Granularity of Constraint-based Analysis for Java [*]

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

Jangwu Jo
Department of Computer Engineering
Pusan University of Foreign Studies
Pusan, Korea
jjw@taejo.pufs.ac.kr

ABSTRACT

This paper proposes a transformation-based approach to design constraint-based analyses for Java at a coarser granularity. In this approach, we design a less or equally precise but more efficient version of an original analysis by transforming the original construction rules into new ones. As applications of this rule transformation, we provide two instances of analysis design by rule-transformation. The first one designs a sparse version of class analysis for Java and the second one deals with a sparse exception analysis for Java. Both are designed based on method-level, and the sparse exception analysis is shown to give the same information for every method as the original analysis.

Keywords

constraint-based analysis, set constraints, construction rules, partition function

1. INTRODUCTION

Constraint-based analysis is a static analysis framework that is applicable to functional, logic and object-oriented programming languages [5, 12, 13, 10]. In constraint-based analysis framework, a specific analysis is designed in terms of set-constraint construction rules.Constraint-based analysis first constructs set-constraints for input programs using the construction rules, and then computes the solution or model of them.

The precision of the analysis depends upon the choice of the finite set of indices of set-variables. We usually design an analysis theoretically at expression-level, which has one set-variable(or index) for every expression. However, the efficiency of expression-level analyses may not be satisfactory for large practical programs [21, 19]. In addition, some analyses (like side-effect analysis [15], exception analysis [20] and synchronization analysis [11]) are not interested in properties of all expressions. So, it can be wasteful to define one set-variable for every expression for this kind of analyses. Hence, the analysis cost-effectiveness need to be addressed by enlarging the analysis granularity.

This paper proposes a transformation-based approach to design constraint-based analyses for Java at a coarser granularity. In this approach, we can design a less or equally precise but more efficient version of an original analysis by *transforming* the original construction rules into new ones. This is done by two steps. The first is to define or design an index determination rule for a new sparse analysis based on some syntactic properties, so that it can partition the original indices. The second is to transform the original construction rules into new ones by replacing the original index of each set variable by the new index.

As applications of this rule-transformation approach, we provide two instances of analysis design by rule-transformation. The first one designs a sparse version of class analysis for Java and the second one deals with a sparse exception analysis for Java. Both are designed based on method-level, and the sparse exception analysis is shown to give the *same* information for each method as the original analysis.

There have been several research directions to improve efficiency of analysis. The first direction is to improve analysis time by simplifying set constraints after constructing the whole constraints [6, 9, 10, 18]. They usually simplify set constraints without losing the precision of the original analysis. The second direction is to design analyses at a coarser granularity. Several constraint-based analyses including CFA and exception analysis are also designed manually at a coarser granularity, experimented and practically applied in [20, 21, 19]. This basic idea is also addressed in data flow analysis and abstract interpretation [2, 4, 14, 17].

The contribution of this paper is to provide a systematic mechanism or bridge to design practical constraint-based analyses for Java by rule transformation. Moreover, this paper provides general soundness proof for them.

Section 2 presents a core of Java, and basic definitions for constraint-based analysis. Section 3 presents class analysis for Java.Section 4 presents a systematic mechanism to design analyses by rule transformation. Section 5 presents some applications of this rule transformation. Section 6 discusses related works and Section 7 concludes this paper.

2. PRELIMINARIES

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

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

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

Figure 1: Abstract Syntax of a Core of Java

Syntax of set expressions:

se	$::=$	$\mathcal{X}_i$	set variables
	$\mid$	c	class names
	$\mid$	$app(\mathcal{X}_i, m, \mathcal{X}_i)$	sets from method call
	$\mid$	$se \cup se$	sets from conditionals
	$\mid$	$se - \{c_1, \cdots, c_n\}$	sets from `try-catch`s
	$\mid$	$\top$	universe set

Semantics of set expressions:

$$\begin{array}{rcl}
\mathcal{I}(\mathcal{X}_i) & \subseteq & Val \\
\mathcal{I}(\top) & = & Val \\
\mathcal{I}(c) & = & \{c\} \\
\mathcal{I}(app(\mathcal{X}_1, m, \mathcal{X}_2)) & = & \{v \mid c \in \mathcal{I}(\mathcal{X}_1), m(x) = e_m \in c, \\
 & & \quad v \in \mathcal{I}(\mathcal{X}_{c.m}), \mathcal{I}(\mathcal{X}_x) \supseteq \mathcal{I}(\mathcal{X}_2)\} \\
\mathcal{I}(se_1 \cup se_2) & = & \mathcal{I}(se_1) \cup \mathcal{I}(se_2) \\
\mathcal{I}(se_1 - \{c_1, \cdots, c_n\}) & = & \mathcal{I}(se_1) - \{c_1, \cdots, c_n\}
\end{array}$$

Figure 2: Set constraints : syntax and semantics

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 the object from the method body. The `try-catch` expression

`try` e_0 `catch (`c x e_1`)`

evaluates the `try`-block e_0 first. If the expression returns a normal object then this object is the result of the `try-catch` 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-catch` 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 raised by `throw` e_0. The programmers have to declare in a method definition any exception classes 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.

We omit the formal semantics of the core language. Its operational semantics should be straightforward, not much different from existing works [8, 16].

Constraint-based analysis consists of two phases [12]: collecting set constraints and solving them. The first phase constructs 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 by the solving rules. A solution is an assignment from set variables in the constraints to the finite descriptions of such sets of values.

Each set constraint is of the form $\mathcal{X} \supseteq se$ where $\mathcal{X}$ is a set variable and se is a set expression. The constraint indicates that the set variable $\mathcal{X}$ must contain the set se. In case of class analysis, the set expression is of the form

$$se \to c|\mathcal{X}|app(\mathcal{X}_1, m, \mathcal{X}_2)|se \cup se|se - \{c_1, ..., c_n\}$$

where c's are class names. Multiple constraints are conjunctions. We write $\mathcal{C}$ for a finite collection of set constraints. Semantics of set expressions naturally follows from their corresponding language constructs. For example, $app(\mathcal{X}_1, m, \mathcal{X}_2)$ represents the classes of objects returned from applications of method m to objects in $\mathcal{X}_1$ with parameters in $\mathcal{X}_2$. The formal semantics of set expressions is defined by an interpretation $\mathcal{I}$ that maps from set expressions to sets of values (see Figure 2). 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 $\mathcal{C}$, $\mathcal{I}(\mathcal{X}) \supseteq \mathcal{I}(se)$.

Our static analysis is defined to be the least model 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 [12]. We write $lm(\mathcal{C})$ for the least model of a collection $\mathcal{C}$ of constraints.

3. CLASS ANALYSIS

We first present a class analysis for Java based on set-based analysis framework [12]. Every expression e of the program has one set constraints: $\mathcal{X}_e \supseteq se$. The set variable $\mathcal{X}_e$ is for the classes that the expression e's normal object belongs to. Figure 3 has the rules to construct set constraints for the classes of the objects of each expression e. The subscript e of a set variable $\mathcal{X}_e$ denotes the current expression to which the rule applies. The subscript $c.x$ denote a field variable x defined in a class c and $c.m$ denotes a method m defined in a class c. The relation "$e \rhd_1 \mathcal{C}$" is read "constraints $\mathcal{C}$ are generated from an expression e."

