Abstract  A transformation–based approach was proposed to design constraint–based analyses at coarser granularity. This approach lets us be able to design a less precise but more efficient version of an original expression–level analysis by transforming the original construction rules. Since this approach assumes that the original analysis is designed at expression–level, it has a limit that it is applicable only to expression–level analyses. However, this idea need not be confined to expression–level, and need to be extended. In this paper, we extend this rule–transformation approach so that it can be applied to any analyses which are designed at any level, and so provide a (em multi–level) mechanism to design practical constraint–based analyses by rule transformation. Using this approach, we also investigate the relationship between a number of CFAs for Java systematically, which determine the interprocedural control flow information.

Keywords: constraint–based analysis, construction rules, multi–level rule transformation, partition
granularity. Several constraint-based analyses including CFA and exception analysis are designed manually at coarser granularity, experimented and practically applied in [5,6]. A number of sparse versions of 0-CFA, called XTA, FTA, MTA, CTA and RTA were designed manually for Java in [6], which make control flow analysis scalable by making set variables for methods, fields, or classes.

The precision of constraint-based analysis depends on the choice of the finite set of indices of set variables. We usually design an analysis theoretically at expression-level, which makes one set variable for each expression. However, the efficiency of an expression-level analysis may not be satisfactory for large practical programs[1]. In addition, some analyses like exception analysis[5,7] are not interested in properties of all expressions. So, the analysis cost-effectiveness need to be investigated by enlarging the analysis granularity.

A transformation-based approach was proposed to design constraint-based analyses at coarser granularity in [8]. This approach lets us be able to design a less precise but more efficient version from an original expression-level analysis by transforming the original construction rules. Since this approach assumes that the original analysis is designed at expression-level, it has a limit that it is applicable only to expression-level analyses, and it cannot relate two analyses designed at other syntactic levels such as methods and classes. For example, it cannot relate XTA and CTA directly by rule transformation, which are CFAs designed at method-level and class-level respectively. However, this idea need not be confined to expressions and need to be extended to more general level of analyses.

In this paper, we extend this rule-transformation approach into a multi-level approach so that it can relate any two analyses by rule transformation, only if there is a partition relation between the two sets of set-variables for the two analyses. This approach can be applied to any analyses designed at any syntactic level, and so provide a multi-level mechanism to design practical analyses by rule transformation. We show the multi-level approach is useful by investigating the multi-level relationship between a number of CFAs for Java as in Figure 1.

Section 2 presents a core of Java, and basic definitions for constraint-based analysis. Section 3 presents a systematic mechanism to design analyses by rule transformation. Section 4 presents the relationship between various CFAs using the rule transformation. Section 5 discusses related works and Section 6 concludes this paper.

2. Preliminaries

We consider an imaginary core of Java with exception constructs. Its abstract syntax is in Figure 1. A program is a sequence of class definitions. A class \( c \) in a program \( P \) is denoted by \( c \in P \). Class bodies consist of field variable declarations and method definitions, where we omit type information for simplicity. A method \( m \) defined in a class \( c \) is denoted by \( m \in c \), and a field variable \( f \) declared in a class \( c \) is denoted by \( f \in c \). A method definition consists of the method name, its parameter, and its body expression. A parameter or local variable \( x \) declared in a method \( m \) is denoted by \( x \in m \).

Every expression’s result is an object. Assignment expression returns the object of its righthand side expression. Sequence expression returns the object of the last expression in the sequence. Conditional expression returns either the object of then or else part. A method call returns the object from the method body. The try-catch expression is an expression for handling exceptions. Its operational semantics is straightforward[9].

\[
\begin{align*}
P &::= C^+ & \text{program} \\
C &::= \text{class } c \text{ ext } c \{ f^+ M^+ \} & \text{class definition} \\
M &::= \text{new } c & \text{method definition} \\
e &::= \text{this } x | x.f | x = e | e.f = e | e; e | \text{if } e \text{ then } e \text{ else } e | \text{throw } e | \text{try } e \text{ catch } \{ c x e \} | e.m(e) & \text{expression} \\
c &\in P & \text{class name} \\
m &\in c & \text{method name} \\
f &\in c & \text{field name} \\
x &\in m & \text{local name}
\end{align*}
\]

