Exception Analysis for Java

Kwangkeun Yi[1,⋆] and Byeong-Mo Chang[2,⋆⋆]

[1] kwang@cs.kaist.ac.kr
ROPAS⋆⋆⋆
Dept. of Computer Science
Korea Advanced Institute of Science & Technology
[2] chang@cs.sookmyung.ac.kr
Dept. of Computer Science
Sookmyung Women's University

Abstract. Current JDK Java compiler relies too much on programmer's specification for checking against uncaught exceptions of the input program. It is not elaborate enough to remove programmer's unnecessary handlers (when programmer's specifications are too many) nor suggest to programmers for specialized handlings (when programmer's specifications are too general).
We propose a static analysis of Java programs that estimates their exception flows independently of the programmer's specifications. This analysis is an extension of a class analysis to Java's exception mechanism. Its cost-effectiveness balance is suggested by sparsely analyzing the program at method-level (hence reducing the number of unknowns in the flow equations).

1 Introduction

The current Java compiler relies on the programmer's specifications to check that the input program will have no uncaught exceptions at run-time. The programmers have to declare in a method definition any exception class whose exceptions may escape from its body.

The problem is that the current compiler is not elaborate enough to do "better" than as specified by the programmers. It cannot avoid programmer's unnecessary handlers nor suggest to programmers for specialized handlings. It is foreseeable for careless (or inconfident) programmers to excessively declare at every method that some exceptions can be uncaught. Then every use of the method have to be wrapped with handlers, whose installation at run-time would

⋆ This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology.

⋆⋆ This work is partly supported by KISTEP project "Research on Basic and Applied Technology for Information Systems Security" and STEPI project "Highspeed Computing 2G-12".

⋆⋆⋆ Research On Program Analysis System (http://ropas.kaist.ac.kr), National Creative Research Initiative Center.

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

Fig. 1. Abstract Syntax of a Core of Java

be useless. Similarly, programmers can specify exceptions in too a broad sense. Programmers can declare that a method throws exceptions of the most general class `Exception` even if the actual exceptions are of much lower, specific classes. Then its handler cannot offer proper treatments specific to the exact classes of actual exceptions.

We propose a static analysis of Java programs that estimates their exception flows independently of the programmer's specifications. This analysis is an extension of a class analysis[1] to Java's exception mechanism. The class analysis estimates for each expression e_1 at method call $e_1.m(e_2)$ the classes S to which the method m belongs. The classes of uncaught exceptions from this call is then the classes of exceptions that can be raised *and* unhandled during the execution of $c.m$'s body for every class c in S.

2 Language

For presentation brevity we consider an imaginary core of Java with its exception constructs. 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 right hand side expression. Sequence expression returns the object of the last expression in the sequence. A method call returns

[1] You can consider our class analysis a simplified version of DeFouw et. al's analysis[3].

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 raised 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 raised exception is not covered by class c then the raised 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. The exception object e_0 is raised by `throw` e_0.

We assume that (1) class inheritance is explicitly expanded. That is, subclass's body has all the inheritted parts from its super-classes. (2) For the self object's field varialbe x is always explicitly prefixed as `this.`x. Variable x without the prefix is only for method parameter or handler variable x in "`try` e `catch` (c x e)." (3) all variables are distinct.

3 Uncaught Exception Analysis

We present our exception analysis in the set-constraint framework[4]. Every expression e of the program has two set constraints: $\mathcal{X}_e \supseteq se, \mathcal{P}_e \supseteq se$. The $\mathcal{X}_e$ is for the object classes that the expression e's normal object belongs to. The $\mathcal{P}_e$ is for the exception classes that the expression e's uncaught exception belongs to. The meaning of a set constrainst $\mathcal{X} \supseteq se$ 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. 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 [4]. Our implementation computes the solution by the conventional iterative fixpoint method because our solution space is finite: exception classes in the program. Correctness proofs are done by the fixpoint induction over the continuous functions that are derived [1] from our constraint system.