Consider the rules in Figure 3. The `new` expression will have the newly created object from the class c, hence $\mathcal{X}_e \supseteq \{c\}$. The conditional expression will have the objects from e_1 or e_2, hence $\mathcal{X}_e \supseteq \mathcal{X}_{e_1} \cup \mathcal{X}_{e_2}$.

Consider the rule for method call:

$$\frac{e_1 \rhd_1 \mathcal{C}_1 \quad \rhd e_2 \rhd_1 \mathcal{C}_2}{e_1 .m(e_2) \rhd_1 \{\mathcal{X}_e \supseteq app(\mathcal{X}_{e_1}, m, \mathcal{X}_{e_2})\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[New$_1$] $\texttt{new}\,c \rhd_1 \{\mathcal{X}_e \supseteq \{c\}\}$ [This$_1$] $$\frac{c \text{ is the enclosing class}}{\texttt{this} \rhd_1 \{\mathcal{X}_e \supseteq \{c\}\}}$$

[FieldAss$_1$] $$\frac{e_1 \rhd_1 \mathcal{C}_1 \quad id \rhd_1 \mathcal{C}_{id}}{id.x := e_1 \rhd_1 \{\mathcal{X}_{c.x} \supseteq \mathcal{X}_{e_1} \mid c \in \mathcal{X}_{id}, x \in c\} \cup \{\mathcal{X}_e \supseteq \mathcal{X}_{e_1}\} \cup \mathcal{C}_1}$$

[ParamAss$_1$] $$\frac{e_1 \rhd_1 \mathcal{C}_1}{x \texttt{ := } e_1 \rhd_1 \{\mathcal{X}_x \supseteq \mathcal{X}_{e_1}, \mathcal{X}_e \supseteq \mathcal{X}_{e_1}\} \cup \mathcal{C}_1}$$

[Seq$_1$] $$\frac{e_1 \rhd_1 \mathcal{C}_1 \quad e_2 \rhd_1 \mathcal{C}_2}{e_1 \texttt{;} e_2 \rhd_1 \{\mathcal{X}_e \supseteq \mathcal{X}_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[Cond$_1$] $$\frac{e_0 \rhd_1 \mathcal{C}_0 \quad e_1 \rhd_1 \mathcal{C}_1 \quad e_2 \rhd \mathcal{C}_2}{\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rhd_1 \{\mathcal{X}_e \supseteq \mathcal{X}_{e_1} \cup \mathcal{X}_{e_2}\} \cup \mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[FieldVar$_1$] $$\frac{id \rhd_1 \mathcal{C}_{id}}{id.x \rhd_1 \{\mathcal{X}_e \supseteq \mathcal{X}_{c.x} \mid c \in \mathcal{X}_{id}, x \in c\} \cup \mathcal{C}_{id}}$$

[Param$_1$] $$x \rhd_1 \mathcal{X}_e \supseteq \mathcal{X}_x$$

[Throw$_1$] $$\frac{e_1 \rhd_1 \mathcal{C}_1}{\texttt{throw}\ e_1 \rhd_1 \mathcal{C}_1}$$

[Try$_1$] $$\frac{e_0 \rhd_1 \mathcal{C}_0 \quad \rhd\ e_1 \rhd_1 \mathcal{C}_1}{\texttt{try}\ e_0\ \texttt{catch}(c_1\ x_1\ e_1) \rhd_1 \{\mathcal{X}_e \supseteq \mathcal{X}_{e_0} \cup \mathcal{X}_{e_1}, \mathcal{X}_{x_1} \supseteq \{c_1\}^*\} \cup \mathcal{C}_0 \cup \mathcal{C}_1}$$

[MethCall$_1$] $$\frac{e_1 \rhd_1 \mathcal{C}_1 \quad e_2 \rhd_1 \mathcal{C}_2}{e_1.m(e_2) \rhd_1 \{\mathcal{X}_e \supseteq app(\mathcal{X}_{e_1}, m, \mathcal{X}_{e_2})\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[MethDef$_1$] $$\frac{e_m \rhd_1 \mathcal{C}}{m(x)\texttt{=}e_m \rhd_1 \{\mathcal{X}_{c.m} \supseteq \mathcal{X}_{e_m}\} \cup \mathcal{C}}$$

[ClassDef$_1$] $$\frac{m_i \rhd_1 \mathcal{C}_i\ i = 1, \cdots, n}{\texttt{class}\ c\ \texttt{= \{var}\ x_1, \cdots, x_k, m_1, \cdots, m_n\texttt{\}} \rhd_1 \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

[Program$_1$] $$\frac{C_i \rhd_1 \mathcal{C}_i\ i = 1, \cdots, n}{C_1, \cdots, C_n \rhd_1 \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

Figure 3: Class analysis at expression-level

$$\frac{\mathcal{X} \supseteq \mathcal{X}_1 \cup \mathcal{X}_2}{\mathcal{X} \supseteq \mathcal{X}_1} \qquad \frac{\mathcal{X} \supseteq \mathcal{X}_1 \cup \mathcal{X}_2}{\mathcal{X} \supseteq \mathcal{X}_2} \qquad \frac{\mathcal{X} \supseteq \mathcal{Y} \quad \mathcal{Y} \supseteq ae}{\mathcal{X} \supseteq ae}$$

$$\frac{\mathcal{X} \supseteq app(\mathcal{X}_1, m, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq c \quad m(x)\texttt{=}e \in c}{\mathcal{X} \supseteq \mathcal{X}_{c.m} \quad \mathcal{X}_x \supseteq \mathcal{X}_2}$$

Figure 4: Rules S for solving set constraints

The call expression will have the objects returned from the method m. This method $m(x)$ = e_m is the one defined inside the classes $c \in \mathcal{X}_{e_1}$ of e_1's objects. Hence, in the solving phase, $\mathcal{X}_e \supseteq \mathcal{X}_{c.m}$ is added by the solving rule in Figure 4, and the constraint $\mathcal{X}_x \supseteq \mathcal{X}_{e_2}$ is also added for parameter binding.

Consider the rule for `try` expression:

$$\frac{e_0 \rhd_1 \mathcal{C}_0 \quad \rhd\ e_1 \rhd_1 \mathcal{C}_1}{\texttt{try}\ e_0\ \texttt{catch}(c_1\ x_1\ e_1) \rhd_1 \begin{array}{l}\{\mathcal{X}_e \supseteq \mathcal{X}_{e_0} \cup \mathcal{X}_{e_1}, \mathcal{X}_{x_1} \supseteq \{c_1\}^*\} \\ \cup\, \mathcal{C}_0 \cup \mathcal{C}_1\end{array}}$$

Normal objects are either from e_0 or from e_1 (after handling), hence $\mathcal{X}_e \supseteq \mathcal{X}_{e_0} \cup \mathcal{X}_{e_1}$. The objects of x_1 are raised exceptions from the `try`-block and can be approximated by all the subclasses (denoted by $\{c_1\}^*$) of its class c_1.