Fig. 1 Abstract syntax of a core of Java
Constraint-based analysis consists of two phases [1]: collecting set constraints and solving them. The first constructs set-constraints by the construction rules, that describe dataflows between the expressions of the program. The second 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 \( X \supseteq s \) where \( X \) is a set variable and \( s \) is a set expression. The constraint indicates that the set variable \( X \) must contain the set \( s \). In case of CFA for Java, the set expression is of the form

\[
X \supseteq s \cup s
\]

where \( c \) is a class name. Multiple constraints are conjunctions. We write \( C \) for a finite collection of set constraints. Semantics of set expressions naturally follows from their corresponding language constructs. For example, \( \text{app}(X_1, m_1) \) represents the classes of objects returned from applications of method \( m_1 \) to target objects in \( X_1 \) with parameters in \( X_2 \). A constraint \( X \supseteq \text{app}(X_1, m_1) \) can also be represented as a conditional constraint of the form:

\[
c \in X, m(x) = e_m \Rightarrow X \supseteq X_2, X \supseteq X_n
\]

The formal semantics of set expressions is defined by an interpretation \( I \) that maps from set expressions to sets of values (class names) (see Figure 2). We call an interpretation \( I \) a model (solution) of a conjunction \( C \) of constraints if, for each constraint \( X \supseteq s \) in \( C \), \( I(X) \supseteq I(s) \). Our analysis is defined to be the least model of constraints [1]. We write \( \text{lm}(C) \) for the least model of a collection \( C \) of constraints.

### Syntax of set expressions:

\[
\begin{align*}
se & ::= c & \text{class names} \\
& \mid X & \text{set variables} \\
& \mid \text{app}(X_1, m, X_2) & \text{sets from method call} \\
& \mid se \cup se & \text{sets from conditionals} \\
& \mid \top & \text{universe set}
\end{align*}
\]

### Semantics of set expressions:

\[
\begin{align*}
I(X) & \subseteq \text{Val} \\
I(\top) &= \text{Val} \\
I(c) &= \{c\} \\
I(\text{app}(X_1, m, X_2)) & = \{ v \mid c \in I(X_1), m(x) = e_m \in c, I(X_2) \supseteq I(\text{app}(X_1, m, X_2)), v \in I(X_n) \} \\
I(se_1 \cup se_2) & = I(se_1) \cup I(se_2)
\end{align*}
\]

Fig. 2 Set constraints : syntax and semantics

The solving phase closes the initial constraint set \( C \) under the rules \( S \) in Figure 3. Intuitively, the rules propagate values along all the possible dataflow paths in the program. Each propagation rule decomposes compound set constraints into smaller ones, which approximates the steps of the value flows between expressions. Let’s consider the rule for method call. It introduces \( X \supseteq X_n \) if a method \( m \) is defined in a class \( c' \) and if so, adds \( X \supseteq X_2 \) to simulate the parameter binding. Other rules are similarly straightforward from the semantics of corresponding set expressions.

### 3. Multi-level rule transformation

In constraint-based analysis, every set variable has its index, usually denoted by labels of syntactic constructs like expression, statement, field, method, and so on. To design a constraint-based analysis, we first determine an index set for set-variables and then define set-constraint construction rules. Indices of set variables are determined depending on granularity of analysis. For simplicity, we will define an index set in terms of syntactic constructs rather than their labels.

In this section, we describe how to design a new analysis at coarser granularity by transforming the set-constraint rules of an original analysis. We extend this rule-transformation approach into a multi-level approach so that it can relate any two analyses, only if there is a partition relation between the two sets of set-variables for the two analyses.

We first define a new index set for a new sparse analysis based on some syntactic properties, so that it can partition the original index set, and then transform the original construction rules into new ones by replacing the original index of each set variable by the new index.

For example, we can start from the original control flow analysis called XTA which makes one set variable(or index) for each method and field. This
index set $I_{XTA}$ is a set of all methods and fields in a program. Instead of making one set variable for each method and field, it is also possible to make one set variable for a class which includes a set of all methods and fields in it. Starting from XTA, we can design a class-level CFA called CTA which makes one set variable for a class. The index set for CTA is $I_{CTA} = \{ c \mid c \in \mathcal{P} \}$ where $c$ represents the set of all methods and fields in a class $c$. While every method and field has its unique index in $I_{XTA}$, all methods and fields in a class have the same index in $I_{CTA}$, only if they appear in the same class. Naturally $I_{CTA}$ is a partition of $I_{XTA}$ in that a class $c$ represents a set of all methods and fields in the class $c$.

A partition of a set $X$ is a set of nonempty subsets of $X$ such that every element $x$ in $X$ is in exactly one of these subsets called parts. A partition of a set $X$ can also be represented by a partition function $\pi$ from a set $X$ into a set of its parts, which maps an element in a set $X$ into the part it belongs to.

