, C_1}$$

Thrown exceptions from e_0 can be caught by x_1 only when their classes are covered by c_1. After this catching, exceptions can also be thrown during the handling inside e_1. Uncaught exceptions from this expression are followed by the label l' to record the exception propagation path. Hence, $X_e \supseteq ((X_{e0} - \{c_1\}^*) \cup X_{e1}) \cdot \lambda'$, where $\{c\}^*$ represents all the descendant classes of a class c including itself.

Consider the rule for method call:

$$[\text{MethCall}] \quad \frac{e_1 > C_1 \qquad e_2 > C_2}{e_1.m(e_2) > \{\chi_e \supseteq \chi_{c.m} \mid c \in Class(e_1), m(x) = e_m \in c\} \; \mathrm{Y}\{\chi_e \supseteq \chi_{e_1} \mathrm{Y}\, \chi_{e_2}\} \mathrm{Y} C_1 \mathrm{Y} C_2}$$

Uncaught exceptions from the call expression first include those from the subexpressions e_1 and e_2 : $X_e \supseteq X_{e1} \cup X_{e2}$. The method $m(x) = e_m$ is the one defined inside the classes $c \in class(e_1)$ of e_1's objects. Hence, $X_e \supseteq X_{c.m}$ for uncaught exceptions. (The subscript $c.m$ indicates the index for the method m of the class c.)

Consider the rule for method definition with a label l' :

[MethDef] $$\frac{e_m > C}{\lambda : m(x) = e_m > \{\chi_{c.m} \supseteq \chi_{e_m} \cdot \lambda'\} \mathrm{Y} C} \; m \in c$$

Uncaught exceptions from the this method m include those from the method body e_m, which are followed by the label l' to record exception propagation paths.

We can construct the following set-constraints by applying the construction rules to the example program in Figure 2. When we write down the set-constraints, we use the statements with some simplification instead of labels for better understanding.

$X_{main} \supseteq X_{try-catch} \cdot main$
$X_{main} \supseteq X_{m3} \cdot main$
$X_{try-catch} \supseteq (X_{m1} - \{Exception\}*) \cdot try-catch$
$X_{m1} \supseteq X_{m2} \cdot m1$
$X_{m2} \supseteq X_{throwE1} \cdot m2$
$X_{throwE1} \supseteq \langle E1, throwE1 \rangle$
$X_{m3} \supseteq X_{throwE2} \cdot m3$
$X_{throwE2} \supseteq \langle E2, throwE2 \rangle$
$X_{m3} \supseteq X_{m3} \cdot m3$

4. Solving the set-constraints

We first design naive constraint solving rules S. We can compute the possibly infinite solution $lm_S(C)$ of the constraints C by applying the naive solving rules S. This solution can be infinite due to recursive calls in the input program.

$$\frac{X \supseteq X_1 \cup X_2}{X \supseteq X_1} \quad \frac{X \supseteq X_1 \cup X_2}{X \supseteq X_2} \quad \frac{X \supseteq \nu \quad \nu \supseteq \langle c^\lambda, \tau \rangle}{X \supseteq \langle c^\lambda, \tau \rangle}$$

$$\frac{X \supseteq X_1 \cdot \lambda' \quad X_1 \supseteq \langle c^\lambda, \tau \rangle}{X \supseteq \langle c^\lambda, \tau \cdot \lambda' \rangle}$$

$$\frac{X \supseteq X_1 - \{c_1, \ldots, c_k\} \quad X_1 \supseteq \langle c^\lambda, \tau \rangle \quad c \notin \{c_1, \ldots, c_k\}}{X \supseteq \langle c^\lambda, \tau \rangle}$$

Fig. 4. Rules S for solving set constraints

The naive solving phase closes the initial constraint set C under the rules S in Figure 4. Intuitively, the rules propagate values along all the possible data flow paths in the program. Each propagation rule decomposes compound set constraints into smaller ones, which approximates the steps of the value flows between expressions.

Consider the rule for tracing exception propagation path :

$$\frac{X \supseteq X_1 \cdot \lambda' \quad X_1 \supseteq \langle c^\lambda, \tau \rangle}{X \supseteq \langle c^\lambda, \tau \cdot \lambda' \rangle}$$

This rule simulates the propagation path of thrown exceptions by appending a label l' to the exception trace τ in X_1. Other rules are similarly straightforward from the semantics of corresponding set expressions.