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 decomposes compound set constraints into smaller ones, which approximates the steps of the value flows between expressions. Consider the rule for method call result $\mathcal{X} \supseteq app(\mathcal{X}_1, m, \mathcal{X}_2)$:

$$\frac{\mathcal{X} \supseteq app(\mathcal{X}_1, m, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq c \quad m(x)\texttt{=}e \in c}{\mathcal{X} \supseteq \mathcal{X}_{c.m} \quad \mathcal{X}_x \supseteq \mathcal{X}_2}$$

It introduces $\mathcal{X} \supseteq \mathcal{X}_{c.m}$ if a method m to call is defined in a class c and if so, adds $\mathcal{X}_x \supseteq \mathcal{X}_2$ to simulate the parameter binding. Other rules are similarly straightforward from the semantics of corresponding set expressions.

Implementation can compute the solution by the conventional iterative fixpoint method because the solution space is finite: classes in the program. Correctness proofs can be done by the fixpoint induction over the continuous functions that are derived [5] from our constraint system.

4. RULE-TRANSFORMATION

In this section, we describe how to design an analysis at a coarser granularity by rule-transformation. We first define or design an index determination rule for a new sparse analysis based on some syntactic properties, so that it can partition the original indices, and then transform the original construction rules by applying the partition.

We represent index determination as an index function $I : Expr \cup Name \to Index$ where $Expr$ is a set of expressions, $Name$ is a set of the names of variables and methods, and $Index$ is a set of indices (natural numbers). We assume an original analysis is designed at expression-level, that is, one set-variable (or index) is defined for every expression and name. This index determination can be represented as an index function $I_E : Expr \cup Name \to Index$ where every expression and name is mapped to its unique index. In the following, because I_E is one-to-one, we abuse notation by denoting $\mathcal{X}_{I_E(e)}$ just by $\mathcal{X}_e$.

To design an analysis at a coarser granularity, we first need an index function to determine indices of set-variables. Instead of defining one set-variable for one expression, we can make one set variable (or index) for *a set* of expressions. One simple and extreme example is to make one index for all expressions in a program. That can be represented as an index function $I_P : Expr \cup Name \to Index$ where $I_P(e) = 1$ for every expression and name e. This index function is used in the rapid type analysis [1].

We can define an index function in terms of some syntactic properties. For example, we can design a class-level analysis by defining one index for each class.

Example 1. The index function $I_C : Expr \cup Name \to Index$ for a class-level analysis is defined as :

$I_C(c.m) = c$	if m is a method defined in a class c
$I_C(c.x) = c$	if x is a field variable defined in a class c
$I_C(e) = c$	if an expression e appears in a class c
$I_C(x) = c$	if x is a parameter of a method defined in a class c

where c denotes a class name or its unique index number. □

While every expression is mapped to its unique index in I_E, a set of expressions is mapped to one index in I_C, only if they appear in the same class. We generalize this idea by defining a *partition* as follows:

Definition 1. Let I_1 and I_2 be two index functions. I_2 is a *partition* of I_1 if there exist a function π such that $I_2 = \pi \circ I_1$, where π is called a *partition function* from I_1 to I_2.

It is easy to show that I_P and I_C are partitions of I_E.

If we have designed a new index function I for a sparse analysis such that $I = \pi \circ I_E$ for a partition function π, we can then transform the original construction rules by applying the partition function π to the original indices. The basic idea of this rule transformation is to replace the index of each set variable $\mathcal{X}_e$ in the original construction rules by the new index $\mathcal{X}_{\pi(e)}$. This rule-transformation can be formalized as follows:

Definition 2. Let I be an index function such that $I = \pi \circ I_E$. Consider a generic expression $e = \kappa(e_1, \cdots, e_n)$, where κ is a language construct. If r is a construction rule of the form :

$$\frac{e_1 : \mathcal{C}_1, \cdots, e_n : \mathcal{C}_n}{\kappa(e_1, \cdots, e_n) : \cup_{1 \le i \le n} \mathcal{C}_i \cup \{\mathcal{X}_e \supseteq se\}}$$

then, the transformed rule r/π by applying the partition function π is defined as:

$$\frac{e_1 : \mathcal{C}_1, \cdots, e_n : \mathcal{C}_n}{\kappa(e_1, \cdots e_n) : \cup_{1 \le i \le n} \mathcal{C}_i \cup \{\mathcal{X}_{\pi(e)} \supseteq se/\pi\}}$$

where se/π is obtained by replacing every set variable $\mathcal{X}_{e'}$ in se by $\mathcal{X}_{\pi(e')}$.

Semantics of analysis functions *app* needs to be changed after this rule transformation, since it introduces new set-variables while solving. This change will be reflected and implemented by transforming the corresponding solving rule (see the end of this section).

For example, we can design a class analysis at class-level in [19] by transforming the original rules in Figure 4.

Example 2. Let I_C be an index function for a class-level analysis and π be a partition function such that $I_C = \pi \circ I_E$. Assuming that e is the current expression to which the rule applies and e appears in a class c', we can design a class analysis at class-level by applying π to the original rules in Figure 3.

We first consider the construction rule for `new`-expression e in Figure 3 :

$$\texttt{new}\ c \rhd_1 \{\mathcal{X}_e \supseteq \{c\}\}$$

Since $\pi(e) = c'$,by applying π, the rule can be transformed into:

$$\texttt{new}\ c \rhd_2 \{\mathcal{X}_{c'} \supseteq \{c\}\}$$

In case of the rule for field variable access in Figure 3 :

$$\frac{id \rhd_1 \mathcal{C}_{id}}{id.x \rhd_1 \{\mathcal{X}_e \supseteq \mathcal{X}_{c.x} \mid c \in \mathcal{X}_{id}, x \in c\} \cup \mathcal{C}_{id}}$$

Since $\pi(id.x) = \pi(id) = c'$ and $\pi(c.x) = c$, the rule can be transformed into:

$$\frac{id \rhd_2 \mathcal{C}_{id}}{id.x \rhd_2 \{\mathcal{X}_{c'} \supseteq \mathcal{X}_c \mid c \in \mathcal{X}_{c'}, x \in c\} \cup \mathcal{C}_{id}}$$

Consider the rule for `if`-expression e in Figure 3 :

$$\frac{e_0 \rhd_1 \mathcal{C}_0 \quad \rhd e_1 \rhd_1 \mathcal{C}_1 \quad \rhd e_2 \rhd_1 \mathcal{C}_2}{\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rhd_2 \{\mathcal{X}_e \supseteq \mathcal{X}_{e_1} \cup \mathcal{X}_{e_2}\} \cup \mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

If this expression e appears in a class c', then e_1 and e_2 are also in c'. So, the rule can be transformed into :

$$\frac{e_0 \rhd_2 \mathcal{C}_0 \quad e_1 \rhd_2 \mathcal{C}_1 \quad e_2 \rhd_2 \mathcal{C}_2}{\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rhd_2 \{\mathcal{X}_{c'} \supseteq \mathcal{X}_{c'} \cup \mathcal{X}_{c'}\} \cup \mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

which can be simplified into :

$$\frac{e_0 \rhd_2 \mathcal{C}_0 \quad e_1 \rhd_2 \mathcal{C}_1 \quad e_2 \rhd_2 \mathcal{C}_2}{\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rhd_2 \mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