Let $I_1$ and $I_2$ be two index sets such that $I_2$ is a partition of $I_1$, and $\pi$ is a partition function from $I_1$ to $I_2$. If $I_2$ is a new index set for a new sparse analysis such that $I_2$ is a partition of an index set $I_1$ for an original analysis, we can get the construction rules for the new sparse analysis by transforming the original construction rules by applying the partition function $\pi$ to their indices.

The basic idea of this rule transformation is to replace the index of each set variable $X_\kappa$ in the original construction rules by the new index $X_{\kappa(\kappa)}$ where $\kappa$ is an index in $I_1$. This rule-transformation can be formalized as follows:

**Definition 1.** Let $I_1$ and $I_2$ be two index sets such that $I_2$ is a partition of $I_1$ and $\pi$ is a partition function from $I_1$ to $I_2$. Let’s consider a language construct $e_1(\cdots,e_n)$ for a generic expression, which has an index $\kappa$ in $I_1$. If $r$ is a construction rule for the expression of the form:

$$
e_1 \triangleright_1 C_1, \ldots, e_n \triangleright_1 C_n$$

$$
\{(e_1(\cdots,e_n) \triangleright_1 \cup_{1 \leq i \leq n} C_i \cup \{X_\kappa \triangleright se\}
$$

then, the transformed rule $r/\pi$ by applying the partition function $\pi$ is defined as:

$$
e_1 \triangleright_2 C_1, \ldots, e_n \triangleright_2 C_n$$

$$
\{(e_1(\cdots,e_n) \triangleright_2 \cup_{1 \leq i \leq n} C_i \cup \{X_\kappa \triangleright se/\pi\}
$$

where $se/\pi$ is obtained by replacing each set variable $X_\kappa$ in $se$ by $X_{\kappa(\kappa)}$.

For example, let’s consider a rule in XTA for a field variable access $e_1 \cdot f$ in a method $m$ in a class $c$:

$$
e_1 \triangleright_1 C_1$$

$$
e_1.f \triangleright_1 \{X_{c.m} \triangleright X_{c'.f} \mid \forall c' \in X_{c.m}, f \in c'\} \cup C_1$$

If we design a class-level analysis $CTA$ from XTA, then the partition function $\pi$ from $I_{XTA}$ to $I_{CTA}$ is defined as:

$$\pi(c.m) = c \text{ if a method } m \in c$$

$$\pi(c.f) = c \text{ if a field } f \in c$$

Since $\pi(c.m) = c$ and $\pi(c'.f) = c'$, we can transform the original rule into:

$$
e_1 \triangleright_2 C_1$$

$$
e_1.f \triangleright_2 \{X_{c.m} \triangleright X_{c'.f} \mid \forall c' \in X_{c.m}, f \in c'\} \cup C_1$$

by replacing $X_{c.m}$ with $X_c$ and $X_{c'.f}$ with $X_c$. Deriving CTA from XTA will be described in Section 4.4.

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

**Definition 2.** Let $R$ be a set of construction rules whose index set is $I_1$. Let $I_2$ be a partition of $I_1$ such that $\pi$ is a partition function from $I_1$ to $I_2$. The set $R/\pi$ of transformed rules by a partition function $\pi$ is defined as:

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

We denote by $R(p)$ (or $(R/\pi)(p)$) the set of set-constraints constructed by applying the construction rules in $R$ (or $R/\pi$) 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 $R(p)$. The proof is based on the observation in [2] that the least model $C$ is equivalent to the least fixpoint of the continuous function $F$ derived from $C$.

**Theorem 1.** Let $p$ be a program, $R$ be a set of construction rules, and $I_1$ be an index set for $R$. Let $I_2$ be a partition of $I_1$ and $\pi$ be a partition function from $I_1$ to $I_2$. Let $C = R(p)$ and $C_{\pi} = R/\pi(p)$. Then, $\text{Im}(C_{\pi})(X_{\kappa(\kappa)}) \supseteq \text{Im}(C)(X_{\kappa})$ for every index $\kappa$ in $I_1$. 