We can computes the solution $lm_S(C)$ of set-constraints C by applying the rules S in Figure 3. We can sketch the soundness of the solution as follows:

[**Theorem** 1] Let P be a program and C be the set-constraints constructed by the rules in Figure 3. Every exception trace of

P is included in the solution $lm_S(C)$.

Proof sketch.

We first have to lift the standard semantics to a collecting semantics called set-based approximation so as to collect sets of concrete traces, because a static program point can be associated with a set of traces. Correctness proofs can be done with respect to this collecting semantics by the fixpoint induction over the continuous functions that are derived from our constraint system as in [4]. □

We can compute the infinite solution for the set-constraints C in Figure 3 by applying the rule S. Some of the solution are as follows:

$$
\begin{aligned}
lm_S(C)(X_{m1}) &\supseteq \{ \langle E1, throwE1 \cdot m2 \cdot m1\rangle\}\\
lm_S(C)(X_{m2}) &\supseteq \{ \langle E1, throwE1 \cdot m2\rangle\}\\
lm_S(C)(X_{m3}) &\supseteq \{ \langle E2, throwE2 \cdot m3\rangle,\\
&\qquad \langle E2, throwE2 \cdot m3 \cdot m3\rangle,\\
&\qquad \langle E2, throwE2 \cdot m3 \cdot m3 \cdot m3\rangle,\\
&\qquad \ldots\}\\
lm_S(C)(X_{main}) &\supseteq \{ \langle E2, throwE2 \cdot m3 \cdot main\rangle,\\
&\qquad \langle E2, throwE2 \cdot m3 \cdot m3 \cdot main\rangle,\\
&\qquad \langle E2, throwE2 \cdot m3 \cdot m3 \cdot m3 \cdot main\rangle,\\
&\qquad \ldots\}
\end{aligned}
$$

The solution can be infinite in case there are recursive methods, which contain uncaught exception(s). We need to find a finite representation for the possibly infinite solution.

So, we design the new solving rules S' for finite solution by modifying the exception propagation rule in S. The main idea is to represent an exception propagation path, that is a trace, with the edges consisting the path and the unique identifer of the thrown exception. They are finite because the number of exception names and labels is finite.

To do this, at every step of exception propagation, we record the last two labels(that is an edge) together with the unique identifer of the thrown exception. We modify the rule for tracing exception propagation as follows :

$$\frac{X \supseteq X_1 \cdot \lambda' \quad X_1 \supseteq \langle c^{\lambda}, \tau \rangle}{X \supseteq \langle c^{\lambda}, |\tau \cdot \lambda'|_2 \rangle}$$

where $|\lambda_1 \cdots \lambda_n|_2 = \lambda_{n-1}\lambda_n$ when $n \geq 2$

This rule simulates the propagation of thrown exceptions, by recording the last two labels together with the thrown exception's unique identifier c^l. Because this is done at every step of exception propagation, the dropped information has also been included into the solution together with the unique identifier c^l.

In the following, S' denotes the solving rules S with the propagation rule being replaced by the new one. Our analysis computes the least model $lm_{S'}(C)$ of set-constraints C by applying the new solving rules S'. We can compute the

solution for the set-constraints C in Figure 3 by applying the new rule S'. Some of the solution are as follows:

$lm_{S'}(C)(X_{m1}) \supseteq \{ <E1, throwE1 \cdot m2 \cdot m1>\}$
$lm_{S'}(C)(X_{m2}) \supseteq \{ <E1, throwE1 \cdot m2>\}$
$lm_{S'}(C)(X_{m3}) \supseteq \{ <E2, throwE2 \cdot m3>,$
$<E2, m3 \cdot m3>\}$
$lm_{S'}(C)(X_{main}) \supseteq \{ <E2, m3 \cdot main>\}$

We can see exception propagation paths by defining the *exception propagation graph* of the solution $lm_{S'}(C)$.

