Constraint-based Analysis for Java

Byeong-Mo Chang, Kwangkeun Yi and Jangwoo Jo

Abstract— **This paper presents constraint-based analysis for Java, and presents a systematic way to design a less costly analysis from an existing analysis. The basic idea is to approximate original set constraints by partitioning their set variables and replacing each set variable in them by the block, to which it belongs. We design a more efficient analysis by transforming the original constraint derivation rules using partition function. As an instance of the transformation, we provide a sparse uncaught exception analysis for Java, which approximates all the uncaught exceptions at method-level.**

Keywords— **set-based analysis, set constraints, derivation rules, partition function.**

I. Introduction

For constraint-based static analyses [3], [11], [12], [17], [16], there are three stages. First, a specific analysis is designed in the form of constraint derivation rules at *design-time*. Provided the derivation rules, set-based analysis constructs *set-constraints* for input programs using the rules, and then compute the least solution or model of the constructed constraints.

We first give the basic idea of constraint-based analysis in terms of class analysis for Java. Class analysis estimates for each expression the classes of objects, to which that expression may refer.

Constraint-based analysis usually sets up one set-variable for every expression. However, some analyses (like exception analysis [16] or race condition detection [10]) do not interested in properties or information for all expressions. In these analyses, generating constraints for every expression is wasteful, because the property-related expressions are sparse in programs. Hence, the analysis cost-effectiveness need to be addressed by enlarging the analysis granularity.

This paper presents a systematic way to design sparse constraint system that analyzes the properties at a larger granularity than at every expression. We propose a way to design a new less costly analysis by transforming the derivation rules for the existing analysis at expression-level into ones for the new sparse analysis. This is done by first defining (or designing) a *partition function* at design time, which specifies that only one set variable is to be made for such expressions that satisfy some syntactic properties.

Most of works [4], [8], [9], [15] proposed so far try to simplify set constraints after setting up the whole constraints.

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

Our approach tries to design sparse and efficient analysis by rule transformation at design time, so as to reduce the overhead for the whole constraints set-up and simplification.

We present exception analysis for Java as an instance for the rule transformation. Exception analysis estimates uncaught exceptions from expressions of the input program. It can be done after class analysis [1], [5] or type inference[6], [7], [14], using type information from them. Our exception analysis estimates their exception flows independently of the programmer's specifications.

Because exception-related expressions are sparse in programs, generating constraints for every expression is wasteful. Hence we present a sparse constraint system that analyzes uncaught exceptions at a method-level than at every expression. Moreover, the sparse exception analysis is shown to provide the same solution for each method as the expression-level exception analysis.

Section II presents the source language, and basic definitions for constraint-based analysis. Section III presents class analysis for Java. Section IV presents a systematic way to transform existing derivation rules into efficient ones using partition function. Section V presents an application of the framework. Section VI discusses related works and concludes.

II. Preliminaries

A. Source 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

This work is partly supported by KISTEP and by National Creative Research Initiatives of the Korean Ministry of Science and Technology. Byeong-Mo Chang: Dept. of Computer Science, Sookmyung Women's University, Seoul, Korea. E-mail: `chang@cs.sookmyung.ac.kr`. Kwangkeun Yi: Dept. of Computer Science, Korea Advanced Institute of Science & Technology, Taejon, Korea. E-mail: `kwang@cs.kaist.ac.kr`. Jangwoo Jo is with the Dept. of Computer Engineering, Pusan University of Foreign Studies, Pusan, Korea. E-mail: `jjw@saejong.pufs.ac.kr`.

Syntax of set expressions:

$$\begin{array}{rcll} se & ::= & \mathcal{X}_i & \text{set variables} \\ & | & c & \text{class names} \\ & | & app(\mathcal{X}_i, m, \mathcal{X}_i) & \text{sets from method call} \\ & | & se \cup se & \text{sets from conditionals} \\ & | & se - \{c_1, \cdots, c_n\} & \text{sets from try-catchs} \\ & | & \top & \text{universe set} \end{array}$$

Semantics of set expressions:

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

Fig. 2. Set constraints : syntax and semantics

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

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

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 :

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

The exception object e_0 is raised by `throw` e_0. The programmers have to declare in a method definition any exception 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.