be recasted as constraint-based analyses. The relations use a variety of formalism, most of them can be defined from their classes. For any given call site, we use 1. Galois insertion: Let $\Delta = \text{Vars}(C)$ and $\Delta_k = \text{Vars}(C_k)$ be the set of set–variables in the set constraint $C$ and $C_k$. Let $D = \Delta \rightarrow \varphi(\text{Val})$ be the domain of interpretations $I$ and $D_k = \Delta_k \rightarrow \varphi(\text{Val})$ be the domain of partitioned interpretations $I_k$. For every interpretation $I$, we define $\alpha(I) = I$, where $I_k, \Delta_k \rightarrow \varphi(\text{Val})$ is defined as $I_k(X_k) = \cup_{\kappa \in \theta} I(X_k)$ for every $X_k \in \Delta_k$, where we denote $\kappa \in \theta$ if $\pi(\kappa) = \theta$ for $\kappa \in I_k$ and $\theta \in I_k$. We define $\gamma(I_{k}) = I$ such that $I'(X_k) = I_k(X_k \mid \gamma(I_{k}))$ for every set variable $X_k \in \Delta_k$. Then, $(D, \alpha, D_k, \gamma)$ is a Galois insertion, since $\alpha(\gamma(I_{k})) = I_k$.

2. Soundness of the operation $\gamma \circ F_{\#}(I_{k}) \supseteq F \circ \gamma(I_k)$: A transformed rule is obtained by replacing each set variable $X_k$ by $X_k \mid \gamma(I_k)$ in the original rule. So, if there is a constraint $X_k \supseteq se$ constructed by an original rule, then there must be a constraint $X_k \mid \gamma(I_k) \supseteq se/\pi$ constructed by the corresponding transformed rule. Let the function $F$ be defined as a collection of equations of the form: $X_k \mid \gamma(I_k) = se$ for every $X_k \in \Delta_k$, and $F_k$ as a collection of equations of the form: $X_k \mid \gamma(I_k) = se/\pi$ for every $X_k \in \Delta_k$. As $X_k$ is replaced by $X_k \mid \gamma(I_k)$ in $F_k$, and every set expression is monotone, $F_k(I_k)(X_k) \supseteq F \circ \gamma(I_k)(X_k)$ for every set variable $X_k$. By the definition of $\gamma$, $\gamma \circ F_{\#}(I_k) \supseteq F \circ \gamma(I_k)$.

4. Multi-level of control flow analyses

Static analyses for call-graph construction have been studied intensively. While their original formulations use a variety of formalism, most of them can be recasted as constraint–based analyses. The common idea is to abstract an object into the name of its class, and to abstract a set of objects into the set of their classes. For any given call site $e.m(\cdot)$, the goal is to compute a set of class names that approximates the run-time values of the receiver expression, and possible methods to be called. Using the multi-level rule-transformation, we will design and relate a number of sparse versions of 0-CFA, called XTA, FTA, MTA, CTA and RTA systematically as in Figure 4, which were designed manually for Java in [5]. They make control flow analysis scalable by making set variables for methods, fields, or classes. RTA makes just one set variable for a program.

Let $p$ denote a program and $\text{Class}$ denote the set of all classes in $p$, more formally, $\text{Class} = \{c | c \in p\}$. Let $\text{Method}$ denote the set of all methods in $p$, $\text{Method} = \{c.m | c \in p, m \in c\}$. Let $\text{Field}$ denote the set of all field variables in $p$, $\text{Field} = \{c.f | c \in p, f \in c\}$. Let $\text{Expr}$ denote the set of all expressions in $p$ and $\text{Var}$ denote the set of all (local or field) variables in $p$.

4.1 0-CFA

We first present 0-CFA for Java based on constraint–based analysis framework [1], which has one set variable for each expression and each variable. Let $I_{0-CFA} = \text{Expr} \cup \text{Var}$ be the index set for 0-CFA, where every expression has one set variable and every variable has one set variable. Every expression $e$ has one set constraints: $X_e \supseteq se$. The set variable $X_e$ is for the classes that the expression $e$’s normal object belongs to. Figure 5 has the rules to construct set constraints for the classes of the objects of each expression $e$. The subscript $e$ of a set variable $X_e$ denotes the expression, to which the rule applies. The subscript $e.f$ denote a field variable $f$ declared in a class $c$. The relation between $e$ and $C$ is read “constraints $C$ are generated from an expression $e$.”

