Estimating Exception-Induced Control Flow for Java *

Byeong-Mo Chang and Jang-Wu Jo

Department of Computer Science
Sookmyung Women's University, Seoul 140-742, Korea
chang@cs.sookmyung.ac.kr

Department of Computer Engineering
Pusan University of Foreign Studies
Pusan, Korea
jjw@taejo.pufs.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 estimates exception propagation of Java programs. To formalize exception propagation, we first describes an operational semantics with exception propagation taken into consideration. We design a set-based analysis to estimates exception propagation based on this operational semantics, and show its correctness. We also provide some applications of the analysis.

Keywords: Java, exception propagation, exception analysis, set-based analysis

1 Introduction

Exception facilities in Java allow the programmer to define, throw and catch exceptional conditions. Exceptional conditions are brought (by a `throw` expression) 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 exception analyses, that estimates uncaught exceptions. Current JDK Java compiler also provides an exception analysis which relies on programmer's specification for checking against uncaught exceptions[9]. An interprocedural exception analysis of Java programs was presented in [21, 1] that estimates their exception flows independently of the programmer's specifications.

However, they estimate uncaught exceptions only by their names, so that they cannot provide information on the *propagation* of thrown exceptions, which is necessary to construct interprocedural control flow graph [18], 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 of Java programs. Our analysis needs class (or type) information $class(e)$ for expression e, which can be obtained by type inference or class analysis as in [5, 6, 13, 7].

To formalize exception propagation, we first describes an operational semantics with exception propagation taken into consideration. Based on the operational semantics, we design a static analysis to estimate exception propagation by defining set-constraint construction rules and constraint solving rules, and then show its correctness. 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.

* This work was supported in part by grant No. 2000-1-30300-009-2 from the Basic Research Program of the Korea Science & Engineering Foundation.

The next section describes the core of Java, on which our presentation is based. Section 3 describes an operational semantics with exception propagation taken into consideration. Section 4 describes a static analysis to estimate exception propagation. Section 5 describes constraint solving and its correctness. Section 6 presents some applications of this analysis. Section 7 discusses related works and Section 8 concludes this paper.

P	::=	C^*	program
C	::=	`class` c `ext` c' { `var` x^* M^* }	class definition
M	::=	$m(x)$ = e [`throws` c^*]	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

Fig. 1. Abstract Syntax of a Core of Java

2 Source Language

For presentation brevity we consider an imaginary core of Java with its exception constructs [21]. Its abstract syntax is in Figure 1. 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

$$\texttt{try}\ e_0\ \texttt{catch}\ (c\ x\ e_1)$$

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 nested `try` expression can express multiple handlers for a single expression e_0 :

$$\texttt{try (try}\ e_0\ \texttt{catch}\ (c_1\, x_1\, e_1))\ \texttt{catch}\ (c_2\, x_2\, e_2).$$

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, passes as parameters, etc. Exception facilities in Java allow the programmer to define, throw and catch exceptional conditions.

$\sigma \in$	Env	$= Name \to Loc$	environment
$s \in$	$Store$	$= Loc \to Obj$	store
$o \in$	Obj	$= Name \to Loc$	object
$l \in$	Loc	$\supseteq \{@true, @false, @1, @2, ...\}$	location
$n, \hat{n} \in$	$ExnLoc$	$\subseteq Loc$	exception location
$\tau \in$	$Trace$	$= Label^*$	trace

$\mathbf{Eval} \in Expr \times Env \times Store \to Loc \times Trace \times Store$