We omit the formal semantics of the core language. In the natural semantics formalism, operational semantics should be straightforward, not much different from existing works [6], [7], [14].

B. Set constraints for analyses of Java

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 $\mathcal{X}$ must have the set se. The set expression is of the form $se \rightarrow c|\mathcal{X}|app(\mathcal{X}_1, m, \mathcal{X}_2)|se \cup se|se - \{c_1, ..., c_n\}$ where c's are class names. Multiple constraints are conjunctions. We write $\mathcal{C}$ for a finite collection of set constraints and Δ for the set of set variables in $\mathcal{C}$.

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 $\mathcal{X}_2$. Note that $\mathcal{X}_{this}$ is the set-variable for `this` in the method m. The formal semantics of set expressions is defined by an interpretation $\mathcal{I}$ that maps from set expressions to sets of values (see Figure 4). 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 [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 constraints by the derivation rules, that describe the data flows between the expressions of the analyzed program. The second phase finds the sets of values by the solving rule that satisfy the constraints. A solution is a table from set variables in the constraints to the finite descriptions of such sets of values.

III. Constraint-based analysis for Java

We first present a class analysis for Java based on set-based analysis framework [11]. Every expression e of the program has one set constraints: $\mathcal{X}_e \supseteq se$. The $\mathcal{X}_e$ is for the classes that the expression e's normal object belongs to. Figure 3 has the derivation rules to generate set constraints for the classes of the objects of *every* expression. The subscript e of set variables $\mathcal{X}_e$ 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 `new`. The `new` expression will have the newly created object from the class c, hence $\mathcal{X}_e \supseteq \{c\}$. Consider the rule for `if`. 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:

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

[New]	$\triangleright \texttt{new}\ c : \{\mathcal{X}_e \supseteq \{c\}\}$
[FieldAss]	$\dfrac{\triangleright e_1 : \mathcal{C}_1}{\triangleright id.x := e_1 : \{\mathcal{X}_{c.x} \supseteq \mathcal{X}_{e_1} \mid c \in \mathcal{X}_{id}\} \cup \{\mathcal{X}_e \supseteq \mathcal{X}_{e_1}\} \cup \mathcal{C}_1}$
[ParamAss]	$\dfrac{\triangleright e_1 : \mathcal{C}_1}{\triangleright x \texttt{ := } e_1 : \{\mathcal{X}_x \supseteq \mathcal{X}_{e_1}, \mathcal{X}_e \supseteq \mathcal{X}_{e_1}\} \cup \mathcal{C}_1}$
[Seq]	$\dfrac{\triangleright e_1 : \mathcal{C}_1 \quad \triangleright e_2 : \mathcal{C}_2}{\triangleright e_1 ; e_2 : \{\mathcal{X}_e \supseteq \mathcal{X}_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$
[Cond]	$\dfrac{\triangleright e_0 : \mathcal{C}_0 \quad \triangleright e_1 : \mathcal{C}_1 \quad \triangleright e_2 : \mathcal{C}_2}{\triangleright \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \{\mathcal{X}_e \supseteq \mathcal{X}_{e_1} \cup \mathcal{X}_{e_2}\} \cup \mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C}_2}$
[FieldVar]	$\dfrac{\triangleright id : \mathcal{C}_{id}}{\triangleright id.x : \{\mathcal{X}_e \supseteq \mathcal{X}_{c.x} \mid c \in \mathcal{X}_{id}\} \cup \mathcal{C}_{id}}$
[Throw]	$\dfrac{\triangleright e_1 : \mathcal{C}_1}{\triangleright \texttt{throw}\ e_1 : \mathcal{C}_1}$
[Try]	$\dfrac{\triangleright e_0 : \mathcal{C}_0 \quad \triangleright e_1 : \mathcal{C}_1}{\triangleright \ \texttt{try}\ e_0\ \texttt{catch}(c_1\ x_1\ e_1) : \ \{\mathcal{X}_e \supseteq \mathcal{X}_{e_0} \cup \mathcal{X}_{e_1}, \mathcal{X}_{x_1} \supseteq \{c\}^*\} \cup \mathcal{C}_0 \cup \mathcal{C}_1}$
[MethCall]	$\dfrac{\triangleright e_1 : \mathcal{C}_1 \quad \triangleright e_2 : \mathcal{C}_2}{\triangleright \ e_1.m(e_2) : \ \{\mathcal{X}_e \supseteq app(\mathcal{X}_{e_1}, m, \mathcal{X}_{e_2})\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$
[MethDef]	$\dfrac{\triangleright e_m : \mathcal{C}}{\triangleright m(x) = e_m : \{\mathcal{X}_{c.m} \supseteq \mathcal{X}_{e_m}\} \cup \mathcal{C}}$
[ClassDef]	$\dfrac{\triangleright m_i : \mathcal{C}_i \ i = 1, \cdots, n}{\triangleright \texttt{class}\ c\ \texttt{= \{var}\ x_1, \cdots, x_k, m_1, \cdots, m_n\texttt{\}} : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$
[Program]	$\dfrac{\triangleright C_i : \mathcal{C}_i \ i = 1, \cdots, n}{\triangleright C_1, \cdots, C_n : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$

