ELSEVIER

Information Processing Letters 83 (2002) 79–88

Information Processing Letters

www.elsevier.com/locate/ipl

Managing the granularity of constraint-based analyses by rule transformation ☆

Byeong-Mo Chang

Department of Computer Science, Sookmyung Women's University, Yongsan-ku, Seoul 140-742, Republic of Korea

Received 4 October 2001; received in revised form 16 October 2001
Communicated by M. Yamashita

Abstract

This paper proposes a transformation-based approach to design efficient constraint-based analysis at a larger granularity. In this approach, we can design a less or equally precise but more efficient version of an original analysis by rule transformation. To do this, 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 into new ones by applying the partition. As applications of this approach, we presents sparse versions of side-effect analysis and exception analysis, which give equally precise information for functions as the original ones. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Set-constraints; Set-based analysis; Rule transformation

1. Introduction

Set-based analysis is a static analysis framework that is applicable to functional, logic and object-oriented languages [8,4,7]. In set-based analysis framework, a specific analysis is designed in terms of set-constraint construction rules. Set-based analysis first constructs set-constraints for input programs using the construction rules, and then computes the least solution or model of them.

As noted in [3,4], 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, that has one set-variable (or index) for every expression. However, its efficiency may not be satisfactory for large practical programs [14,11]. In addition, some analyses (like side-effect analysis, exception analysis, and synchronization analysis [10, 14]) are not interested in properties of all expressions. So, it is wasteful to define one set-variable for every expression for this kind of analyses.

This paper proposes a transformation-based approach to design analyses at a larger granularity than at expression-level, in terms of a simple functional language. 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. 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.

☆ This work was supported by a grant No. R01-2000-00286 from Korea Science & Engineering Foundation.

E-mail address: chang@cs.sookmyung.ac.kr (B.-M. Chang).

0020-0190/01/$ – see front matter
PII: S0020-0190(01)00314-3

As applications of this rule transformation, we provide two instances of analysis design by rule transformation. The first one designs a sparse version of an uncaught exception analysis and the second one deals with a side-effect analysis. Both are basically based on function-level and they are shown to give the *same* information for each function as the original analyses.

Section 2 presents basic definitions. Section 3 presents a systematic way to design sparse analyses by rule transformation. Section 4 presents some applications of this rule transformation. Section 5 discusses related work and future research directions.

2. Preliminaries

For presentation brevity, we consider a simple call-by-value functional language whose terms e are defined by

$e ::= c$		constant
	x	variable
	$\lambda x.e$	function
	$e_1\ e_2$	application
	$\texttt{case}\ e_1\ c\ e_2\ e_3$	branch

Expressions in the language are either constants, variables, functions, function applications, conditional branches. Values are either constants or functions. The case expression "$\texttt{case}\ e_1\ c\ e_2\ e_3$" branches to e_2 or e_3 depending on the value of e_1. Let *Var* be a set of program variables.

We review basics for constraint-based analysis in terms of 0-CFA. Each expression e and program variable x has set-variables $\mathcal{X}_e$ and $\mathcal{X}_x$, respectively representing expression's values and variable's bound 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 $\mathcal{X}$ must have the set *se*. The set-expression *se* has six kinds, each of which corresponds to each program construct (see Fig. 1). We write $\mathcal{C}$ for a finite collection of set-constraints.

Semantics of set-expressions naturally follows from their corresponding language constructs. For example, $app(\mathcal{X}_1, \mathcal{X}_2)$ represents the set of values returned from applications of functions in $\mathcal{X}_1$ to 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 Fig. 1). 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 $lm(\mathcal{C})$ of a collection $\mathcal{C}$ of constraints. The constraint system guarantees the existence of the least model because every operator is monotonic and each constraint's left-hand side is a single variable [8]. The solving phase closes the initial constraint-set $\mathcal{C}$ under the solving rules S in Fig. 1. 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 approximate the steps of the value flows between expressions. Consider the rule for application $\mathcal{X} \supseteq app(\mathcal{X}_1, \mathcal{X}_2)$ in Fig. 1. It introduces $\mathcal{X} \supseteq \mathcal{X}_e$ if a function to call has body-expression e, 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.

3. Rule transformation

In this section, we describe how to design an analysis at a larger 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 partitions the original indices, and then transform the original construction rules by applying the partition.