$\mathbf{Eval}(\lambda e \lambda \sigma \lambda s)$ case e of

$id.x$:	letx	$\langle l, \tau, s\rangle = \mathbf{Eval}(id, \sigma, s)$
	in	$\langle s(l)(x), s\rangle$
this :		$\langle \sigma(\mathsf{this}), \epsilon, s\rangle$
ℓ : throw e_1 :	letx	$\langle n, \tau, s'\rangle = \mathbf{Eval}(e_1, \sigma, s)$
	in	$\langle \hat{n}, \ell, s'\rangle$
ℓ : try e_0 catch $(c_1 x_1 e_1)$:	let	$\langle l_0, \tau_0, s_0\rangle = \mathbf{Eval}(e_0, \sigma)$
	in	if $l_0 = \hat{n} \in ExnLoc$ and $s_0(n) \in c_1$
		then $\mathbf{Eval}(e_i, \sigma[x \mapsto l], s_0[' \mapsto s_0(n)])$ $(new\ l)$
		else $\langle l_0, \tau_0 \cdot \ell, s_0\rangle$
if e_1 then e_2 else e_3 :	letx	$\langle l_1, \tau_1, s_1\rangle = \mathbf{Eval}(e_1, \sigma, s)$
	in	if $l_1 = @true$
		then $\mathbf{Eval}(e_2, \sigma, s_1)$
		else $\mathbf{Eval}(e_3, \sigma, s_1)$
$e_1; e_2$:	letx	$\langle l_1, \tau_1, s_1\rangle = \mathbf{Eval}(e_1, \sigma, s)$
	in	$\mathbf{Eval}(e_2, \sigma, s_1)$
$id.x = e_1$:	letx	$\langle l, \tau, s\rangle = \mathbf{Eval}(id, \sigma, s)$
		$\langle l_1, \tau_1, s_1\rangle = \mathbf{Eval}(e_1, \sigma, s)$
	in	$\langle l_1, \epsilon, s_1[s_1(l)(x) \mapsto s_1(l_1)]\rangle$
new$c$:	if	$c = \{\mathsf{var} x_1, ..., x_n M*\}$
	let	$o = [x_1 \mapsto \mathsf{null}, ..., x_n \mapsto \mathsf{null}]$ and $s' = s[l \mapsto o]$ $(new\ l)$
	in	$\langle l, \epsilon, s'\rangle$
$e_1.m(e_2)$:	letx	$\langle l_1, s_1\rangle = \mathbf{Eval}(e_1, \sigma, s)$
		$\langle l_2, s_2\rangle = \mathbf{Eval}(e_2, \sigma, s_1)$
	in	if $\ell : m = \lambda x.e_b \in s_0(l_0)$
	let	$\langle l_m, \tau_m, s_m\rangle = \mathbf{Eval}(e_b, \sigma[\mathsf{this} \mapsto l, x \mapsto l'], s[l \mapsto s_1(l_1), l' \mapsto s_2(l_2)](new\ l,\ l')$
	in	if $l_0 = \hat{n} \in ExnLoc$ then $\langle l_m, \tau_m, s_m\rangle$ else $\langle l_m, \tau_m \cdot \ell, s_m\rangle$

Fig. 2. Standard operational semantics

3 Operational Semantics

Recently several formal operational semantics for Java have been proposed [6, 13]. However, they do not consider exception propagation in their semantics. So, we define an operational semantics with exception propagation taken into consideration.

This semantics does not consider method overloading because it can be resolved using type information. Before describing operational semantics and analysis, we assume the following preprocessing for the input program:

- expand all classes considering method overriding
- every field variable access without qualification, for example x, is translated into `this`.x for a consistent treatment.

We define some basic notations for describing the semantics. We denote a method m in a class c by $m \in c$, formally if `class` $c = \{$`var` $x_1, ..., x_k, m_1, ..., m_n\}$ and $m = m_i$ for some i. We also denote

by $m \in o$ if $m \in c$ and o is an instance of c. We denote an instance(object) o of a class c by $o \in c$, and also denote $o \in c$ if $o \in c'$ and c' is a subclass of c. Note that a class is a subclass of itself.

The semantics of the language is specified in Figure 2 following the style in [22]. To express the exception convention, we use the "letx" notation as in [**?**]

$$\text{letx } l = []_1 \text{ in } []_2$$

as a shorthand of

$$\text{let } l = []_1 \text{ in if } l = \hat{n} \in ExnLoc \text{ then } l \text{ else } []_2$$

That is, the evaluation of the "letx" bindings terminates with the first if its result is a thrown exception. This thrown exception becomes the result in the conclusion of the "letx" expression. When no exception is thrown, "letx" is the same as "let".

The standard evaluation function **Eval** returns a location(reference) to an object and an exception trace for a given environment and store.

$$\textbf{Eval} \in Expr \times Env \times Store \rightarrow Loc \times Trace \times Store$$

When a new object o need to be bound to a variable x, a new location l is allocated in the store s

$$s \in Store = Loc \rightarrow Obj$$

and the object is written in that location $s[l \mapsto o]$.