In case of the rule for method call e in Figure 3 :

$$\frac{e_1 \rhd_1 \mathcal{C}_1 \quad e_2 \rhd_1 \mathcal{C}_2}{e_1.m(e_2) \rhd_1 \{\mathcal{X}_e \supseteq app(\mathcal{X}_{e_1}, m, \mathcal{X}_{e_2})\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

since $\pi(e) = \pi(e_1) = \pi(e_2) = c'$, the rule can be transformed into :

$$\frac{e_1 \rhd_2 \mathcal{C}_1 \quad \rhd e_2 \rhd_2 \mathcal{C}_2}{e_1.m(e_2) \rhd_2 \{\mathcal{X}_{c'} \supseteq app_\pi(\mathcal{X}_{c'}, m, \mathcal{X}_{c'})\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

where app_π is a modified analysis function , and its semantics is defined by applying π as follows:

$$\begin{aligned}\mathcal{I}(app_\pi(\mathcal{X}_1, m, \mathcal{X}_2)) = \ & \{v \mid c \in \mathcal{I}(\mathcal{X}_1), m(x){=}e_m \in c, v \in \mathcal{I}(\mathcal{X}_c), \\ & \mathcal{I}(\mathcal{X}_c) \supseteq \mathcal{I}(\mathcal{X}_2)\}\end{aligned}$$

□

If a method call $e_1.m(e_2)$ is analyzed with this transformed rule, every method in the class will be considered for this method call. This type of analysis is manually designed and static type information is integrated in the analysis to refine the precision in [19].

We now can design a new sparse analysis by a set of the transformed rules. This can be formalized as follows:

$$[\mathrm{New}_3] \quad \texttt{new}\; c \rhd_3 \{\mathcal{X}_{c'.m'} \supseteq \{c\}\} \qquad [\mathrm{This}_3] \quad \frac{c \text{ is the enclosing class}}{\texttt{this} \rhd_3 \{\mathcal{X}_{c'.m'} \supseteq \{c\}\}}$$

$$[\mathrm{FieldAss}_3] \quad \frac{e_1 \rhd_3 \mathcal{C}_1 \qquad id \rhd_3 \mathcal{C}_{id}}{id.x := e_1 \rhd_3 \{\mathcal{X}_{c.x} \supseteq \mathcal{X}_{c'.m'} \mid c \in \mathcal{X}_{c'.m'}, x \in c\} \cup \mathcal{C}_1}$$

$$[\mathrm{ParamAss}_3] \quad \frac{e_1 \rhd_3 \mathcal{C}_1}{x \;\texttt{:=} e_1 \rhd_3 \{\mathcal{X}_{owner(x)} \supseteq \mathcal{X}_{c'.m'}\} \cup \mathcal{C}_1}$$

$$[\mathrm{Seq}_3] \quad \frac{e_1 \rhd_3 \mathcal{C}_1 \quad \rhd\; e_2 \rhd_3 \mathcal{C}_2}{e_1 \texttt{;} e_2 \rhd_3 \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\mathrm{Cond}_3] \quad \frac{e_0 \rhd_3 \mathcal{C}_0 \;\; e_1 \rhd_3 \mathcal{C}_1 \;\; e_2 \rhd_3 \mathcal{C}_2}{\texttt{if}\; e_0 \;\texttt{then}\; e_1 \;\texttt{else}\; e_2 \rhd_3 \mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\mathrm{FieldVar}_3] \quad \frac{id \rhd_3 \mathcal{C}_{id}}{id.x \rhd_3 \{\mathcal{X}_{c'.m'} \supseteq \mathcal{X}_{c.x} \mid c \in \mathcal{X}_{c'.m'}, x \in c\} \cup \mathcal{C}_{id}}$$

$$[\mathrm{Param}_3] \quad x \rhd_3 \mathcal{X}_{c'.m'} \supseteq \mathcal{X}_{owner(x)}$$

$$[\mathrm{Throw}_3] \quad \frac{e_1 \rhd_3 \mathcal{C}_1}{\texttt{throw}\; e_1 \rhd_3 \mathcal{C}_1}$$

$$[\mathrm{Try}_3] \quad \frac{e_0 \rhd_3 \mathcal{C}_0 \;\; e_1 \rhd_3 \mathcal{C}_1}{\texttt{try}\; e_0 \;\texttt{catch}(c_1\; x_1\; e_1) \rhd_3 \{\mathcal{X}_{c'.m'} \supseteq \{c_1\}^*\} \cup \mathcal{C}_0 \cup \mathcal{C}_1}$$

$$[\mathrm{MethCall}_3] \quad \frac{e_1 \rhd_3 \mathcal{C}_1 \;\; e_2 \rhd_3 \mathcal{C}_2}{e_1.m(e_2) \rhd_3 \{\mathcal{X}_{c'.m'} \supseteq app_\pi(\mathcal{X}_{c'.m'}, m, \mathcal{X}_{c'.m'})\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\mathrm{MethDef}_3] \quad \frac{e_m \rhd_3 \mathcal{C}}{m(x) = e_m \rhd_3 \mathcal{C}}$$

$$[\mathrm{ClassDef}_3] \quad \frac{m_i \rhd_3 \mathcal{C}_i,\; i = 1, \cdots, n}{\texttt{class}\; c \;\texttt{= \{var}\; x_1, \cdots, x_k, m_1, \cdots, m_n \texttt{\}} \rhd_3 \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

$$[\mathrm{Program}_3] \quad \frac{C_i \rhd_3 \mathcal{C}_i \; i = 1, \cdots, n}{C_1, \cdots, C_n \rhd_3 \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

Figure 5: Class analysis at method-level

Definition 3. Let R be a set of construction rules. The set R/π of transformed rules by a partition function π is defined as

$$R/\pi = \{r/\pi | r \in R\}$$

Now we describe constraints set-up with the transformed construction rules for a given program. Whenever a program variable or expression is encountered during constraints set-up, its set variable (or index) is determined by applying the partition function to it, and then set-constraints are constructed.

As in Example 2, semantics of an analysis function like *app* needs to be changed after transformation, since it introduces new set-variables while solving.

To reflect this change, the corresponding *solving rule* in Figure 4 must be transformed by applying the partition function, if it introduces new set-variables. If a solving rule introduces new set-variables, their indices must be determined by applying the partition function.

The solving rule for the function application must be transformed by applying π to the newly introduced set-variables as follows:

$$\frac{\mathcal{X} \supseteq app_\pi(\mathcal{X}_1, m, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq c \quad m(x){=}e \in c}{\mathcal{X} \supseteq \mathcal{X}_{\pi(c.m)} \quad \mathcal{X}_{\pi(x)} \supseteq \mathcal{X}_2}$$

The other rules are not changed.