Let’s consider the rules in Figure 5. The new expression will have the newly created object from the class $e_0$, hence $X_e \supseteq \{e_0\}$. The conditional expression will have the objects from $e_1$ or $e_2$, hence $X_e \supseteq X_{e_1} \cup X_{e_2}$. The expression for field variable assignment $e_1.f = e_2$ will assign $e_1$’s objects to the field of $e_2$’s, hence,

$$
\begin{align*}
& e_1 \triangleright_{0} C_1 \quad e_2 \triangleright_{0} C_2 \\
& \quad e_1.f = e_2 \triangleright_{0} \{X_{e_1.f} \supseteq X_{e_2} \mid c \in X_{e_1}\} \cup C_1 \cup C_2
\end{align*}
$$
Let’s consider the rule for an expression $e$ for method call:

$$e_1 \triangleright_0 C_1, e_2 \triangleright_0 C_2 \quad \frac{e_1 \cdot m'(e_2) \triangleright_0 \{X_e \supseteq \text{app}(X_{e_1}, m', X_{e_2})\} \cup C_1 \cup C_2}{e_1 \cdot m'(e_2) \triangleright_0 \{X_e \supseteq \text{app}(X_{e_1}, m', X_{e_2})\} \cup C_1 \cup C_2}$$

The call expression will have the objects returned from the method $m'$, defined as $m'(x) = e_m$ inside the classes $e \in X_{e_i}$ of $e_i$’s objects. Hence, as in the conditional constraint of $X \supseteq \text{app}(X_{e_1}, m', X_{e_2})$, the constraint $X_e \supseteq X_{e_i}$ will be added for parameter binding, and $X_e \supseteq X_{e_i}$ will be added for return.

Let’s consider the rule for try expression $e$. Normal objects are either from $e_1$ or from $e_2$ (after handling), hence $X_e \supseteq X_{e_1} \cup X_{e_2}$. The objects of $x_2$ are for raised exceptions from the try-block, but are approximated by all the subclasses (denoted by $e'_{c_2}$) of its class $c_2$.

The precision of control flow analyses can be improved by integrating static type information into the analysis as in [5].

4.2 XTA

The analysis XTA uses only two groups of set variables: set variables for methods and fields. The number of set variables is thus proportional only to the number of methods and fields, not to the number of expressions. We can design the new analysis XTA from 0-CFA by transforming the construction rules of 0-CFA.

This design decision can be represented by defining the index set for XTA as $I_{XTA} = \text{Method} \cup \text{Field}$. Naturally $I_{XTA}$ is a partition of $I_{0-CFA}$ in that a method $e.m$ represents a set of all expressions in the method $e.m$. Let $\pi_1$ be a partition function from $I_{0-CFA}$ to $I_{XTA}$. Then, a partition function

$$\pi_1: \text{Expr} \cup \text{Var} \rightarrow \text{Method} \cup \text{Field}$$

is defined as

$$\pi_1(e) = e.m \quad \text{if an expression} \ e \in m, m \in c$$

$$\pi_1(x) = e.m \quad \text{if a local} \ x \in m, m \in c$$

$$\pi_1(e.f) = e.f \quad \text{if a field} \ f \in e$$

For the current expression $e$ in a method $m$ in a class $c$, that is, $\pi_1(e) = c.m$, we can design the construction rules of XTA in Figure 6 by transforming the original rules of 0-CFA in Figure 5 with the partition function $\pi_1$.

Let’s consider the rule for an expression $e$ for field variable assignment $e_1.f = e_2$ in Figure 5:

$$e_1 \triangleright_0 C_1, e_2 \triangleright_0 C_2 \quad \frac{e_1 \cdot f = e_2 \triangleright_0 \{X_{e.f} \supseteq X_{e_2} \mid e' \in X_{e_1}\} \cup C_1 \cup C_2}{e_1 \cdot f = e_2 \triangleright_0 \{X_{e.f} \supseteq X_{e_2} \mid e' \in X_{e_1}\} \cup C_1 \cup C_2}$$

Fig. 5 0-CFA
Since $\pi_1(e) = \pi_1(e_1) = \pi_1(e_2) = e.m$ and $\pi_1(e'.f) = e'.f$, this rule is transformed into:

$$e_1 \triangleright_1 C_1 \quad e_2 \triangleright_1 C_2$$

$$e_1.f = e_2 \triangleright_1 C_1 \quad \{X_{e.f} \supseteq X_{e.m} \mid e' \in X_{e.m}\} \cup C_1 \cup C_2$$

where the set variable $X_{e.m}$ for a method instead of an expression collects the values (class names) of the field $f$. Let’s consider the rule for an expression $e$ for method call $e_1.m'(e_2)$ in Figure 5:

$$e_1 \triangleright_0 C_1 \quad e_2 \triangleright_0 C_2$$

$$e_1.m'(e_2) \triangleright_0 \{X_{e.m} \supseteq \text{app}(X_{e_1.m', X_{e_2}})\} \cup C_1 \cup C_2$$