Fig. 3. Class analysis

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

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

Fig. 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 (The subscript $c.m$ indicates the index for the body expression of class c's method m.), and the constraint $\mathcal{X}_x \supseteq \mathcal{X}_{e_2}$ is also added for parameter binding: objects of e_2 are passed to the method parameter x.

Consider the rule for `try` expression: [Try]

$$\frac{\triangleright e_0 : \mathcal{C}_0 \quad \triangleright e_1 : \mathcal{C}_1}{\triangleright \texttt{try}\ e_0\ \texttt{catch}(c_1\ x_1\ e_1) : \{\mathcal{X}_e \supseteq \mathcal{X}_{e_0} \cup \mathcal{X}_{e_1}, \mathcal{X}_{x_1} \supseteq \{c\}^*\} \cup \mathcal{C}_0 \cup \mathcal{C}_1}$$

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't be approximated finely without exception analysis. So we just approximate $\mathcal{X}_{x_1}$ by all the subclasses (denoted by c^*) of its class c.

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

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

It introduces $\mathcal{X} \supseteq \mathcal{X}_{c.m}$ if a method m to call is in a class c and if so, adds $\mathcal{X}_x \supseteq \mathcal{X}_2$ and $\mathcal{X}_{this} \supseteq \mathcal{X}_1$ to simulate the parameter binding (`this` can be viewed as a paramter receiving the target object). 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 [3] from our constraint system.

IV. Design of analyses by rule-transformation

Constraint-based analysis usually sets up one set-variable for every expression. However, some analyses (like exception analysis [16] or race condition detection [10]) do not interested in properties or information for all expressions. In these analyses, generating constraints for every expression is wasteful, because the property-related expressions are sparse in programs. Hence, the analysis cost-effectiveness need to be addressed by enlarging the analysis granularity.

We present a systematic way to design sparse but more efficient analyses from a given original analysis. For this, we transform existing derivation rules into new ones using partitioning. This will be done by defining (or designing) a *partition function* at design time, which specifies that one set variable is to be set up for all sub-expressions that satisfy some syntactic properties.

As noted in [3], the precision of the analysis depends upon the choice of the finite set of indices of set-variable. We formalize this idea by defining a partition function at design-time. Let $Expr$ be the set of expressions. A partitioning is defined as a function $\pi : Expr \rightarrow \wp(Expr)$, which can be defined in terms of syntactic properties of programs. If $\pi(e_1) = \pi(e_2)$, then it specifies that one set variable is constructed for two expressions e_1 and e_2. It should be noted that partition functions must be universally defined (or designed) independent of a particular program, since it must be able to be applied to any input programs. For example, instead of constructing one set variable for each expression and for each variable, we can define a method-level analysis which constructs one set variable for each method.