In Section 3.1 we present a constraint system that analyzes uncaught exceptions from *every* expression of the input program. Because exception-related expressions are sparse in programs, generating constraints for every expression is wasteful. The analysis cost-accuracy balance need to be addressed by enlarging the analysis granularity. Hence in Section 3.2 we present a sparse constraint system that analyzes uncaught exceptions at a larger granularity than at every expression. Similar technique of enlarging constraint granularity has already been successfuly used in ML [5]'s exception analysis [6]. Our analysis result is the solution of this sparse constraints.

3.1 Exception Analysis at Expression-Level

Figure 2 has the rules to generate set constraints for the object classes of *every* expression. The subscript e of set variables $\mathcal{X}_e$ and $\mathcal{P}_e$ denotes the current expression to which the rule applies. The relation "$\triangleright e\ : \mathcal{C}$" is read "constraints $\mathcal{C}$ are generated from expression e."

Consider the rule for method call:

$$[\text{MethCall}]\ \frac{\triangleright e_1 : \mathcal{C}_1 \quad \triangleright e_2 : \mathcal{C}_2}{\begin{array}{r}\triangleright\ e_1.m(e_2) : \{\mathcal{X}_x \supseteq \mathcal{X}_{e_2}, \mathcal{X}_e \supseteq \mathcal{X}_{c.m} | c \in \mathcal{X}_{e_1}, \texttt{method}\ m\texttt{(}x\texttt{)}\ \texttt{=}\ e_m \in c\} \\ \cup\{\mathcal{P}_e \supseteq \mathcal{P}_{c.m} | c \in \mathcal{X}_{e_1}, \texttt{method}\ m\texttt{(}x\texttt{)}\ \texttt{=}\ e_m \in c\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

The call expression will have the objects returned from the method m. This method $m\texttt{(}x\texttt{)}\ \texttt{=}\ e_m$ is the one defined inside the classes $c \in \mathcal{X}_{e_1}$ of e_1's objects. Hence $\mathcal{X}_e \supseteq \mathcal{X}_{c.m}$, and similary, $\mathcal{P}_e \supseteq \mathcal{P}_{c.m}$ for uncaught exceptions. (The subscript $c.m$ indicates the index for the body expression of class c's method m.) The constraint $\mathcal{X}_x \supseteq \mathcal{X}_{e_2}$ is for parameter binding: object of e_2 is passed to the method parameter x.

Consider the rule for `throw` expression:

$$[\text{Throw}]\ \frac{\triangleright e_1 : \mathcal{C}_1}{\triangleright\ \texttt{throw}\, e_1 : \{\mathcal{P}_e \supseteq \mathcal{X}_{e_1} \cup \mathcal{P}_{e_1}\} \cup \mathcal{C}_1}$$

It throws exceptions e_1 or, prior to throwing, it can have uncaught exceptions from inside e_1 too.

Consider the rule for `try` expression:

$$[\text{Try}]\ \frac{\triangleright e_0 : \mathcal{C}_0 \quad \triangleright e_1 : \mathcal{C}_1}{\triangleright\ \texttt{try}\, e_0\ \texttt{catch}\ \texttt{(}c_1\ \ x_1\ \ e_1\texttt{)} : \begin{array}{l}\{\mathcal{X}_e \supseteq \mathcal{X}_{e_0} \cup \mathcal{X}_{e_1}, \mathcal{X}_{x_1} \supseteq \mathcal{P}_{e_0} \dot{\cap} \{c\}\} \\ \cup\{\mathcal{P}_e \supseteq (\mathcal{P}_{e_0} \dot{-} \{c_1\}) \cup \mathcal{P}_{e_1}\} \cup \mathcal{C}_0 \cup \mathcal{C}_1\end{array}}$$

Normal objects are either from e_0 or from e_1 (after handling), hence $\mathcal{X}_e \supseteq \mathcal{X}_{e_0} \cup \mathcal{X}_{e_1}$. Raised exceptions from e_0 can be catched by x_1 only when their classes are covered by c_1. After this catching, exceptions can also be raised during the handling inside e_1. Hence, $\mathcal{P}_e \supseteq (\mathcal{P}_{e_0} \dot{-} \{c_1\}) \cup \mathcal{P}_{e_1}$.