Since $\pi_1(e) = \pi_1(e_1) = \pi_1(e_2) = e.m$, this rule is transformed into:

$$e_1 \triangleright_1 C_1 \quad e_2 \triangleright_1 C_2$$

$$e_1.m'(e_2) \triangleright_1 \{X_{e.m} \supseteq \text{app}(X_{e_1.m', X_{e_2}})\} \cup C_1 \cup C_2$$

where the constraint $X_{e.m} \supseteq \text{app}(X_{e_1.m', X_{e_2}})$ represents the transformed conditional constraint $c' \in X_{e.m}, m'(x) = e_m' \in c' \Rightarrow X_{e.m} \supseteq X_{e_2}, X_{e_2} \supseteq X_{e.m'}$, which is obtained by replacing $X_{e_i}$ with $X_{e_{i'}}$ and replacing $X_{e}$ with $X_{e_{i'}}$ in the conditional constraint of $X_{e} \supseteq \text{app}(X_{e_1}, m', X_{e_2})$.

### 4.3 FTA and MTA

The analysis FTA uses a distinct set variable for each method as in XTA, but uses one set variable for all fields in a class. Intuitively, the set variable unifies the flow information for all fields in a class but not methods. We can design the new sparse analysis FTA by transforming the construction rules of XTA.

Let $\text{Class}_c$ denote the set of classes, each of which includes all the fields of it, except methods. Formally, $\text{Class}_c = \{c_e \mid e \in p\}$ where $c_p$ denotes the set of all the fields in a class $c$. Then the index set for FTA is $I_{FTA} = \text{Method} \cup \text{Class}_c$. $I_{FTA}$ is a partition of $I_{XTA}$ in that $\text{Class}_c$ partitions all field variables in a program into classes, where they are declared. Let $\pi_2$ be a partition function from $I_{XTA}$ to $I_{FTA}$. Then, a partition function $\pi_2 : \text{Method} \cup \text{Field} \rightarrow \text{Method} \cup \text{Class}_c$ is defined as

$$\pi_2(c.m) = c.m \text{ if a method } m \in c$$

$$\pi_2(e.f) = c_p \text{ if a field } f \in e$$

For the current expression $e$ in a method $m$ in a class $c$, we can design the construction rules of FTA by transforming the original rules of XTA in Figure 6.
with the partition function \( \pi_2 \).

Let’s consider the rule for an expression for field variable assignment \( e_{1,f} = e_2 \) in Figure 6:

\[
\frac{e_1 \triangleright_3 C_1 \; e_2 \triangleright_3 C_2}{e_1.f = e_2 \triangleright_1 \{ X_{c,f} \supseteq X_{c,m} \mid c' \in X_{c,m} \} \cup C_1 \cup C_2}
\]

Since \( \pi_2(c.m) = c.m \) and \( \pi_2(c'.f) = c'.f \), this rule is transformed into:

\[
\frac{e_1 \triangleright_2 C_1 \; e_2 \triangleright_2 C_2}{e_1.f = e_2 \triangleright_2 \{ X_{c,f} \supseteq X_{c,m} \mid c' \in X_{c,m} \} \cup C_1 \cup C_2}
\]

Let’s consider the rule for an expression for method call \( e_{1,m'(e_2)} \) in Figure 6:

\[
\frac{e_1 \triangleright_3 C_1 \; e_2 \triangleright_3 C_2}{e_1.m'(e_2) \triangleright_3 \{ X_{c,m} \supseteq \text{app}_{e_1}(X_{c,m}, m', X_{c,m}) \} \cup C_1 \cup C_2}
\]

where the constraint \( \chi_{e_{1,m'}} \triangleright_3 \{ X_{c,m} \supseteq \text{app}_{e_1}(X_{c,m}, m', X_{c,m}) \} \cup C_1 \cup C_2 \), which is obtained by replacing \( X_{c,m} \) with \( X_{c_{1,m'}} \) with \( X_{c_{1,m'}} \) in the constraint of \( \chi_{e_{1,m'}} \triangleright_3 \{ X_{c,m} \supseteq \text{app}_{e_1}(X_{c,m}, m', X_{c,m}) \} \cup C_1 \cup C_2 \).