We can transform original derivation rules into new ones with a partition function.

Definition 1 (Rule transformation) Let $\pi : Expr \rightarrow \wp(Expr)$ be a partition function. Let $e = \kappa(e_1, \cdots, e_n)$ be a generic expression where κ is a language construct. If r is a derivation rule of the form :

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

then, r/π is defined as:

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

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

An analysis in set-based framework is designed by a set R of derivation rules. We can design a sparse analysis by R/π, which is defined as $\{r/\pi | r \in R\}$. However, partitions designed at design time are apt to be more coarse-grained than the simplification after constraints set-up, because they must be defined universally and independently of a particular program.

We can construct new constraints with the new rules. Whenever an expression is encountered during constraints set-up, its set variable is determined by applying the partition function to it, and constraints are constructed. Then, the number of set-variables to be constructed for a program is usually smaller than the original ones.

To solve the constraints constructed by the rule in Definition 1, we can use the same solving rules in Figure 4 with a little modification in function application as:

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

We denote by $R(p)$ (or $(R/\pi)(p)$) the set of constraints constructed by applying the derivation rules in R (or R/π) to a program p.

We denote by $lm(R(p))/\pi$ the partitioning of the least model $lm(R(p))$ computed for the set constraints $R(p)$. Then, we can show the soundness of constructed constraints by the partitioned derivation rule as follows.

Theorem 1 (Soundness) Let p be a program 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 set variable $\mathcal{X}_e$.

Proof. See Appendix. □

V. Application

As an instance of the partitioning, we apply the partitioning to uncaught exception analysis for Java. We first present an exception analysis at expression-level and then design a sparse exception analysis by applying the partition function to the expression-level exception analysis.

A. Exception analysis

Because unhandled exceptions will abort the program's execution, the current JDK Java compiler does an exception analysis relying on the programmer's specifications to check that the input program will have no uncaught exceptions at run-time. However, programmers can declare that a method throws exceptions of the most general class `Exception` even if the actual exceptions are of much more specific classes. Then its handler cannot offer proper treatments specific to the actual exceptions. As well as that, programmers can also write too broad or unnecessary `catch` clauses. These two cases are shown in the source code in Figure 5.

```
class Demo1 {
  public static void main( ) throws Exception {
          try {
                demoproc();
          } catch (Exception e) {  ; }
  }

   void demoproc() throws Exception {
          try {
                 throw new IOException("demo");
          } catch (Exception e) { ; }
    }
}
```

Fig. 5. The source code for exceptions

Careless (or inconfident) programmers may also 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 be useless.

Since the current JDK compiler is not elaborate enough to do "better" than as specified by the programmers, we present a constraint system that analyzes uncaught exceptions from every expression of the program independent of programmers' specification. We will devise an interprocedural exception analysis so that it can report programmer's unnecessary handlers or suggest to programmers for