The environment σ

$$\sigma \in Env = Name \rightarrow Loc$$

then maps the name to the location $\sigma[x \mapsto l]$. Thus, for example, one argument of a method is mapped to different locations, one for each invocation of the method. When variable x's value(a reference to an object in Java) is needed, we just fetch x's location $\sigma(x)$ from the environment σ. If we need an object itself, for example, when accessing a field of the object, it can be fetched from the store s by $s(\sigma(x))$.

An object o is a mapping from field names to locations(references) to objects:

$$o \in Obj = Name \rightarrow Loc$$

We consider primitive values as special objects for a consistent view in the semantics, and for example, $@true$ means a location to the special object $true$. In Java, exception objects are first-class objects, so they can be assigned, passed, and returned as other objects. In this semantics, we denote $\hat{n}$ a thrown exception of an exception object(in fact, a location to an exception object) n.

The trace τ

$$\tau \in Trace = Label^*$$

is a trace of exception propagation, which is a sequence of labels in $Label$. In this paper, we are only interested in labels of exception related constructs such as `throw`, `try` and methods.

4 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. There are several choices for class information. First, we can approximate it using type information, since Java is shown to be type sound [6, 13, 7]. Second, we can utilize information from class analysis [5, 14], which estimates for each expression e the classes (including exception classes) that the expression e's object belongs to. Note that exception classes are normal classes in Java.

For our analysis, every expression e of the program has a constraint: $\mathcal{X}_e \supseteq se$. The $\mathcal{X}_e$ is a set variable for the traces of thrown exceptions from the expression e. The meaning of a set constraint

$\mathcal{X} \supseteq se$ is intuitive: set $\mathcal{X}$ contains the set represented by set expression se. Multiple constraints are conjunctions. We write $\mathcal{C}$ for such conjunctive set of constraints.

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

$$\begin{array}{lll} se \rightarrow & \langle c^\ell, \ell \rangle & \text{thrown exception from } \ell \\ & | \ \mathcal{X} & \text{set varaible} \\ & | \ se \cup se & \text{union} \\ & | \ se - \{c_1, ..., c_n\} & \text{catching exceptions} \\ & | \ se \cdot \ell & \text{exception propagation} \end{array}$$

The thrown exception from a `throw` expression labelled with ℓ is represented by c^ℓ. Semantics of set expressions naturally follows from their corresponding language constructs. The formal semantics of set expressions is defined by an interpretation $\mathcal{I}$ that maps from set expressions to sets of values in

$$V = Exception \times Trace$$

where $Exception = ExnName \times Label$ where $ExnName$ is the set of exception names, and $Trace = Label^*$. For example, $\mathcal{I}(se \cdot \ell') = \mathcal{I}(se) \cdot \ell'$ where $\mathcal{I}(se) \cdot \ell' = \{\langle c^\ell, \ell_1 \cdots \ell_n \ell' \rangle | \langle c^\ell, \ell_1 \cdots \ell_n \rangle \in \mathcal{I}(se)\}$. We call an interpretation $\mathcal{I}$ a *model* (a solution) of a conjunction $\mathcal{C}$ of constraints if, for each constraint $\mathcal{X} \supseteq se$ in $\mathcal{C}$, $\mathcal{I}(\mathcal{X}) \supseteq \mathcal{I}(se)$.

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 $lm(\mathcal{C})$ for the least model of a collection $\mathcal{C}$ 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 such sets of values.

We first present a constraint system that estimates traces of thrown exceptions from every expression of the input program. This analysis traces exception propagation by recording labels of exception-related constructs such as `throw` expressions, `try-catch` expressions, and method declarations, even though it is possible to record every expression. We assume this kind of expressions e has a label ℓ, which is denoted by $\ell : e$.

Figure 3 has the rules to generate set-constraints for every expression. For our analysis, every expression e of the program has a constraint: $\mathcal{X}_e \supseteq se$. The $\mathcal{X}_e$ is a set-variable for tracing the propagation of the expression e's thrown exceptions. The subscript e of set variables $\mathcal{X}_e$ denotes the current expression to which the rule applies. The relation "$e \rhd \mathcal{C}$" is read "constraints $\mathcal{C}$ are generated from expression e."