[Definition 1] Let C be the set-constraints constructed for a program P. *Exception propagation graph* of the solution $lm_{S'}(C)$ is defined to be a graph $<V,E>$ where V is the set of labels in P and $E = \{\lambda_1 \rightarrow^{c^\lambda} \lambda_2 | <c^\lambda, \lambda_1\lambda_2> \in lm_{S'}(C)(X)$ for a set variable X in $C\}$ where $\lambda_1 \rightarrow^{c^\lambda} \lambda_2$ denotes an edge from λ to λ labelled with c^λ.

We can easily draw the exception propagation graph for the finite solution for the example program by making following labelled edges :

$throwE1 \rightarrow^{E1} m2$
$m2 \rightarrow^{E1} m1$
$throwE2 \rightarrow^{E2} m3$
$m3 \rightarrow^{E2} main$
$m3 \rightarrow^{E2} m3$

We can show the soundness of the finite solution by finding a path in the exception propagation graph for every trace in the possibly infinite solution.

[Theorem 2] Let $lm_S(C)$ and $lm_{S'}(C)$ be the solutions of set-constraints C by applying the solving rules S and S' respectively. For every exception trace $<c^\lambda, \tau>$, in $lm_S(C)$, there is a path for τ with every edge labelled c^λ in the exception propagation graph of $lm_{S'}(C)$.

Proof.

We will prove this theorem by tracing the computation process for the solution $lm_{S'}(C)$. Let $\tau = \lambda_1 \ldots \lambda_i \ldots \lambda_n$. The proof is by induction on i.

Base: When $i = 2$, $<c^\lambda, \lambda_1\lambda_2>$ is trivially included in the solution, so there is a path for $\lambda_1\lambda_2$ labelled with c^λ in the graph.

Hypothesis: Assume that there is a path for $\lambda_1 \ldots \lambda_i$ labelled with c^λ, which means that the solution by applying the new rules S' has already collected $<c^\lambda, \lambda_1\lambda_2> \ldots <c^\lambda, \lambda_{i-1}\lambda_i>$.

Step: We consider $\lambda_i\lambda_{i+1}$. There are two cases for this. If $<c^\lambda, \lambda_i\lambda_{i+1}>$ has not included in the solution yet, then it will be included into the solution in the following reasons:

(1) the solution $lm_S(C)$ includes $<c^\lambda, \lambda_1 \ldots$

$\lambda_i...\lambda_n>$ where λ_i is appended to $\lambda_1...\lambda_{i-1}$ by the propagation rule in S

(2) there is a corresponding propagation rule in S' and

(3) by induction hypothesis, $<c^{\lambda}, \lambda_{i-1}\lambda_i>$ is already included in the solution by applying S'.

We can now find a path for $\lambda_1...\lambda_i...\lambda_{i+1}$ by traversing the existing path and the new edge $\lambda_i\lambda_{i+1}$. If $<c^{\lambda}, \lambda_i\lambda_{i+1}>$ has already been included in the solution by applying the new rules S', then we can find a path for $\lambda_1...\lambda_i.\lambda_{i+1}$ by traversing the existing path for $\lambda_1...\lambda_i$ and the existing edge $\lambda_i.\lambda_{i+1}$. □

Implementation can compute the solution by the conventional iterative fixpoint method because the solution space is finite: exception classes, pairs of labels in the program.

5. Applications

To show the usefulness of the exception trace, we provide three applications of our analysis. The first one is to construct *interprocedural control-flow graph*} (ICFG) which incorporates exception-induced control flow, and the second one is program slicing that accounts for exceptions constructs, and the third one is to visualize exception control flows.

5.1 ICFG

The *control-flow graph* (CFG) is a representation of control flow relation that exists in a program, in which nodes represent statements and edges represent the flow of control between statements. Many program-analysis techniques, such as data-flow, control-dependence and exception analysis, and program slicing depend on control-flow information. For these analyses to be safe and useful, the control-flow representation should incorporate the exception-induced control flow.

Recently, several researchers have considered the effects of exception-induced control flow on various types of analyses. Failure to account for the effects of exception in performing analyses can result in incorrect analysis information. They construct control-flow representation for exception-related constructs [3, 17].