[New]		$\triangleright\ \texttt{new}\ c\ : \emptyset$
[FieldAss]		$\dfrac{\triangleright\ e_1\ : \mathcal{C}_1}{\triangleright\ id.x\ \texttt{:=}\ e_1\ : \{\mathcal{P}_e \supseteq \mathcal{P}_{e_1}\} \cup \mathcal{C}_1}$
[ParamAss]		$\dfrac{\triangleright\ e_1\ : \mathcal{C}_1}{\triangleright\ x\ \texttt{:=}\ e_1\ : \{\mathcal{P}_e \supseteq \mathcal{P}_{e_1}\} \cup \mathcal{C}_1}$
[Seq]		$\dfrac{\triangleright\ e_1\ : \mathcal{C}_1 \quad \triangleright\ e_2\ : \mathcal{C}_2}{\triangleright\ e_1 ; e_2\ : \{\mathcal{P}_e \supseteq \mathcal{P}_{e_1} \cup \mathcal{P}_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$
[Cond]		$\dfrac{\triangleright\ e_0\ : \mathcal{C}_0 \quad \triangleright\ e_1\ : \mathcal{C}_1 \quad \triangleright\ e_2\ : \mathcal{C}_2}{\triangleright\ \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ : \{\mathcal{P}_e \supseteq \mathcal{P}_{e_0} \cup \mathcal{P}_{e_1} \cup \mathcal{P}_{e_2}\} \mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C}_2}$
[FieldVar]		$\dfrac{\triangleright\ id\ : \mathcal{C}_1}{\triangleright\ id.x\ : \mathcal{C}_1}$
[Throw]		$\dfrac{\triangleright\ e_1\ : \mathcal{C}_1}{\triangleright\ \texttt{throw}\ e_1\ : \{\mathcal{P}_e \supseteq Class(e_1) \cup \mathcal{P}_{e_1}\} \cup \mathcal{C}_1}$
[Try]		$\dfrac{\triangleright\ e_g\ : \mathcal{C}_g \quad \triangleright\ e_1\ : \mathcal{C}_1}{\triangleright\ \texttt{try}\ e_g\ \texttt{catch}\ (c_1\, x_1\, e_1)\ : \{\mathcal{P}_e \supseteq \mathcal{P}_{e_g} - \{c_1\}^* \cup \mathcal{P}_{e_1}\} \cup \mathcal{C}_0 \cup \mathcal{C}_1}$
[MethCall]		$\dfrac{\triangleright\ e_1\ : \mathcal{C}_1 \quad \triangleright\ e_2\ : \mathcal{C}_2}{\triangleright\ e_1 .m(e_2)\ : \{\mathcal{P}_e \supseteq \mathcal{P}_{c.m} \mid c \in Class(e_1), m(x) = e_m \in c\} \cup \{\mathcal{P}_e \supseteq \mathcal{P}_{e_1} \cup \mathcal{P}_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$
[MethDef]		$\dfrac{\triangleright\ e_m\ : \mathcal{C}_m}{\triangleright\ m(x) = e_m\ : \{\mathcal{P}_{c.m} \supseteq \mathcal{P}_{e_m}\} \cup \mathcal{C}_m}$
[ClassDef]		$\dfrac{\triangleright\ M_i\ : \mathcal{C}_i \quad i = 1, \cdots, m}{\triangleright\ \texttt{class}\ c\ \texttt{= \{var}\ x_1 \cdots x_n\ M_1 \cdots M_m \texttt{\}}\ : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_m}$
[Program]		$\dfrac{\triangleright\ C_i\ : \mathcal{C}_i \quad i = 1, \cdots, n}{\triangleright\ C_1 \cdots C_n\ : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$

Fig. 6. Exception analysis at expression-level

specialized handlings. Our exception analysis can be done after a class analysis [5] or type inference [6], [14], which estimates, for each expression e, the classes (including exception classes) that the expression e's normal object belongs to. Note that exception classes are normal classes in Java. We assume that class analysis information $Class(e)$ is already available for every expression e in the exception analysis.

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:

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

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:

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

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 $\{c\}^*$ represents all the subclasses of a class c.

Consider the rule for method call:

$$\frac{\triangleright\ e_1 : \mathcal{C}_1 \quad \triangleright\ e_2 : \mathcal{C}_2}{\triangleright\ \ e_1 .m(e_2) : \ \begin{array}{l}\{\mathcal{P}_e \supseteq \mathcal{P}_{c.m} \mid c \in Class(e_1), m(x) = e_m \in c\} \\ \cup \{\mathcal{P}_e \supseteq \mathcal{P}_{e_1} \cup \mathcal{P}_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

Uncaught exceptions from the call expression first include those from the subexpressions e_1 and e_2 : $\mathcal{X}_e \supseteq \mathcal{X}_{e_1} \cup \mathcal{X}_{e_2}$. This 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. (The subscript $c.m$ indicates the index for the body expression of class c's method m.)