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

$$[\text{Throw}] \ \frac{e_1 \rhd \mathcal{C}_1}{\ell : \texttt{throw}\ e_1 \rhd \{\mathcal{X}_e \supseteq \langle c^\ell, \ell \rangle \cup \mathcal{X}_{e_1}\} \cup \mathcal{C}_1} \quad c = class(e_1)$$

It throws exceptions e_1 or, prior to throwing, it can have uncaught exceptions from inside e_1 too. The thrown exception from this expression are followed by the label ℓ for recording exception trace.

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

$$[\text{Try}] \ \frac{e_0 \rhd \mathcal{C}_0 \quad e_1 \rhd \mathcal{C}_1}{\ell' : \texttt{try}\ e_0\ \texttt{catch}(c_1\ x_1\ e_1) \rhd \{\mathcal{X}_e \supseteq ((\mathcal{X}_{e_0} - \{c_1\}^*) \cup \mathcal{X}_{e_1}) \cdot \ell'\} \cup \mathcal{C}_0 \cup \mathcal{C}_1}$$

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. Hence, $\mathcal{X}_e \supseteq ((\mathcal{X}_{e_0} - \{c_1\}^*) \cup \mathcal{X}_{e_1}) \cdot \ell'$, where $\{c\}^*$ represents all the subclasses of a class c and uncaught exceptions from this expression are followed by the label ℓ' for recording the exception propagation.

[New] $\texttt{new}\ c \rhd \emptyset$

[FieldAss] $$\frac{e_1 \rhd \mathcal{C}_1}{id.x := e_1 \rhd \{\mathcal{X}_e \supseteq \mathcal{X}_{e_1}\} \cup \mathcal{C}_1}$$

[ParamAss] $$\frac{e_1 \rhd \mathcal{C}_1}{x\ \texttt{:=}\ e_1 \rhd \{\mathcal{X}_e \supseteq \mathcal{X}_{e_1}\} \cup \mathcal{C}_1}$$

[Seq] $$\frac{e_1 \rhd \mathcal{C}_1 \quad e_2 \rhd \mathcal{C}_2}{e_1\,;e_2 \rhd \{\mathcal{X}_e \supseteq \mathcal{X}_{e_1} \cup \mathcal{X}_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[Cond] $$\frac{e_0 \rhd \mathcal{C}_0 \quad e_1 \rhd \mathcal{C}_1 \quad e_2 \rhd \mathcal{C}_2}{\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rhd \{\mathcal{X}_e \supseteq \mathcal{X}_{e_0} \cup \mathcal{X}_{e_1} \cup \mathcal{X}_{e_2}\} \cup \mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[FieldVar] $$\frac{id \rhd \mathcal{C}_{id}}{id.x \rhd \mathcal{C}_{id}}$$

[Throw] $$\frac{e_1 \rhd \mathcal{C}_1}{\ell : \texttt{throw}\ e_1 \rhd \{\mathcal{X}_e \supseteq \langle c^\ell, \ell\rangle \cup \mathcal{X}_{e_1}\} \cup \mathcal{C}_1} \quad c = class(e_1)$$

[Try] $$\frac{e_0 \rhd \mathcal{C}_0 \quad e_1 \rhd \mathcal{C}_1}{\ell' : \texttt{try}\ e_0\ \texttt{catch}(c_1\ x_1\ e_1) \rhd \{\mathcal{X}_e \supseteq ((\mathcal{X}_{e_0} - \{c_1\}^*) \cup \mathcal{X}_{e_1}) \cdot \ell'\} \cup \mathcal{C}_0 \cup \mathcal{C}_1}$$

[MethCall] $$\frac{e_1 \rhd \mathcal{C}_1 \quad e_2 \rhd \mathcal{C}_2}{e_1.m(e_2) \rhd \{\mathcal{X}_e \supseteq \mathcal{X}_{c.m} | c \in Class(e_1), m(x) = e_m \in c\} \cup \{\mathcal{X}_e \supseteq \mathcal{X}_{e_1} \cup \mathcal{X}_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[MethDef] $$\frac{e_m \rhd \mathcal{C}}{\ell' : m(x)\ \texttt{=}\ e_m \rhd \{\mathcal{X}_{c.m} \supseteq \mathcal{X}_{e_m} \cdot \ell'\} \cup \mathcal{C}} \quad m \in c$$