The operators $\dot{-}$ and $\dot{\cap}$ are corresponding set-operations (set difference and intersection) modulo class hierarchy. We can easily catch the meaning by the following examples:

$$\{c\} \dot{-} \{c'\} = \begin{cases} \emptyset & \text{if } c = c' \text{ or } c \text{ is a subclass of } c' \\ \{c\} & \text{otherwise} \end{cases}$$

$$\{c\} \dot{\cap} \{c'\} = \begin{cases} \{c\} & \text{if } c = c' \text{ or } c \text{ is a subclass of } c' \\ \{c'\} & \text{if } c' \text{ is a subclass of } c \\ \{\} & \text{otherwise} \end{cases}$$

3.2 Exception Analysis at Method-Level

In our new, sparse constraint system, only four groups of set variables are considered: set variables for class' methods, field variables, try-expressions, and catch-variables. The number of unknowns is thus proportional only to the number of methods, field variables, and `try` expressions, not to the total number of expressions. For each method f, set variable $\mathcal{X}_f$ is for classes (including exception classes) that are "available" at f, and $\mathcal{P}_f$ is for classes of uncaught exceptions during the call to f. Similarly $\mathcal{X}_x$ for each field variable x. Every catch-variable x of `try` expressions:

$$\texttt{try}\ e_g\ \texttt{catch}\ (c\ x\ e)$$

has also a separate set variable $\mathcal{X}_x$ which will has the classes of uncaught exceptions from e_g. The try-expression e_g also has separate set variables $\mathcal{X}_g$ and $\mathcal{P}_g$, which are respectively for available and uncaught exception classes in e_g.

Figure 3 shows this new constraint system. The left-hand-side f in relation $f \rhd e : \mathcal{C}$ indicates that the expression e is a sub-expression of method f (or try-expression f).

Consider the rule for `throw` expression:

$$[\text{Throw}]_m\ \frac{f \rhd e_1 : \mathcal{C}_1}{f \rhd \texttt{throw}\ e_1 : \{\mathcal{P}_f \supseteq \mathcal{X}_f \cap \mathit{ExnClasses}\} \cup \mathcal{C}_1}$$

The classes $\mathcal{P}_f$ of uncaught exceptions from method f are the exception classes $(\mathcal{X}_f \cap \mathit{ExnClasses})$ among the classes $\mathcal{X}_f$ available at f.

Consider the rule for `try` expression:

$$[\text{Try}]_m\ \frac{g \rhd e_g : \mathcal{C}_g \quad f \rhd e_1 : \mathcal{C}_1}{f \rhd \texttt{try}\ e_g\ \texttt{catch}\ (c_1 x_1 e_1) : \{\mathcal{X}_{x_1} \supseteq \mathcal{P}_g \mathbin{\dot\cap} \{c_1\}\} \cup \mathcal{C}_g \cup \mathcal{C}_1}$$

The exceptions bound to the handler variable x_1 are the uncaught exceptions $\mathcal{P}_g$ from e_g if the exception's classes are covered by c, hence $\mathcal{X}_{x_1} \supseteq \mathcal{P}_g \mathbin{\dot\cap} \{c_1\}$. Note that the constraints $\mathcal{C}_1$ from the try-expression e_g are derived under g.

The least model of the sparse constraints $\mathcal{C}$, which are derived ($\rhd\ \mathit{pgm} : \mathcal{C}$) from an input program pgm is our analysis result. The solutions for $\mathcal{P}_m$ has the exception classes whose exceptions might be thrown and uncaught during m's execution.