B. Sparse exception analysis

We have presented 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. Hence we present a sparse constraint system that analyzes uncaught exceptions at a larger granularity than at every expression.

We design derivation rules for a sparse exception analysis, which approximates uncaught exception for every method and try-block. In our new sparse constraint system, only two groups of set variables are considered: set variables for class' methods and `try`-blocks. The number of unknowns is thus proportional only to the number of methods and `try` blocks, not to the total number of expressions. For each method m, the set variable is for classes of uncaught exceptions during the call to m. The `try`-block e_g in a `try-catch` 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.

New derivation rules for the sparse exception analysis can be derived *automatically* as in Definition 1 by just applying the following partition function to the original derivation rules in Figure 6. We denote by $e' \in e$ if $e' = e$

[New$_{ME}$] $$f \triangleright \texttt{new}\ c : \emptyset$$

[FieldAss$_{ME}$] $$\frac{f \triangleright e_1 : \mathcal{C}_1}{f \triangleright id.x\ \texttt{:=}\ e_1 : \mathcal{C}_1}$$

[ParamAss$_{ME}$] $$\frac{f \triangleright e_1 : \mathcal{C}_1}{f \triangleright x\ \texttt{:=}\ e_1 : \mathcal{C}_1}$$

[Seq$_{ME}$] $$\frac{f \triangleright e_1 : \mathcal{C}_1 \quad f \triangleright e_2 : \mathcal{C}_2}{f \triangleright e_1 ; e_2 : \mathcal{C}_1 \cup \mathcal{C}_2}$$

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

[FieldVar$_{ME}$] $$\frac{f \triangleright id : \mathcal{C}_1}{f \triangleright id.x : \mathcal{C}_1}$$

[Throw$_{ME}$] $$\frac{f \triangleright e_1 : \mathcal{C}_1}{f \triangleright \texttt{throw}\ e_1 : \{\mathcal{P}_{f*} \supseteq Class(e_1)\} \cup \mathcal{C}_1}$$

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

[MethCall$_{ME}$] $$\frac{f \triangleright e_1 : \mathcal{C}_1 \quad f \triangleright e_2 : \mathcal{C}_2}{f \triangleright e_1.m(e_2) : \{\mathcal{P}_{f*} \supseteq \mathcal{P}_{c.m} | c \in Class(e_1), m(x) = e_m \in c\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[MethDef$_{ME}$] $$\frac{m \triangleright e_m : \mathcal{C}_1}{\triangleright m(x) = e_m : \{\mathcal{P}_{c.m} \supseteq \mathcal{P}_{m*}\} \cup \mathcal{C}_1}$$

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

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

Fig. 7. Sparse exception analysis

or e' is a subexpression of e. We can define the partition function $\pi_1 : Expr \to \wp(Expr)$ as follows :

$\pi_1(e) = \{e' | e' \in e_g\}$ if
$e \in e_g$ for the innermost expression $\texttt{try}\ e_g\ \texttt{catch}(c_1\ x_1\ e_1)$
$\pi_1(e) = \{e' | e' \in e_m,\ e'$ is not in a **try**-block$\}$ if
$e \in e_m$ for a method m=e_m, and e is not in a **try**-block.

This partition function specifies that there is one set variable denoted by $\mathcal{P}_{g*}$ for all sub-expressions of the **try**-block e_g, and one set-variable denoted by $\mathcal{P}_{m*}$ for all sub-expressions of a method m.

We make the new derivation rules in Figure 7 for the sparse exception analysis by transforming the derivation rules in Figure 6 using the partition function. In Figure 7, the left-hand-side f in relation $f \triangleright e : \mathcal{C}$ indicates that the expression e is a sub-expression of method f (or **try**-block f).