Consider the solving rule for the function application for Example 2. The original solving rule can be transformed into :

$$\frac{\mathcal{X} \supseteq app_\pi(\mathcal{X}_1, m, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq c \quad m(x){=}e \in c}{\mathcal{X} \supseteq \mathcal{X}_c \quad \mathcal{X}_c \supseteq \mathcal{X}_2}$$

We denote by $R(p)$ (or $(R/\pi)(p)$) the set of set-constraints constructed by applying the construction rules in R (or R/π) to a program p. We can prove the soundness of the transformed construction rules by showing that the least model of the transformed constraints $R/\pi(p)$ is a sound approximation of the original constraints $R(p)$ for every program p. The proof is based on the observation in [5] that the least model $lm(\mathcal{C})$ is equivalent to the least fixpoint of the continuous function $\mathcal{F}$ derived from $\mathcal{C}$.

THEOREM 1. *Let p be a program, R be a set of construction rules, and π be a partition function. Let $\mathcal{C} = R(p)$ and $\mathcal{C}_\pi = R/\pi(p)$. Then, $lm(\mathcal{C}_\pi)(\mathcal{X}_{\pi(e)}) \supseteq lm(C)(\mathcal{X}_e)$ for every expression e*

Proof. See Appendix. □

5. APPLICATIONS

To show the usefulness of the rule-transformation, we provide two instances of analysis design by rule-transformation. The first one designs a sparse version of a class analysis and the second one deals with an exception analysis. Both are designed basically based on method-level and the sparse exception analysis gives the same information for each function as the original analysis.

5.1 Class analysis

We design a sparse class analysis by rule-transformation, where only two groups of set variables are considered: set-variables for methods and field variables. The number of set-variables is thus proportional only to the number of methods and fields, not to the number of expressions. This design decision can be represented by an index function as follows.

Definition 4. The index function $I_M : Expr \cup Name \rightarrow Index$ for a method-level analysis is defined as :

$I_M(c.m) = c.m$	if m is a method defined in a class c
$I_M(c.x) = c.x$	if x is a field variable defined in a class c
$I_M(x) = c.m$	if x is a parameter of a method m defined in a class c
$I_M(e) = c.m$	if e appears in a method m defined in a class c

where $c.m$ and $c.x$ indicates the indices for the class c's method m and field x.

Let I_M be an index function for a method-level analysis and π be a partition function such that $I_M = \pi \circ I_E$. Assuming that the current expression e appears in a method m' defined in a class c',i.e.$\pi(e) = c'.m'$, we can design a class analysis at method-level in Figure 5 by applying π to the original rules in Figure 3.

In case of the rule for field variable access in Figure 3 :

$$\frac{id \rhd_1 \mathcal{C}_{id}}{id.x \rhd_1 \{\mathcal{X}_e \supseteq \mathcal{X}_{c.x} | c \in \mathcal{X}_{id}\} \cup \mathcal{C}_{id}}$$

Since $\pi(id.x) = c'.m'$ and $\pi(c.x) = c.x$, this rule can be transformed into:

$$\frac{id \rhd_3 \mathcal{C}_{id}}{id.x \rhd_3 \{\mathcal{X}_{c'.m'} \supseteq \mathcal{X}_{c.x} | c \in \mathcal{X}_{c.m}, x \in c\} \cup \mathcal{C}_{id}}$$

In case of the rule for method call e in Figure 3 :

$$\frac{e_1 \rhd_1 \mathcal{C}_1 \quad e_2 \rhd_1 \mathcal{C}_2}{e_1.m(e_2) \rhd_1 \{\mathcal{X}_e \supseteq app(\mathcal{X}_{e_1}, m, \mathcal{X}_{e_2})\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

since $\pi(e) = \pi(e_1) = \pi(e_2) = c'.m'$, this rule can be transformed into :

$$\frac{e_1 \rhd_3 \mathcal{C}_1 \quad e_2 \rhd_3 \mathcal{C}_2}{e_1.m(e_2) \rhd_3 \{\mathcal{X}_{c'.m'} \supseteq app_\pi(\mathcal{X}_{c'.m'}, m, \mathcal{X}_{c'.m'})\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

where app_π is a modified analysis function , and its semantics is defined by applying π as follows:

$$\mathcal{I}(app_\pi(\mathcal{X}_1, m, \mathcal{X}_2)) = \{v \mid c \in \mathcal{I}(\mathcal{X}_1), m(x) = e_m \in c, v \in \mathcal{I}(\mathcal{X}_{c.m}), \mathcal{I}(\mathcal{X}_{c.m}) \supseteq \mathcal{I}(\mathcal{X}_2)\}$$

The precision of this analysis can be improved by integrating type information into the analysis as in [19, 20]

5.2 Exception analysis

We first present a constraint system that analyzes uncaught exceptions from every expression independent of programmers' specification. The analysis can report programmer's unnecessary handlers or suggest to programmers for specialized handlings. We assume the analysis is done after a class analysis [7] or type inference [8, 16] and that class analysis information $Class(e)$ is already available for every expression e. Note that exception classes are normal classes in Java.

We first consider the rules to generate set constraints for the object classes of *every* expression in Figure 6. For exception analysis, every expression e of the program has a constraint: $\mathcal{P}_e \supseteq se$. The $\mathcal{P}_e$ is for the exception classes that the expression e's uncaught exception belongs to. Consider the rule for `throw` expression:

$$\frac{e_1 \rhd_1 \mathcal{C}_1}{\texttt{throw}\ e_1 \rhd_1 \{\mathcal{P}_e \supseteq Class(e_1) \cup \mathcal{P}_{e_1}\} \cup \mathcal{C}_1}$$

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:

$$\frac{e_0 \rhd_1 \mathcal{C}_0 \quad \rhd\ e_1 \rhd_2 \mathcal{C}_1}{\texttt{try}\ e_0\ \texttt{catch}\ (c_1\ x_1\ e_1) \rhd_1 \{\mathcal{P}_e \supseteq (\mathcal{P}_{e_0} - \{c_1\}^*) \cup \mathcal{P}_{e_1}\} \cup \mathcal{C}_0 \cup \mathcal{C}_1}$$

Raised 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 raised during the handling inside e_1. Hence, $\mathcal{P}_e \supseteq (\mathcal{P}_{e_0} - \{c_1\}^*) \cup \mathcal{P}_{e_1}$, where $\{c_1\}^*$ represents all the subclasses of a class c_1.

Consider the rule for method call:

$$\frac{e_1 \rhd_1 \mathcal{C}_1 \quad e_2 \rhd_1 \mathcal{C}_2}{e_1.m(e_2) \rhd_1 \{\mathcal{P}_e \supseteq \mathcal{P}_{c.m} | c \in Class(e_1), m(x) = e_m \in c\} \cup \{\mathcal{P}_e \supseteq \mathcal{P}_{e_1} \cup \mathcal{P}_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

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

Now we design a sparse constraint system by rule transformation that analyzes uncaught exceptions at method-level. In our new analysis, only two groups of set variables are considered: set variables for class' methods and `try`-blocks. For each method m, the set variable is for classes of uncaught exceptions during the call to m. The `try`-block e_g in the expression `try` e_g `catch` $(c_1\ x_1\ e_1)$ also has a set variable, which are for uncaught exception classes in e_g. The number of set variables is thus proportional only to the number of methods and `try` blocks, not to the total number of expressions. This design decision can be represented by an index function as follows:

Definition 5. An index function $I_X : Expr \cup MethodName \rightarrow Index$ is defined as follows :

$I_X(e) = c.g$	if e is a `try`-block e_g in a class c or appears in it
$I_X(e) = c.m$	if e appears in a method m of a class c.
$I_X(c.m) = c.m$	if m is a method defined in a class c

Let π be a partition function such that $I_X = \pi \circ I_E$. To design an sparse analysis, we transform the original rules by applying this partition function π to them. Figure 7 shows the transformed rules for each current expression e, assuming that $\pi(e) = c'.m'$.