Given an interprocedural control-flow graph with normal control flow, we can easily merge *exception propagation graph* (ouranalysis result) onto it so as to construct *interprocedural control-flow graph with exceptional control flow.*

5.2 Program Slicing

A *program slice* of a program P, with

respect to a *slicing criterion* <*s, V*>, where *s* is a program point and *V* is a set of program variables, includes statements in *P* that may influence, or to be influenced by, the values of the variables in *V* at *s* [12]. There are two alternative approaches to computing slices, that either propagate solutions of data-flow equations using a control-flow representation [10, 19], or perform graph reachability on system dependence graphs [12, 18].

Using our interprocedural control-flow representation, the slicing technique in [10] can be extended to take into consideration the effects of exception-handling constructs.

Our trace information can also be used to create system dependence graph that incorporates control and data dependence induced by exception constructs.

5.3 Visualizing Exception Flows

The exception trace information can be used to visualize exception propagation. This can include the origin of exceptions, handler of exceptions, and propagation path of exceptions. This information can guide programmers to detect uncaught exceptions, handle exceptions more specifically and declare more exactly. Moreover, this information can guide programmers to put exception handlers at appropriate places by tracing exception propagation.

We are planning to develop a visualization system which highlights or slices only the source codes in the propagation trace of a thrown exception, if programmers select a `throw` statement.

6. Related works

Ryder and colleagues [16] and Sinha and Harrold [17] conducted a study of the usage patterns of exception-handling constructs in Java programs. Their study offers an evidence to support our belief that exception-handling constructs are used frequently in Java programs and more accurate exception flow information is necessary.

There are several research directions for exception constructs. The first one is modeling program execution, which includes constructing CFG with normal and exceptional control flows, and using the representation to perform various types of analysis. The second one is enabling a developer to make better use of exception mechanism, which includes analysis of uncaught exceptions, analysis of exception flow to facilitate understanding of the exception behavior.

Choi and colleagues [3] construct intraprocedural control-flow representation called the factored control-flow graph (FCFG) for exception-handling constructs,

and use the representation to perform data-flow analyses. Sinha and Harold [17] discusses the effects of excpetion-handling constructs on several analyses such as control-flow, data-flow, and control dependence analysis. They present techniques to construct representations for programs with checked exception and exception-handling constructs. Chatterjee and Ryder [2] describe an approach to performing points-to analysis that incorporates exceptional control flow. They also provide an algorithm for computing definition-use pairs that arise because of exception variables, and along exceptional control-flow paths.

In Java [9], the JDK compiler ensures, by an intraprocedural analysis, that clients of a method either handle the exceptions declared by that method, or explicitly redeclare them.

Robillard and Murphy [15] have developed Jex: a tool for analyzing uncaught exceptions in Java. They describe a tool that extracts the uncaught exceptions in a Java program, and generates views of the exception structure.

In our previous work [1, 20], we proposed interprocedural exception analysis that estimates uncaught exceptions independently of programmers's specified exceptions. We compared our analysis with JDK-style analysis by experiments on large realistic Java programs, and have shown that our analysis is able to detect uncaught exceptions, unnecessary `catch` and `throws` clauses effectively.

Several exception analyses have been introduced for ML based on abstract interpretation and set-constraint framework [21]. Fahndrich and Aiken [8] have applied their BANE toolkit to the analysis of uncaught exceptions in SML. Their system is based on equality constraints to keep track of exception values. Fessaux and Leroy desiged an exception analysis for OCaml based on type and effect systems, and provides good performance for real OCaml programs [14].

Our approach is unique to the other exception analyses, in that it can show the exception propagation paths.

7. Conclusions

In this paper, we have proposed a set-based analysis, which estimates exception propagation path of Java programs. For more detailed tracing, our analysis can be simply extended to incorporate labels of other expressions such as method calls and `try`-block. Moreover, we can also check *caught* exceptions by making another set variable Y for every `try` e_0 `catch`$(c\ x\ e_1)$ expression and setting a set-constraint like $Y \supseteq X_{e0} \cap \{c\}^*$.

Our analysis provides information on the propagation of thrown exceptions, which can guide programmers to detect uncaught exceptions, handle exceptions more specifically and declare more exactly. Moreover, this information can guide programmers to put exception handlers at appropriate places by tracing exception propagation path.