4.4 CTA

The class-level analysis CTA uses one set variable for one class, so the number of set variables is thus proportional to the number of classes. We can design the new CTA by transforming the construction rules of XTA. We define the index set for CTA as \( I_{\text{CTA}} = \text{Class} = \{ c \mid c \in \text{Class}_M \} \). \( I_{\text{CTA}} \) is a partition of \( I_{\text{XTA}} \) in that a class \( c \) represents all methods and fields in the class. Let \( \pi_3 \) be a partition function from \( I_{\text{XTA}} \) to \( I_{\text{CTA}} \). Then, a partition function \( \pi_3 : \text{Method} \cup \text{Field} \rightarrow \text{Class} \) is defined as \( \pi_3(c.m) = c.m \) and \( \pi_3(c.f) = c.f \).

For the current expression \( e \) in a class \( c \), we can design the construction rules of CTA by transforming the original rules of XTA in Figure 6 with the partition function \( \pi_3 \). Let’s consider the rule for an expression for field variable assignment \( e_{1,f} = e_2 \) in Figure 6:

\[
\frac{e_1 \triangleright_3 C_1 \; e_2 \triangleright_3 C_2}{e_1.f = e_2 \triangleright_1 \{ X_{c,f} \supseteq X_{c,m} \mid c' \in X_{c,m} \} \cup C_1 \cup C_2}
\]

Since \( \pi_3(c.m) = c.m \) and \( \pi_3(c'.f) = c'.f \), this rule is transformed into:

\[
\frac{e_1 \triangleright_2 C_1 \; e_2 \triangleright_2 C_2}{e_1.f = e_2 \triangleright_2 \{ X_{c,f} \supseteq X_{c,m} \mid c' \in X_{c,m} \} \cup C_1 \cup C_2}
\]

Let’s consider the rule for an expression for method call \( e_{1,m'(e_2)} \) in Figure 6:

\[
\frac{e_1 \triangleright_3 C_1 \; e_2 \triangleright_3 C_2}{e_1.m'(e_2) \triangleright_3 \{ X_{c,m} \supseteq \text{app}_{e_1}(X_{c,m}, m', X_{c,m}) \} \cup C_1 \cup C_2}
\]

Since \( \pi_3(c.m) = c.m \), this rule is transformed into:

\[
\frac{e_1 \triangleright_4 C_1 \; e_2 \triangleright_4 C_2}{e_1.m'(e_2) \triangleright_4 \{ X_{c,m} \supseteq \text{app}_{e_1}(X_{c,m}, m', X_{c,m}) \} \cup C_1 \cup C_2}
\]

Let’s consider the rule for an expression for field variable assignment \( e_{1,f} = e_2 \) in Figure 6:

\[
\frac{e_1 \triangleright_3 C_1 \; e_2 \triangleright_3 C_2}{e_1.f = e_2 \triangleright_3 \{ X_{c,f} \supseteq X_{c,m} \mid c' \in X_{c,m} \} \cup C_1 \cup C_2}
\]
Fig. 7 CTA

\[
\begin{align*}
\frac{e_1 \triangleright_4 C_1 \quad e_2 \triangleright_4 C_2}{e_1 \cdot m'(e_2) \triangleright_4 \{X_c \supseteq app_n(X_c, m', X_c)\} \cup C_1 \cup C_2}
\end{align*}
\]

where the constraint \(X_c \supseteq app_n(X_c, m', X_c)\) represents the conditional constraint:

\[
c' \in X_c, m'(x) = e_m' \rightarrow X_c' \supseteq X_c, X_c \supseteq X_c'
\]

which is obtained by replacing \(X_m\) with \(X_c\) and \(X_{m'}\) with \(X_c\) in the constraint of \(X_m \supseteq app_n\).

4.5 RTA

The rapid type analysis RTA uses just one set variable for a program, which unifies the flow information for all methods and fields in a program.

We can design RTA by transforming the construction rules of CTA. We define the index set for RTA as \(I_{RTA} = \{p\}\) where \(p\) is a program.

We can design the construction rules of RTA by transforming the original rules in Figure 7 with the partition function \(\pi_7\).