For example, consider the rule for `try`-expression :

$$\frac{e_g \rhd_1 \mathcal{C}_g \quad e_1 \rhd_1 \mathcal{C}_1}{\texttt{try}\ e_g\ \texttt{catch}\ (c_1\ x_1\ e_1) \rhd_1 \{\mathcal{P}_e \supseteq \mathcal{P}_{e_g} - \{c_1\}^* \cup \mathcal{P}_{e_1}\} \cup \mathcal{C}_g \cup \mathcal{C}_1}$$

If the expression e is in a method $c.m$, then so does e_1. So, the rule is transformed and simplified into

$$\frac{e_g \rhd_2 \mathcal{C}_g \quad e_1 \rhd_2 \mathcal{C}_1}{\texttt{try}\ e_g\ \texttt{catch}\ (c_1\ x_1\ e_1) \rhd_2 \{\mathcal{P}_{c'.m'} \supseteq \mathcal{P}_{c.g} - \{c_1\}^*\} \cup \mathcal{C}_g \cup \mathcal{C}_1}$$

The two analyses give the same information on uncaught exceptions for every method and try-block. The sparse analysis in Figure 7 is shown to be *equivalent* to the expression-level analysis in Figure 6 with respect to methods and try-blocks :

[New$_1$] $\texttt{new}\ c \rhd_1 \emptyset$ [This$_1$] $\texttt{this} \rhd_1 \emptyset$

[FieldAss$_1$]
$$\frac{e_1 \rhd_1 \mathcal{C}_1}{id.x\ \texttt{:=}\ e_1 \rhd_1 \{\mathcal{P}_e \supseteq \mathcal{P}_{e_1}\} \cup \mathcal{C}_1}$$

[ParamAss$_1$]
$$\frac{e_1 \rhd_1 \mathcal{C}_1}{x\ \texttt{:=}\ e_1 \rhd_1 \{\mathcal{P}_e \supseteq \mathcal{P}_{e_1}\} \cup \mathcal{C}_1}$$

[Seq$_1$]
$$\frac{e_1 \rhd_1 \mathcal{C}_1 \quad e_2 \rhd_1 \mathcal{C}_2}{e_1\texttt{;}e_2 \rhd_1 \{\mathcal{P}_e \supseteq \mathcal{P}_{e_1} \cup \mathcal{P}_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[Cond$_1$]
$$\frac{e_0 \rhd_1 \mathcal{C}_0 \quad e_1 \rhd_1 \mathcal{C}_1 \quad e_2 \rhd_1 \mathcal{C}_2}{\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rhd_1 \{\mathcal{P}_e \supseteq \mathcal{P}_{e_0} \cup \mathcal{P}_{e_1} \cup \mathcal{P}_{e_2}\} \cup \mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[FieldVar$_1$]
$$\frac{id \rhd_1 \mathcal{C}_1}{id\texttt{.}x \rhd_1 \mathcal{C}_1}$$

[Throw$_1$]
$$\frac{e_1 \rhd_1 \mathcal{C}_1}{\texttt{throw}\ e_1 \rhd_1 \{\mathcal{P}_e \supseteq Class(e_1) \cup \mathcal{P}_{e_1}\} \cup \mathcal{C}_1}$$

[Try$_1$]
$$\frac{e_g \rhd_1 \mathcal{C}_g \quad e_1 \rhd_1 \mathcal{C}_1}{\texttt{try}\ e_g\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_1\texttt{)} \rhd_1 \{\mathcal{P}_e \supseteq \mathcal{P}_{e_g} - \{c_1\}^* \cup \mathcal{P}_{e_1}\} \cup \mathcal{C}_g \cup \mathcal{C}_1}$$

[MethCall$_1$]
$$\frac{e_1 \rhd_1 \mathcal{C}_1 \quad e_2 \rhd_1 \mathcal{C}_2}{\begin{array}{c} e_1.m(e_2) \rhd_1 \{\mathcal{P}_e \supseteq \mathcal{P}_{c.m} | c \in Class(e_1), m(x)\ \texttt{=}\ e_m \in c\} \cup \\ \{\mathcal{P}_e \supseteq \mathcal{P}_{e_1} \cup \mathcal{P}_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}}$$

[MethDef$_1$]
$$\frac{e_m \rhd_1 \mathcal{C}_m}{m(x) = e_m \rhd_1 \{\mathcal{P}_{c.m} \supseteq \mathcal{P}_{e_m}\} \cup \mathcal{C}_m}$$

[ClassDef$_1$]
$$\frac{M_i \rhd_1 \mathcal{C}_i \quad i = 1, \cdots, m}{\texttt{class}\ c\ \texttt{= \{var}\ x_1 \cdots x_n\ M_1 \cdots M_m\texttt{\}} \rhd_1 \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_m}$$

[Program$_1$]
$$\frac{C_i \rhd_1 \mathcal{C}_i \quad i = 1, \cdots, n}{C_1 \cdots C_n \rhd_1 \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

Figure 6: Exception analysis at expression-level

[New$_2$] $\texttt{new}\ c \rhd_2 \emptyset$ [This$_2$] $\texttt{this} \rhd_2 \emptyset$

[FieldAss$_2$]
$$\frac{e_1 \rhd_2 \mathcal{C}_1}{id.x\ \texttt{:=}\ e_1 \rhd_2 \mathcal{C}_1}$$

[ParamAss$_2$]
$$\frac{e_1 \rhd_2 \mathcal{C}_1}{x\ \texttt{:=}\ e_1 \rhd_2 \mathcal{C}_1}$$

[Seq$_2$]
$$\frac{e_1 \rhd_2 \mathcal{C}_1 \quad e_2 \rhd_2 \mathcal{C}_2}{e_1\texttt{;}e_2 \rhd_2 \mathcal{C}_1 \cup \mathcal{C}_2}$$

[Cond$_2$]
$$\frac{e_0 \rhd_2 \mathcal{C}_0 \quad e_1 \rhd_2 \mathcal{C}_1 \quad e_2 \rhd_2 \mathcal{C}_2}{\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rhd_2 \mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[FieldVar$_2$]
$$\frac{id \rhd_2 \mathcal{C}_1}{id\texttt{.}x \rhd_2 \mathcal{C}_1}$$

[Throw$_2$]
$$\frac{e_1 \rhd_2 \mathcal{C}_1}{\texttt{throw}\ e_1 \rhd_2 \{\mathcal{P}_{c'.m'} \supseteq Class(e_1)\} \cup \mathcal{C}_1}$$

[Try$_2$]
$$\frac{e_g \rhd_2 \mathcal{C}_g \quad e_1 \rhd_2 \mathcal{C}_1}{\texttt{try}\ e_g\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_1\texttt{)} \rhd_2 \{\mathcal{P}_{c'.m'} \supseteq \mathcal{P}_{c'.g} - \{c_1\}^*\} \cup \mathcal{C}_g \cup \mathcal{C}_1}$$