For example, consider the expression-level rule for **try**-expression :

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

If the expression e is in a method f, then e_1 is also in f. So, we transform $\mathcal{P}_e$ and $\mathcal{P}_{e_1}$ into $\mathcal{P}_{f*}$. We also transform $\mathcal{P}_{e_g}$ into $\mathcal{P}_{g*}$, since $\pi_1(e_g) = g*$, that is, one set-variable is made for **try**-block e_g. Note that $\mathcal{P}_{f*} \supseteq \mathcal{P}_{g*} - \{c_1\}^*$ is the same as $\mathcal{P}_{f*} \supseteq \mathcal{P}_{g*} - \{c_1\}^* \cup \mathcal{P}_{f*}$. Then, we can obtain the following rule for the method-level analysis :

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

Likewise, consider the rule for conditionals. The constraints $\mathcal{P}_e \supseteq \mathcal{P}_{e_0} \cup \mathcal{P}_{e_1} \cup \mathcal{P}_{e_2}$ is transformed into a tautology $\mathcal{P}_f \supseteq \mathcal{P}_f \cup \mathcal{P}_f \cup \mathcal{P}_f$, because if the expression e is in f, then e_0, e_1 and e_2 are also in f. Consider the rules for method calls. The constraint $\mathcal{P}_e \supseteq \mathcal{P}_{c.m}$ is transformed into $\mathcal{P}_{f*} \supseteq \mathcal{P}_{c.m}$, if the expression e is in a method f.

It should be noted that the derivation rules for try-blocks, for example e_g, are the same as those in Figure 7, except that f is replaced by g.

We show that the sparse analysis in Figure 7 gives, for every method and **try**-block, the same information on uncaught exceptions as the expression-level analysis in Figure 6. Let p be a program and π_1 be the partition function. Let $\mathcal{C} = R(p)$ and $\mathcal{C}_{\pi_1} = R/\pi_1(p)$. Then, we call $\mathcal{C}_{\pi_1}$ is *complete* with respect to $\mathcal{C}$ for every method and try-block : if $lm(\mathcal{C}_{\pi_1})(\mathcal{X}_{f*}) = lm(\mathcal{C})(\mathcal{X}_f)$ for every method and try-block f.

Theorem 2 (Completeness) Let p be a program and π_1 be the partition function. Let $\mathcal{C} = R(p)$ and $\mathcal{C}_{\pi_1} = R/\pi(p)$. $lm(\mathcal{C}_{\pi_1})(\mathcal{X}_{f*}) = lm(\mathcal{C})(\mathcal{X}_f)$ for every method and **try**-block f.

Proof. See Appendix. □

VI. Discussions

Most of works [4], [8], [9], [15] on simplifying set constraints proposed so far try to partition set variables after setting up the whole constraints, without losing the precision of the original analysis. Congruence partitioning in data-flow equation [4] is to partition set variables based on idempotence and common subexpression relation. Componential set-based analysis [9] has four rules for simplification, two of them are trivial and the others are basically based on the same idea as congruence partitioning. Sparse analyses have been explored in uncaught exception analysis for ML and Java [17], [16], [18]. However, they were designed manually, not by a systematic rule transformation.

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 set-based analysis can be designed. In particular, the framework can be easily applied to functional languages like ML.

We have implemented the sparse exception analysis for Java, which is designed by the rule transformation. Interesting experimental results are presented in [2]. We are also working on designing other sparse analysis by the rule transformation. Especially, we expect to design a complete sparse analysis for race condition detection by the proposed rule transformation.

References

[1] C. Chambers, Efficient Interprocedural Class Analysis, *Proceedings of ACM Principles of Programming Languages*, January 1998.

[2] B. Chang, J. Jo and K. Yi, Interprocedural Exception Analysis for Java, submitted to *Int. Conf. on Foundation of Software Technology and Theoretical Computer Science.*

[3] 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.

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

[5] 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.

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

[7] S. Drossopoulou, and T. Valkevych, Java exceptions thro no surprises, "http://www-dse.doc.ic.ac.uk/ scd", March 2000.

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

[9] C. Flanagan and M. Felleisen, Componential Set-Based Analysis, In *Proceedings of ACM Symposium on Principles of Programming Languages*, January 1997.