Let’s consider the rule for the new expression in Figure 7:

\[
\text{new } c_0 \triangleright_4 \{X_c \supseteq \{c_0\}\}
\]

Since \(\pi_7(c) = p\), this rule is transformed into:

\[
\text{new } c_0 \triangleright_7 \{X_p \supseteq \{c_0\}\}
\]

where the set variable \(X_p\) collects all the classes of newly created objects in a program. As there is just one set variable \(X_p\) in RTA, most construction rules in RTA are of the form \(X_p \supseteq X_p\), which has no effects in solving the set-constraints. Let’s consider the rule for an expression for field variable assignment \(e_1.f = e_2\) in Figure 7.

\[
\frac{e_1 \cdot f \triangleright_4 \{X_c \supseteq \{c\}\}}{e_1 \cdot f = e_2 \triangleright_7 \{X_p \supseteq \{c\}\} \cup C_1 \cup C_2}
\]

Since \(\pi_7(c) = p\), this rule is transformed into:

\[
\frac{e_1 \cdot f \triangleright_7 \{X_p \supseteq \{c\}\}}{e_1 \cdot f = e_2 \triangleright_7 \{X_p \supseteq \{c\}\} \cup C_1 \cup C_2}
\]
Let's consider the rule for an expression for method call
\[ e_1, m'(e_2) \] in Figure 7:
\[
\frac{e_1 \triangleright_1 C_1, e_2 \triangleright_1 C_2}{e_1, m'(e_2) \triangleright_1 \{ X_c \supseteq \text{app}_e (X_c, m', X_c) \} \cup C_1 \cup C_2}
\]
Since \( \pi_e (e) = p \), this rule is transformed into:
\[
\frac{e_1 \triangleright_1 C_1, e_2 \triangleright_1 C_2}{e_1, m'(e_2) \triangleright_1 \{ X_p \supseteq \text{app}_e (X_p, m', X_p) \} \cup C_1 \cup C_2}
\]
where the constraint \( X_p \supseteq \text{app}_e (X_p, m', X_p) \) represents the conditional constraint:
\[
e' \in X_p, m'(x) = e_{m'} \in e' \Rightarrow X_p \supseteq X_p, X_p \supseteq X_p,
\]
which is obtained by replacing \( X_c \) and \( X_c \) with \( X_p \) in the conditional constraint of \( X_p \supseteq \text{app}_e (X_p, m', X_p) \).

When we construct a call graph for a method call \( e_1, m(e_2) \) with the result of RTA, we can consider only the methods \( m \) defined in the classes in \( X_p \). Similarly we can also construct call graphs with the results of XTA, MTA, FTA and CTA, by considering only the methods defined in the classes in \( X_{m'} \), \( X_{m''} \) or \( X_m \) respectively.

5. Discussion

The 0-CFA analysis computes the solution by the conventional iterative fixpoint method because the solution space is finite: classes in the program [2]. It has an \( O(n^2 \times C) \) time bound where \( n \) is the number of set variables for expressions and variables and \( C \) is the number of classes in a program [10]. A sparse analysis like XTA or CTA has the same order of time complexity, but the number \( n \) of set variables is much smaller than that of expressions and variables.

There have been several research directions to improve efficiency of constraint-based analysis. The first is to improve analysis time by simplifying set constraints after constructing the whole constraints [11]. The second is to design analyses at a coarser granularity. Sparse exception analyses are designed for Java and SML, and a function(method)-level exception analysis for SML and Java was manually designed and experimented, and it is shown to give the same information for each method as the expression-level analysis [6,7]. It is shown in [8] that the function-level exception analysis for Java can be directly derived by transforming the original expression-level analysis by applying the partition function \( \pi \) such that \( \pi(e) = m \) if an expression \( e \) occurs in a method \( m \), and that the two analyses give the same information on uncaught exceptions for every method and try-block.

Several sparse versions of 0-CFA, called XTA, CTA, MTA and CTA, were designed manually for Java in [5]. They make class analysis scalable by making set variables for methods, fields, or classes. It was shown by experiments that they are fast for large practical programs and give relatively precise information. This idea was also applied to data flow analysis, and to abstract interpretation [12].

The work in [8] assumed that the original analysis is designed at expression-level. So it has the limit which cannot relate two analyses designed at other syntactic levels such as methods and classes by rule transformation. The multi-level rule transformation mechanism can relate any two analyses, only if there is a partition relation between the two sets of set variables for the two analyses.

6. Conclusion

We have extended the rule-transformation approach so that it can be applied to any analyses which are designed at any level, and provide a multi-level mechanism to design constraint-based analysis for Java at coarser granularity by rule transformation. Using this approach, we investigated the relationship between several CFAs for Java systematically to determine the interprocedural control flow information. This framework can be applied to any programming languages and analyses, only if constraint-based analysis can be designed. A further research topic is on equivalence of analysis information. 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, and design other sparse versions of concurrency and security analysis by rule transformation.

References