[MethCall$_2$]
$$\frac{e_1 \rhd_2 \mathcal{C}_1 \quad e_2 \rhd_2 \mathcal{C}_2}{e_1\texttt{.}m(e_2) \rhd_2 \{\mathcal{P}_{c'.m'} \supseteq \mathcal{P}_{c.m} \mid c \in Class(e_1), m(x) = e_m \in c\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[MethDef$_2$]
$$\frac{e_m \rhd_2 \mathcal{C}_1}{m(x)\ \texttt{=}\ e_m \rhd_2 \mathcal{C}_1}$$

[ClassDef$_2$]
$$\frac{M_i \rhd_2 \mathcal{C}_i \quad i = 1, \cdots, m}{\texttt{class}\ c\ \texttt{= \{var}\ x_1 \cdots x_n\ M_1 \cdots M_m\texttt{\}} \rhd_2 \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_m}$$

[Program$_2$]
$$\frac{C_i \rhd_2 \mathcal{C}_i \quad i = 1, \cdots, n}{C_1 \cdots C_n \rhd_2 \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

Figure 7: Exception analysis at method-level

Theorem 2. *Let p be a program and π be a partition function such that $I_X = \pi \circ I_E$. Let $\mathcal{C} = R(p)$ for the rules R in Figure 6 and $\mathcal{C}_\pi = R/\pi(p)$. Then, $lm(\mathcal{C}_\pi)(\mathcal{P}_{c.f}) = lm(\mathcal{C})(\mathcal{P}_{e_f})$ for every method $f(x) = e_f$ and try-block e_f in a class c.*
Proof. *See Appendix.* □

6. DISCUSSION

The class analysis has an $O(n^3)$ time bound where n is the number of expressions and variables in a program. Even if we consider the method-level analysis in Example 4, the order of time complexity does not change, but the number n of set variables is the same as the size of $I_M(Expr \cup Name)$, the number of methods and fields, which is usually much smaller than the number of expressions and variables. In case of the sparse exception analysis, the number of set-variables is proportional only to the number of methods and try-blocks, which is much smaller than the number of expressions. In general, if I is the index function for a new sparse analysis, then the number of set variables is the same as the size of $I(Expr \cup Name)$.

There have been several research directions to improve efficiency of constraint-based analysis. The first direction is to improve analysis time by simplifying set constraints after constructing the whole constraints [6, 9, 10, 18]. They usually simplify set constraints without losing the precision of the original analysis. Basic idea of congruence partitioning in [6] is to partition set variables based on idempotence and common subexpression relation. Componential set-based analysis [10] has added more relations for partitioning over congruence partitioning.

The second direction is to design analyses at a coarser granularity. Sparse exception analyses are designed for ML and Java [21, 20]. A function-level exception analysis for ML is manually designed in [21] where it is shown to be competitive in speed and precision by experimental study. An exception analysis with class analysis is manually designed for Java, and type information is also integrated in the analysis to refine precision in [20]. It is shown theoretically in [3] that the method-level exception analysis for Java gives the same information for each method as the expression-level analysis.

Several sparse versions of 0-CFA, called XTA, CTA,MTA and CTA, are designed individually for Java in [19]. They make class analysis scalable by making set variables for methods, fields, or classes. It is shown by experiments that they are fast for large practical programs and give relatively precise information. Static type information is also integrated in the analysis to refine precision as in [21, 20]. The idea of designing analyses at a coarser granularity is also applied in data flow analysis [14, 17], where syntactic tokens are used to group execution traces and coalesce the memory states associated with them, and abstract interpretation framework [2, 4], where a semantic function for every control point is approximated by partitioning control points.

7. CONCLUSION

We have assumed that the original analysis is designed at expression level and index determination functions are defined in terms of expressions. However, this idea need not be confined to expressions. We can assume an original analysis is designed at any level. For example, an original analysis can be defined for every expression and context as in k-CFA analysis. Then, 0-CFA can also be derived by transforming the rules of k-CFA.

Even though we present the framework based on a core Java language, it can also be applied to any programming languages and analyses, only if constraint-based analysis can be designed.

Another further research topic is on equivalence of analysis information. As in exception analysis, the sparse version can give the same information for some syntactic constructs like function as the original analysis. It is interesting and open to find general conditions for this equivalence. It is also interesting and promising to design other sparse versions of concurrency and security analysis by rule transformation.

ACKNOWLEDGEMENTS

We would like to thank K. Yi and R. Giacobazzi for helpful discussions. We are also grateful to the anonymous referees for many helpful suggestions that improved the presentation of the work.

8. REFERENCES

[1] D.F. Bacon and P.F. Sweeney, Fast static analysis of C++ virtual function calls. In *Proceedings of ACM Conference of Object-Oriented Programming Systems, languages, and Applications*, October 1996.

[2] F. Bourdoncle, Abstract interpretation by dynamic partitioning, *Journal of Functional Programming*, 2(4) (1992) 407-435.

[3] B.-M. Chang, J. Jo, K. Yi and K.-M. Choe, Interprocedural exception analysis for Java, In *Proceedings of ACM Symposium on Applied Computing*, LasVegas, USA, March 2001.

[4] P. Cousot and R. Cousot, Abstract interpreation and application to logic programs, *Journal of Logic Programming*, Vol 13, no. 2-3, pp. 103-179, 1992.

[5] P. Cousot and R. Cousot, Formal Language, Grammars and Set-Constraint-Based Program Analysis by Abstract Interpretation, In *Proceedings of '95 Conference on Functional Programming Languages and Computer Architecture*, pp. 25-28, June 1995.

[6] E. Duesterwald, R. Gupta and M. L. Soffa, Reducing the Cost of Data Flow Analysis by Congruence Partitioning, In *Proceedings of International Conference on Compiler Construction*, April 1994.

[7] G. DeFouw, D. Grove, and C. Chambers, Fast interprocedural class analysis, In *Proceedings of 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* pages 222-236, Januaray 1998.

[8] S. Drossopoulou, and S. Eisenbach, Java is type safe-probably, In *Proceedings of 1997 European Conference on Object-Oriented Programming*, 1997.

[9] M. Fahndrich, J. S. Foster, Z. Su and A. Aiken, Partial Online Cycle Elimination in Inclusion Constraint Graphs, In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.

[10] C. Flanagan and M. Felleisen, Componential Set-Based Analysis, In *Proceedings of the 24th ACM*

Symposium on Principles of Programming Languages, January 1997.

[11] C. Flanagan and M. Freund, Type-based Race Detection for Java In *Proceedings of the 27th ACM Symposium on Programming Languages Design and Implementation*, June 2000.

[12] N. Heintze, *Set-Based Program Analysis*, Ph.D Thesis, School of Computer Science, Carneige Mellon University, 1992.

[13] N. Heintze, Set-based analysis of ML programs, In *Proceedings of ACM Conference on Lisp and Functional Programming*, pp 306-317, 1994.

[14] N. D. Jones and S. Muchnick, A flexible approach interprocedural data flow analysis and programs with recursive data structures, In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, 1982.