[ClassDef] $$\frac{m_i \rhd \mathcal{C}_i,\ i = 1, \cdots, n}{\texttt{class}\ c = \{\text{var}\ x_1, \cdots, x_k, m_1, \cdots, m_n\} \rhd \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

[Program] $$\frac{C_i \rhd \mathcal{C}_i,\ i = 1, \cdots, n}{C_1, \cdots, C_n \rhd \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

Fig. 3. Set-constraint construction rules

Consider the rule for method call:

$$\text{[MethCall]}\ \frac{e_1 \rhd \mathcal{C}_1 \quad e_2 \rhd \mathcal{C}_2}{e_1.m(e_2) \rhd \{\mathcal{X}_e \supseteq \mathcal{X}_{c.m} | c \in class(e_1), m(x) = e_m \in c\} \cup \{\mathcal{X}_e \supseteq \mathcal{X}_{e_1} \cup \mathcal{X}_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

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

Consider the rule for method definition with a label ℓ:

$$\text{[MethDef]}\ \frac{e_m \rhd \mathcal{C}}{\ell' : m(x)\ \texttt{=}\ e_m \rhd \{\mathcal{X}_{c.m} \supseteq \mathcal{X}_{e_m} \cdot \ell'\} \cup \mathcal{C}} \quad m \in c$$

Uncaught exceptions from the this method include those from the method body e_m, which are followed by the label ℓ' for recording exception propagation.

5 Solving the set-constraints

The solving phase closes the initial constraint set $\mathcal{C}$ 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

$$\frac{\mathcal{X} \supseteq \mathcal{X}_1 \cup \mathcal{X}_2}{\mathcal{X} \supseteq \mathcal{X}_1} \quad \frac{\mathcal{X} \supseteq \mathcal{X}_1 \cup \mathcal{X}_2}{\mathcal{X} \supseteq \mathcal{X}_2} \quad \frac{\mathcal{X} \supseteq \mathcal{Y} \quad \mathcal{Y} \supseteq \langle c^\ell, \tau \rangle}{\mathcal{X} \supseteq \langle c^\ell, \tau \rangle}$$

$$\frac{\mathcal{X} \supseteq \mathcal{X}_1 \cdot \ell' \quad \mathcal{X}_1 \supseteq \langle c, \tau \rangle}{\mathcal{X} \supseteq \langle c^\ell, \tau \cdot \ell' \rangle}$$

$$\frac{\mathcal{X} \supseteq \mathcal{X}_1 - \{c_1, \cdots, c_k\} \quad \mathcal{X}_1 \supseteq \langle c^\ell, \tau \rangle \quad c \notin \{c_1, \cdots, c_k\}}{\mathcal{X} \supseteq \langle c^\ell, \tau \rangle}$$

Fig. 4. Rules S for solving set constraints

decomposes compound set constraints into smaller ones, which approximates the steps of the value flows between expressions.

Consider the rule for tracing exception propagation :

$$\frac{\mathcal{X} \supseteq \mathcal{X}_1 \cdot \ell' \quad \mathcal{X}_1 \supseteq \langle c^\ell, \tau \rangle}{\mathcal{X} \supseteq \langle c, \tau \cdot \ell' \rangle}$$

This rule simulates the propagation of thrown exceptions by appending a label ℓ' to the exception trace τ in $\mathcal{X}_1$. Other rules are similarly straightforward from the semantics of corresponding set expressions.

Our analysis computes the solution $lm_S(\mathcal{C})$ of set-constraints $\mathcal{C}$ by applying the rules S in Figure 3. We can show the correctness of the solution as follows:

Theorem 1. *Let P be a program and $\mathcal{C}$ be the set-constraints constructed by the rules in Figure 3. Every exception trace of P is included in the solution $lm_S(\mathcal{C})$.*
Proof. Correctness proofs can be done by the fixpoint induction over the continuous functions that are derived [4] from our constraint system. □

However, the solution may be infinite in case there are recursive methods, which contain uncaught exception(s). So, we need to find a finite representation for the possibly infinite solution.