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

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

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

[13] F. Nielson, H. Nielson and S. Abramsky, *Principles of Program Analysis*, Springer-Verlag, December 1999.

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

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

[16] Kwangkeun 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.

[17] Kwangkeun Yi and Sukyoug 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.

[18] Kwangkeun Yi and Sukyoung Ryu, A Cost-effective Estimation of Uncaught Exceptions in Standard ML Programs. *Theoretical Computer Science*, volume 237, number 1, 2000. (to appear)

Appendix. Proofs

Theorem 1 *Proof.* As in [3], 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 method $m \in \Delta_\pi$. 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 $\mathcal{X}_{\pi(e)} \in \pi(\Delta)$. Assume that, for each set variable $\mathcal{X}_{e'}$ in se, $\gamma(\mathcal{I}_\pi)(\mathcal{X}_{e'}) = S$. Then $\mathcal{I}_\pi(\mathcal{X}_{\pi(e')}) = S$ by the definition of γ. $\mathcal{X}_{e'}$ is replaced by $\mathcal{X}_{\pi(e')}$ in $\mathcal{X}_{\pi(e)} = se/\pi$ in $\mathcal{F}_\pi$, and every set expression is monotone. Therefore, $\mathcal{F}_\pi(\mathcal{I}_\pi)(\mathcal{X}_{\pi(e)}) \supseteq \mathcal{F} \circ \gamma(\mathcal{I}_\pi)(\mathcal{X}_e)$ for every set variable $\mathcal{X}_e$, and $\gamma \circ \mathcal{F}_\pi(\mathcal{I}_\pi) \supseteq \mathcal{F} \circ \gamma(\mathcal{I}_\pi)$ by the definition of γ. □

Theorem 2 *Proof.* As in the soundness proof, the continuous function $\mathcal{F}$ and $\mathcal{F}_{\pi_1}$ can be defined. We prove this theorem by showing that $lfp(\mathcal{F}_{\pi_1})(\mathcal{X}_f) = lfp(\mathcal{F})(\mathcal{X}_f)$ for every method and try-block f. By the soundness theorem, $lfp(\mathcal{F}_{\pi_1})(\mathcal{X}_f) \supseteq lfp(\mathcal{F})(\mathcal{X}_f)$. So, we just prove that $lfp(\mathcal{F}_{\pi_1})(\mathcal{X}_f) \subseteq lfp(\mathcal{F})(\mathcal{X}_f)$ for every method and try-block f.

The proof is by induction on the number of iterations in computing $lfp(\mathcal{F}_{\pi_1})$.

Induction hypothesis : Suppose $\mathcal{I}_{\pi_1}(\mathcal{X}_f) \subseteq \mathcal{I}(\mathcal{X}_f)$ for every method and try-block. *Induction step* : f. Let $\mathcal{I}'_{\pi_1} = \mathcal{F}_{\pi_1}(\mathcal{I}_{\pi_1})$. Then there exists $\mathcal{I}'$ such that $\mathcal{I}' = \mathcal{F}^i(\mathcal{I})$ for some i and $\mathcal{I}'_{\pi_1}(\mathcal{X}_f) \subseteq \mathcal{I}'(\mathcal{X}_f)$ for every method and try-block f.

(1) For every set variable $\mathcal{X}_f$, suppose $\mathcal{I}'_{\pi_1}(\mathcal{X}_f) = \mathcal{I}_{\pi_1}(\mathcal{X}_f) \cup \alpha$.

(2) Then, α must be added by some of the rules [Throw$_M E$], [Try$_M E$], and [MethodCall$_M E$] in Figure 7.

(3) There must be the corresponding rules [Throw], [Try], and [MethodCall] in Figure 6.

(4) By (3) and induction hypothesis, there must be $\mathcal{X}_e$ such that $\mathcal{F}(\mathcal{I})(\mathcal{X}_e) \supseteq \alpha$, which will be eventually included in $\mathcal{X}_f$ in some more iterations $\mathcal{F}^i(\mathcal{I})$ by the rules in Figure 6, because e is in f. □