[15] F. Nielson, H. Nielson and C. Hankin, *Principles of Program Analysis*, Springer-Verlag, December 1999.

[16] Tobias Nipkow and David von Oheimb. Java is type safe-definitely, In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*,1998.

[17] M. Sharir and A. Pnueli, Two approaches to interprocedural data flow analysis, in Muchnick and Jones Eds., *Program Flow Analysis, Theory and Applications*, Prentice-Hall, 1981.

[18] Z. Su, M. Fahndrich and A. Aiken, Projection Merging: Reducing Redundancies in Inclusion Constraint Graphs, In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, January 2000.

[19] F. Tip and J. Palsberg, Scalable propagation-based call graph construction algorithms, In *Proceedings of ACM Conference of Object-Oriented Programming Systems, languages, and Applications*, October 2000.

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

[21] K. Yi and S. Ryu, A Cost-effective estimation of uncaught exceptions in Standard ML programs. *Theoretical Computer Science*, volume 237, number 1, 2000.

Appendix. Proofs

Theorem 1 *Proof* As in [5], the continuous function $\mathcal{F}$ can be defined from $\mathcal{C}$, and $\mathcal{F}_\pi$ can also be defined from $\mathcal{C}_\pi$ likewise. So, we will prove this theorem by showing $\gamma \circ lfp(\mathcal{F}_\pi) \supseteq lfp(\mathcal{F})$.

We can prove this by showing that :
(1) Galois insertion: Let $\Delta = Vars(\mathcal{C})$ and $\Delta_\pi = Vars(\mathcal{C}_\pi)$. Let $\mathcal{D} = \Delta \rightarrow \wp(Val)$ be the domain of interpretations $\mathcal{I}$ and $\mathcal{D}_\pi = \Delta_\pi \rightarrow \wp(Val)$ be the domain of partitioned interpretations $\mathcal{I}_\pi$. For every interpretation $\mathcal{I}$, we define $\alpha(\mathcal{I}) = \mathcal{I}_\pi$ where $\mathcal{I}_\pi : \Delta_\pi \rightarrow \wp(Val)$ is defined as $\mathcal{I}_\pi(\mathcal{X}_m) = \cup_{e\in m}\mathcal{I}(X_e)$ for every $m \in \Delta_\pi$. We define $\gamma(\mathcal{I}_\pi) = \mathcal{I}'$ such that $\mathcal{I}'(\mathcal{X}_e) = \mathcal{I}_\pi(\mathcal{X}_{\pi(e)})$ for every set variable $\mathcal{X}_e \in \Delta$. Then, $(\mathcal{D}, \alpha, \mathcal{D}_\pi, \gamma)$ is a Galois insertion, since $\alpha(\gamma(\mathcal{I}_\pi)) = \mathcal{I}_\pi$.
(2) Soundness of the operation $\gamma \circ \mathcal{F}_\pi(\mathcal{I}_\pi) \supseteq \mathcal{F} \circ \gamma(\mathcal{I}_\pi)$: For this proof, it should be noted that the derivation rules in Figure 7 are obtained by replacing every set variable $\mathcal{X}_e$ by $\mathcal{X}_{\pi(e)}$ in the corresponding rules in Figure 6. So, if there is a constraint $\mathcal{X}_e \supseteq se$ constructed by the rules in Figure 6, then there must be a constraint $\mathcal{X}_{\pi(e)} \supseteq se/\pi$ constructed by the rules in Figure 7, where se/π denotes se with its set variable $\mathcal{X}_e$ replaced by $\mathcal{X}_{\pi(e)}$.

Let the function $\mathcal{F}$ be defined as a collection of equations of the form : $\mathcal{X}_e = se$ for every $\mathcal{X}_e \in \Delta$, and $\mathcal{F}_\pi$ as a collection of equations of the form : $\mathcal{X}_{\pi(e)} = se/\pi$ for every $\mathcal{X}_{\pi(e)} \in \pi(\Delta)$. For each set variable $\mathcal{X}_{e'}$ in se, $\gamma(\mathcal{I}_\pi)(\mathcal{X}_{e'}) = \mathcal{I}_\pi(\mathcal{X}_{\pi(e')}) = S$ by the definition of γ. Since $\mathcal{X}_{e'}$ is replaced by $\mathcal{X}_{\pi(e')}$ in $\mathcal{X}_{\pi(e)} = se/\pi$ in $\mathcal{F}_\pi$, and every set expression is monotone, $\mathcal{F}_\pi(\mathcal{I}_\pi)(\mathcal{X}_{\pi(e)}) \supseteq \mathcal{F} \circ \gamma(\mathcal{I}_\pi)(\mathcal{X}_e)$ for every set variable $\mathcal{X}_e$. Therefore, $\gamma \circ \mathcal{F}_\pi(\mathcal{I}_\pi) \supseteq \mathcal{F} \circ \gamma(\mathcal{I}_\pi)$ by the definition of γ.

Theorem 2 *Proof* As in the soundness proof, the continuous function $\mathcal{F}$ and $\mathcal{F}_\pi$ can be defined. We prove this theorem by showing that $lfp(\mathcal{F}_\pi)(\mathcal{P}_{c.f}) = lfp(\mathcal{F})(\mathcal{P}_{e_f})$ for every method and try-block. By the soundness theorem, $lfp(\mathcal{F}_\pi)(\mathcal{P}_{c.f}) \supseteq lfp(\mathcal{F})(\mathcal{P}_{e_f})$. So, we just prove that $lfp(\mathcal{F}_\pi)(\mathcal{P}_{c.f}) \subseteq lfp(\mathcal{F})(\mathcal{P}_{e_f})$ for every method and try-block.

The proof is by induction on the number of iterations in computing $lfp(\mathcal{F}_\pi)$.
Induction hypothesis : Suppose $\mathcal{I}_\pi(\mathcal{P}_{c.f}) \subseteq \mathcal{I}(\mathcal{P}_{e_f})$ for every method and try-block.
Induction step : If $\mathcal{I}'_\pi = \mathcal{F}_\pi(\mathcal{I}_\pi)$, then there exists $\mathcal{I}'$ such that $\mathcal{I}' = \mathcal{F}^i(\mathcal{I})$ for some i and $\mathcal{I}'_\pi(\mathcal{P}_{c.f}) \subseteq \mathcal{I}'(\mathcal{P}_{e_f})$.
(1) For every set variable $\mathcal{P}_{c.f}$, suppose $\mathcal{I}'_\pi(\mathcal{P}_{c.f}) = \mathcal{I}_\pi(\mathcal{P}_{c.f}) \cup \alpha$.
(2) Then, α must be added by some of the rules [Throw$_2$], [Try$_2$], and [MethodCall$_2$] in Figure 7.
(3) There must be the corresponding rules [Throw$_1$], [Try$_1$], and [MethodCall$_1$] in Figure 6.
(4) By (3) and induction hypothesis, there must be $\mathcal{X}_e$ such that e appears in e_f and $\mathcal{F}(\mathcal{I})(\mathcal{P}_e) \supseteq \alpha$, which will be eventually included in $\mathcal{P}_{e_f}$ in some more iterations $\mathcal{F}^i(\mathcal{I})$ by the rules in Figure 6.