3.3 Typeful Constraints for Improved Accuracy

The method-level analysis' accuracy can be improved using types. For example, the constraint rule $[\text{MethCall}]_m$ for method call $e_1.m(e_2)$ can be sharpened using m's type: $c_1 \to c_2$:

$$\frac{f \rhd e_1 : \mathcal{C}_1 \quad f \rhd e_2 : \mathcal{C}_2}{f \rhd e_1\texttt{.}m\texttt{(}e_2\texttt{)} : \begin{array}{l}\{\mathcal{X}_f \supseteq \mathcal{X}_{c.m} \mathbin{\dot\cap} \{c_2\}, \mathcal{X}_{c.m} \supseteq \mathcal{X}_f \mathbin{\dot\cap} \{c_1\}, \mathcal{P}_f \supseteq \mathcal{P}_{c.m} | c \in X_f, c.m : c_1 \to c_2\} \\ \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

If f's body has a method call $e_1\texttt{.}m\texttt{(}e_2\texttt{)}$, the classes $\mathcal{X}_f$ of available objects in f include the classes $\mathcal{X}_{c.m}$ of the objects available at the called method $c.m$ only if the classes are covered by m's return type. Similarly, method $c.m$ receives objects from current method f via the parameter passing only if their classes are covered by m's parameter type.

4 Discussion

Though we cannot claim the cost-effectiveness of our sparse analysis (Section 3.2) until its implementation [2] is tested with realistic Java programs, our experience of similarly developing a sister analysis [6, 7] for ML programs makes us positive about our design decision. Because exceptions are sparse objects in Java programs our gross estimation at the method-level (e.g., the $[\text{Throw}]_m$ and the $[\text{MethCall}]_m$ rules in Figure 3) would rarely be exposed in the analysis accuracy. First, an exception to raise is usually constructed (by `new` expression) inside the method that raises it. Second, method to call is usually explicit at the call-site. Even for dynamic-binding cases the situation that methods are exception-raising and also overridden in a class hierarchy may not be frequent.

References

1. Patrick Cousot and Radhia Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. In *Lecture Notes in Computer Science*, volume 939, pages 293–308. Springer-Verlag, proceedings of the 7th international conference on computer-aided verification edition, 1995.
2. J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. *OOPSLA'96 Conference Proceedings.*
3. G. DeFouw, D. Grove, and C. Chambers, Fast interprocedural class analysis, *Proceedings of 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* pages 222-236, Januaray 1998.
4. N. Heintze, Set-based program analysis. Ph.D thesis, Carnegie Mellon University, October 1992.
5. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.
6. Kwangkeun Yi and Sukyoung Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Lecture Notes in Computer Science*, volume 1302, pages 98–113. Springer-Verlag, proceedings of the 4th international static analysis symposium edition, 1997.
7. Kwangkeun Yi and Sukyoung Ryu. SML/NJ Exception Analysis version 0.98. `http://compiler.kaist.ac.kr/pub/exna/`, December 1998.

[2] We are currently implementing our analysis inside the Vortex compiler [2].

[New] $$\triangleright \texttt{new}\, c : \{\mathcal{X}_e \supseteq \{c\}\}$$

[This] $$\frac{c \text{ is the enclosing class}}{\triangleright \texttt{this} : \{\mathcal{X}_e \supseteq \{c\}\}}$$

[FieldAss] $$\frac{\triangleright\, e_1 : \mathcal{C}_1}{\triangleright\, id\,.x \;:=\; e_1 : \{\mathcal{X}_{c.x} \supseteq \mathcal{X}_{e_1} | c \in \mathcal{X}_{id}\} \cup \{\mathcal{X}_e \supseteq \mathcal{X}_{e_1}, \mathcal{P}_e \supseteq \mathcal{P}_{e_1}\} \cup \mathcal{C}_1}$$