The analysis information can also be applied to construct interprocedural control flow graph [17]}, visualize exception propagation, and slice exception-related parts of programs. In particular, we are planning to develop a visualization system which highlights or slices only the source codes in the propagation trace of a thrown exception, if programmers select a `throw` statement.

References

[1] B.-M. chang, J. Jo. K. Yi, and K. Choe, "Interprocedural Exception Analysis for Java", *Proceedings of ACM Symposium on Applied Computing*, pp 620-625, Mar. 2001.

[2] R. K. Chatterjee, B. G. Ryder, and W. A. Landi, "Complexity of concrete type-inference in the presence of exceptions", *Lecture notes in Computer Science*, vol. 1381, pp. 57-74, Apr. 1998.

[3] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar, "Efficient and precise modeling of exceptions for analysis of Java programs", *Proceedings of '99 ACM SIGPLAN-SIGSOFT Workshop on PASTE*, September 1999, pp. 21-31.

[4] Patrick Cousot and Radhia Cousot. "Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form", *Proceedings of the 7th international conference on computer-aided verification edition*, 1995.

[5] G. DeFouw, D. Grove, and C. Chambers, "Fast interprocedural class analysis", *Proceedings of 25th ACM POPL* pp 222-236, January 1998.

[6] S. Drossopoulou, and S. Eisenbach, "Java is type safe-probably", *Proceedings of 97 ECOOP*, 1997.

[7] S. Drossopoulou, and T. Valkevych, "Java type soundness revisited", Techical Report, Imperial College, November 1999.

[8] M. Fähndrich, J.S. Foster, A. Aiken, and J. Cu, "Tracking down exceptions in Standard ML programs", Technical report, UC Berkeley, Computer Science Division, 1998.

[9] J. Gosling, B. Joy, and G. Steele, *The Java Programming Language Specification*, Addison-Wesley, 1996.

[10] M. Harrold and N. Ci, "Reuse Driven Interprocedural Slicing", *Proceedings of ICSE*, April 1998.

[11] N. Heintze, Set-based program

analysis. Ph.D thesis, Carnegie Mellon University, October 1992.

[12] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs", *ACM TOPLAS*, 11(3), pp 345-387, July 1989.

[13] T. Nipkow and D. V. Oheimb, "Java is type safe-definitely", *Proceedings of 25th ACM POPL*, January 1998.

[14] F. Pessaux and X. Leroy, "Type-based analysis of uncaught exceptions", *Proceedings of 26th ACM POPL*, January 1999.

[15] M. P. Robillard and G. C. Murphy, "Analyzing exception flow in Java programs", in *Proc. of '99 European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of software Engineering*, pp. 322-337, Springer-Verlag.

[16] B. G. Ryder, D. Smith, U. Kremer, M. Gordon, and N. Shah, "A static study of Java exceptions using JESP," Tech. Rep. DCS-TR-403, Rutgers University, Nov. 1999.

[17] S. Sinha and M. Harrold, "Analysis and testing of programs with exception-handling constructs", *IEEE Transactions on Software Engineering* 26(9) (2000).

[18] S. Sinha, M. Harrold, and G. Rothermel, "System dependence graph based slicing of programs with arbitrary interprocedural control flow", *Proceedings of ICSE*, pp. 432-441.

[19] M. Weiser, "Program Slicing", *IEEE Transactions on Software Engineering*, 10(4), pp 352-357, July 1984.

[20] K. Yi and B.-M. Chang "Exception analysis for Java", ECOOP Workshop on Formal Techniques for Java Programs, June 1999, Lisbon, Portugal.

[21] K. Yi and S. Ryu. "Towards a cost-effective estimation of uncaught exceptions in SML programs", *Proceedings of the 4th Static Analysis Symposium*, September 1997.

Jang-Wu Jo

1997 - now Associate Professor, Pusan University of Foreign Studies
2003, Ph.D. Computer Science, KAIST
1994, M.S. Computer Science, Seoul National University
1992, B.S. Computer Science & Statistics, Seoul National University

Byeong-Mo Chang

1995 - now Associate Professor, Sookmyung Womens's Univ.
1994 - 1995 Postdoctoral Fellow, ETRI 1994, Ph.D. Computer Science, KAIST
1990, M.S. Computer Science, KAIST
1988, B.S. Computer Engineering, Seoul National University