To represent traces finitely, we record just the last two labels of traces instead of collecting entire traces. They are finite because the number of exception names and labels is finite. To do this, we modify the rule for tracing exception propagation as follows :

$$\frac{\mathcal{X} \supseteq \mathcal{X}_1 \cdot \ell' \quad \mathcal{X}_1 \supseteq \langle c^\ell, \tau \rangle}{\mathcal{X} \supseteq \langle c, |\tau \cdot \ell'|_2 \rangle}$$

where

$$|\ell_1 \cdots \ell_n|_2 = \ell_{n-1}\ell_n \text{ when } n > 2$$

This rule simulates the propagation of thrown exceptions, by recording the last two labels of traces. Even though we throw away traces except the last two labels, we do not lose the information because they have already been included into the solution.

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

We will show the safety of this analysis by defining *exception propagation graph* of the solution $lm_{S'}(\mathcal{C})$.

Definition 1. Let $\mathcal{C}$ be the set-constraints constructed for a program P. *Exception propagation graph* of the solution $lm_{S'}(\mathcal{C})$ is defined to be a graph $\langle V, E \rangle$ where V is the set of labels in P and $E = \{\ell_1 \rightarrow^{c^\ell} \ell_2 | \langle c^\ell, \ell_1\ell_2 \rangle \in lm_{S'}(\mathcal{C})(\mathcal{X}),\ \mathcal{X}$ is a set variable in $\mathcal{C}\}$ where $\ell_1 \rightarrow^{c^\ell} \ell_2$ denotes an edge from ℓ_1 to ℓ_2 labelled with c^ℓ.

The safety of the analysis using the new rules S' can be stated as follows.

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

Proof. We will prove this theorem by tracing the computation process for the solution $lm_{S'}(\mathcal{C})$. Let $\tau = \ell_1 \cdots \ell_i \cdots \ell_n$. The proof is by induction on i.

Base: When $i = 2$, $\langle c^\ell, \ell_1\ell_2\rangle$ is trivially included in the solution, so there is a path for $\ell_1\ell_2$ labelled with c^ℓ in the graph.

Hypothesis: Assume that there is a path for $\ell_1 \cdots \ell_i$ labelled with c^ℓ, which means that the solution by applying the new rules S' has already collected $\langle c^\ell, \ell_1\ell_2\rangle \cdots \langle c^\ell, \ell_{i-1}\ell_i\rangle$.

Step: We consider $\ell_i\ell_{i+1}$. There are two cases for this. If $\langle c^\ell, \ell_i\ell_{i+1}\rangle$ has not included in the solution yet, then it will be included into the solution in the following reasons:

(1) the solution $lm_S(\mathcal{C})$ includes $\langle c^\ell, \ell_1 \cdots \ell_i \cdots \ell_n\rangle$ where ℓ_i is appended to $\ell_1 \cdots \ell_{i-1}$ by the propagation rule in S

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

(3) by induction hypothesis, $\langle c^\ell, \ell_{i-1}\ell_i\rangle$ is already included in the solution by applying S' .

We can now find a path for $\ell_1 \cdots \ell_i\ell_{i+1}$ by traversing the existing path and the new edge $\ell_i\ell_{i+1}$. If $\langle c^\ell, \ell_i\ell_{i+1}\rangle$ has already been included in the solution by applying the new rules S', then it is not added into the solution. In this case, we can find a path for $\ell_1 \cdots \ell_i\ell_{i+1}$ by traversing the existing path for $\ell_1 \cdots \ell_i$ and the existing edge $\ell_i\ell_{i+1}$. □

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.

6 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.

6.1 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 and control-dependence analysis, and software-engineering techniques, such as program slicing and testings, use 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, 18].

Given an interprocedural control-flow graph with normal control flow, we can easily merge *exception propagation graph* (our analysis result) onto it so as to construct *interprocedural control-flow graph with exceptional control flow* As proposed in [18], this ICFG can consist of CFGs for each procedure; at each call site, normal control flow is represented by call edge and return edge. Exceptional control flow are represented by nodes for exception handling constructs, edges for intraprocedural exceptional control flow, *exceptional exit* node and *exceptional return* edge, where*exceptional exit* node models the propagation of exceptions by a procedure, and *exceptional return* edge represents interprocedural exceptional control flow.