[ParamAss] $$\frac{\triangleright\, e_1 : \mathcal{C}_1}{\triangleright\, x \;\texttt{:=}\; e_1 : \{\mathcal{X}_x \supseteq \mathcal{X}_{e_1}, \mathcal{X}_e \supseteq \mathcal{X}_{e_1}, \mathcal{P}_e \supseteq \mathcal{P}_{e_1}\} \cup \mathcal{C}_1}$$

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

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

[FieldVar] $$\frac{\triangleright\, id : \mathcal{C}_{id}}{\triangleright\, id.x : \{\mathcal{X}_e \supseteq \mathcal{X}_{c.x} | c \in \mathcal{X}_{id}\} \cup \mathcal{C}_{id}}$$

[Param] $$\triangleright x : \emptyset$$

[Throw] $$\frac{\triangleright\, e_1 : \mathcal{C}_1}{\triangleright\, \texttt{throw}\; e_1 : \{\mathcal{P}_e \supseteq \mathcal{X}_{e_1} \cup \mathcal{P}_{e_1}\} \cup \mathcal{C}_1}$$

[Try] $$\frac{\triangleright\, e_0 : \mathcal{C}_0 \quad \triangleright\, e_1 : \mathcal{C}_1}{\begin{array}{l}\triangleright\, \texttt{try}\; e_0 \;\texttt{catch}\; (c_1\;\; x_1\;\; e_1) : \{\mathcal{X}_e \supseteq \mathcal{X}_{e_0} \cup \mathcal{X}_{e_1}, \mathcal{X}_{x_1} \supseteq \mathcal{P}_{e_0} \dot{\cap} \{c\}\} \\ \qquad\qquad\qquad\qquad\qquad \cup \{\mathcal{P}_e \supseteq (\mathcal{P}_{e_0} \dot{-} \{c_1\}) \cup \mathcal{P}_{e_1}\} \cup \mathcal{C}_0 \cup \mathcal{C}_1\end{array}}$$

[MethCall] $$\frac{\triangleright\, e_1 : \mathcal{C}_1 \quad \triangleright\, e_2 : \mathcal{C}_2}{\begin{array}{l}\triangleright\; e_1\,.m(e_2) : \{\mathcal{X}_x \supseteq \mathcal{X}_{e_2}, \mathcal{X}_e \supseteq \mathcal{X}_{c.m} | c \in \mathcal{X}_{e_1}, \texttt{method}\;\; m(x) \;\texttt{=}\; e_m \in c\} \\ \qquad\qquad \cup \{\mathcal{P}_e \supseteq \mathcal{P}_{c.m} | c \in \mathcal{X}_{e_1}, \texttt{method}\;\; m(x) \;\texttt{=}\; e_m \in c\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

[MethDef] $$\frac{\triangleright\, e_m : \mathcal{C}}{\triangleright\, \texttt{method}\;\; m(x) \;\texttt{=}\; e_m : \{\mathcal{X}_{c.m} \supseteq \mathcal{X}_{e_m},\; \mathcal{P}_{c.m} \supseteq \mathcal{P}_{e_m}\} \cup \mathcal{C}}$$

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

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

Fig. 2. Exception Analysis at Expression-Level

$$[\text{New}]_m \qquad f \;\triangleright\; \texttt{new}\; c \;:\{\mathcal{X}_f \supseteq \{c\}\} \qquad [\text{This}]_m \qquad \frac{\text{c is the enclosing class}}{f \;\triangleright\; \texttt{this} \;:\{\mathcal{X}_f \supseteq \{c\}\}}$$

$$[\text{FieldAss}]_m \qquad \frac{f \;\triangleright\; e_1 \;:\mathcal{C}_1}{f \;\triangleright\; id.x \;\texttt{:=}\; e_1 \;:\{\mathcal{X}_{c.x} \supseteq \mathcal{X}_f | c \in \mathcal{X}_f\} \cup \mathcal{C}_1}$$

$$[\text{ParamAss}]_m \qquad \frac{f \;\triangleright\; e_1 \;:\mathcal{C}_1}{f \;\triangleright\; x \;\texttt{:=}\; e_1 \;:\mathcal{C}_1} \qquad [\text{Seq}]_m \qquad \frac{f \;\triangleright\; e_1 \;:\mathcal{C}_1 \quad f \;\triangleright\; e_2 \;:\mathcal{C}_2}{f \;\triangleright\; e_1\texttt{;}e_2 \;:\mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\text{Cond}]_m \qquad \frac{f \;\triangleright\; e_0 \;:\mathcal{C}_0 \quad f \;\triangleright\; e_1 \;:\mathcal{C}_1 \quad f \;\triangleright\; e_2 \;:\mathcal{C}_2}{f \;\triangleright\; \texttt{if}\; e_0 \;\texttt{then}\; e_1 \;\texttt{else}\; e_2 \;:\mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\text{FieldVar}]_m \qquad \frac{f \;\triangleright\; id \;:\mathcal{C}_1}{f \;\triangleright\; id\texttt{.}x \;:\{\mathcal{X}_f \supseteq \mathcal{X}_{c.x} | c \in \mathcal{X}_f\} \cup \mathcal{C}_1}$$

$$[\text{Param}]_m \qquad f \;\triangleright\; x \;:\emptyset$$

$$[\text{Throw}]_m \qquad \frac{f \;\triangleright\; e_1 \;:\mathcal{C}_1}{f \;\triangleright\; \texttt{throw}\; e_1 \;:\{\mathcal{P}_f \supseteq \mathcal{X}_f \cap \mathit{ExnClasses}\} \cup \mathcal{C}_1}$$

$$[\text{Try}]_m \qquad \frac{g \;\triangleright\; e_g \;:\mathcal{C}_g \quad f \;\triangleright\; e_1 \;:\mathcal{C}_1}{f \;\triangleright\; \texttt{try}\; e_g \;\texttt{catch}\; \texttt{(}c_1 x_1 e_1\texttt{)} \;:\{\mathcal{X}_{x_1} \supseteq \mathcal{P}_g \,\dot{\cap}\, \{c_1\}\} \cup \mathcal{C}_g \cup \mathcal{C}_1}$$

$$[\text{MethCall}]_m \qquad \frac{f \;\triangleright\; e_1 \;:\mathcal{C}_1 \quad f \;\triangleright\; e_2 \;:\mathcal{C}_2}{f \;\triangleright\; e_1\texttt{.}m\texttt{(}e_2\texttt{)} \;:\{\mathcal{X}_f \supseteq \mathcal{X}_{c.m}, \mathcal{P}_f \supseteq \mathcal{P}_{c.m}, \mathcal{X}_{c.m} \supseteq \mathcal{X}_f | c \in X_f\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\text{MethDef}]_m \qquad \frac{m \;\triangleright\; e_m \;:\mathcal{C}_m}{\triangleright \texttt{method}\; m\texttt{(}x\texttt{)} \;\texttt{=}\; e_m \;:\mathcal{C}_m}$$

$$[\text{ClassDef}]_m \qquad \frac{\triangleright M_i \;:\mathcal{C}_i \quad i = 1, \cdots, m}{\triangleright \texttt{class}\; c \;\texttt{= \{var}\; x_1 \cdots x_n \; M_1 \cdots M_m \texttt{\}} \;:\mathcal{C}_1 \cup \cdots \cup \mathcal{C}_m}$$

$$[\text{Program}]_m \qquad \frac{\triangleright C_i \;:\mathcal{C}_i \quad i = 1, \cdots, n}{\triangleright C_1 \cdots C_n \;:\mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

Fig. 3. Exception Analysis at Method-Level