6.2 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 [20, 10], or perform graph reachability on system dependence graphs[12, 19].

Using our interprocedural control-flow representation in Figure **??**, 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.

6.3 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.

7 Related works

Ryder and colleagues [17] and Sinha and Harrold [18] 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 [18] discusses the effects of excpetion-handling constructs on several analyses such as control-flow, data-flow, and control dependence analysis. They presents 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 [16] have developed Jex: a tool for analyzing exception flows in Java. They describe a tool that extracts the flow of exceptions in a Java program, and generates views of the exception structure.

In our previous work[21, 1], we proposed interprocedural exception analysis that estimates uncaught exceptions independently of programmers's declared 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 by Yi for ML based on abstract interpretation and set-constraint framework [22]. Fähndrich 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 [15].

8 Conclusions

In this paper, we have proposed a set-based analysis, which estimates exception propagation of Java programs. To formalize exception propagation, we first describe an operational semantics with exception propagation taken into consideration. We design set-based analysis to estimates exception propagation based on this operational semantics, and show its correctness.

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.

The analysis information can also be applied to construct interprocedural control flow graph [18], 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 Program Analysis for Software Tools and Engineering,* 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. *Lecture Notes in Computer Science*, volume 939, pages 293–308. Springer-Verlag, *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 SIGPLAN-SIGACT Symposium on Principles of Programming Languages* pp 222-236, Januaray 1998.
6. S. Drossopoulou, and S. Eisenbach, Java is type safe-probably, *Proceedings of 97 European Conference on Object-Oriented Programming*, 1997
7. S. Drossopoulou, and T. Valkevych, Java type soundness revisited. Techical Report, Imperial College, November 1999. Also available from: http://www-doc.ic.ac.uk/ scd.
8. M. Fähndrich, J.S. Foster, A. Aiken,a nd J. Cu. Tracking down exceptions in Standard ML programs. Techical report, University of California at Berkeley, Computer Science Division, 1998.
9. J. Gosling, B. Joy, and G. Steele, *The Java Programming Language Specification*, Addison-Wesley Longman,1996.
10. M. Harrold and N. Ci, Reuse Driven Interprocedural Slicing, *Proceedings of the International Conference on Software Engineering,* 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 Transactions on Programming Languages and Systems*, 11(3), pp 345-387, July 1989.
13. T. Nipkow and D. V. Oheimb Java is type safe-definitely, *Proceedings of 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*,January 1998.
14. J. Palsberg and M. I. Schwarzbach, Object-oriented type inference, *Proceedings of '91 ACM Conference on OOPSLA*, pp. 141-161, 1991.
15. F. Pessaux and X. Leroy, Type-based analysis of uncaught exceptions. *Proceedings of 26th ACM Conference on Principles of Programming Languages*, January 1999.
16. 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.
17. B. G. Ryder, D. Smith, U. Kremer, M. Gordon, and N. Shah, "A static study of Java exceptions using JSEP," Tech. Rep. DCS-TR-403, Rutgers University, Nov. 1999.
18. S. Sinha and M. Harrold, Analysis and Testing of Programs With Exception-Handling Constructs, *IEEE Transations on Software Engineering* 26(9) (2000).

19. S. Sinha, M. Harrold, and G. Rothermel, System-Dependence-Graph-Based Slicing of Programs with Arbitrary Interprocedural Control Flow, *Proceedings of the International Conference on Software Engineering,* May 1999, pp. 432-441.
20. M. Weiser, Program Slicing, *IEEE Transations on Software Engineering,* 10(4), pp 352-357, July 1984.
21. K. Yi and B.-M. Chang Exception analysis for Java, ECOOP Workshop on Formal Techniques for Java Programs , June 1999, Lisbon, Portugal.
22. K. Yi and S. Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. *Lecture Notes in Computer Science*, volume 1302, pages 98–113. Springer-Verlag, *Proceedings of the 4th Static Analysis Symposium*, September 1997.