As noted in [3,4], the precision of the analysis depends upon the choice of the finite set of indices of set-variables. We represent index determination as index function

$$I : Expr \cup Var \to Index,$$

where *Expr* is a set of expressions, *Var* is a set of variables, 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. This index determination can be represented as an index function

$$I_E : Expr \cup Var \to Index,$$

Syntax of set-expressions:

$se ::=$	$\mathcal{X}$	set variables
	c	constants
	$\lambda x.e$	lambdas
	$app(\mathcal{X}_1, \mathcal{X}_2)$	sets from function call
	$\mathcal{X}_1 \cup \mathcal{X}_2$	sets from switch
	$\top$	universe set

Semantics of set-expressions:

$$\begin{aligned}
\mathcal{I}(\mathcal{X}) &\subseteq \mathit{Val}\\
\mathcal{I}(\top) &= \mathit{Val}\\
\mathcal{I}(c) &= \{c\}\\
\mathcal{I}(\lambda x.e) &= \{\lambda x.e\}\\
\mathcal{I}(app(\mathcal{X}_1, \mathcal{X}_2)) &= \{v \mid \lambda x.e \in \mathcal{I}(\mathcal{X}_1), \mathcal{I}(\mathcal{X}_x) \supseteq \mathcal{I}(\mathcal{X}_2), v \in \mathcal{I}(\mathcal{X}_e)\}\\
\mathcal{I}(\mathcal{X}_1 \cup \mathcal{X}_2) &= \{v \mid v \in \mathcal{I}(\mathcal{X}_1) \cup \mathcal{I}(\mathcal{X}_2)\}
\end{aligned}$$

Rules $e \rhd \mathcal{C}$ for constructing constraints $\mathcal{C}$ from each expression e:

$$x \rhd \{\mathcal{X}_e \supseteq \mathcal{X}_x\} \qquad c \rhd \{\mathcal{X}_e \supseteq c\}$$

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

$$\frac{e_1 \rhd \mathcal{C}_1 \quad e_2 \rhd \mathcal{C}_2 \quad e_3 \rhd \mathcal{C}_3}{\texttt{case}\ e_1\ c\ e_2\ e_3 \rhd \{\mathcal{X}_e \supseteq \mathcal{X}_{e_2} \cup \mathcal{X}_{e_3}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

Rules S for solving set-constraints

$$\frac{\mathcal{X} \supseteq \mathcal{X}_1 \cup \mathcal{X}_2}{\mathcal{X} \supseteq \mathcal{X}_1 \quad \mathcal{X} \supseteq \mathcal{X}_2} \qquad \frac{\mathcal{X} \supseteq app(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq \lambda x.e}{\mathcal{X} \supseteq \mathcal{X}_e \quad \mathcal{X}_x \supseteq \mathcal{X}_2} \qquad \frac{\mathcal{X} \supseteq \mathcal{Y} \quad \mathcal{Y} \supseteq ae}{\mathcal{X} \supseteq ae}$$

Fig. 1. Set-constraints: syntax, semantics, derivation rules, and solving rules.

where every expression and variable 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 larger granularity, we first need an index function to determine indices of set-variables. Instead of defining one set-variable for one expression, we make one set-variable 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 $I_P : \mathit{Expr} \to \mathit{Index}$ where $I_P(e) = 1$ for every expression 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

function-level analysis by defining one index for each function.

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

$$I_F(x) = owner(x) \quad \text{where } owner(x) = f \text{ if the variable } x \text{ is an argument of a function } f,$$
$$I_F(e) = f \quad \text{if the expression } e \text{ appears in a function } f,$$

where f denotes a function name or its unique index.

While every expression is mapped to its unique index in I_E, a set of expressions are mapped to one index in I_F if they appear in the same function. We can 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_F 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 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 $e = \kappa(e_1, \ldots, e_n)$, where κ is a language construct. If r is a construction rule of the form:

$$\frac{e_1 \rhd_1 \mathcal{C}_1, \ldots, e_n \rhd_1 \mathcal{C}_n}{\kappa(e_1, \ldots, e_n) \rhd_1 \bigcup_{1 \leqslant i \leqslant n} \mathcal{C}_i \cup \{\mathcal{X}_e \supseteq se\}},$$

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

$$\frac{e_1 \rhd_2 \mathcal{C}_1, \ldots, e_n \rhd_2 \mathcal{C}_n}{\kappa(e_1, \ldots, e_n) \rhd_2 \bigcup_{1 \leqslant i \leqslant n} \mathcal{C}_i \cup \{\mathcal{X}_{\pi(e)} \supseteq se/\pi\}},$$

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

For example, we can design a new function-level 0-CFA by transforming the original rules in Fig. 1. We assume all functions are uniquely named as f, g, h, etc., and subscripted as λ_f.

Example 2. Let I_F be an index function for a function-level analysis and π be a partition function such that $I_F = \pi \circ I_E$. Consider the construction rule for case expression e in Fig. 1:

$$\frac{e_1 \rhd_1 \mathcal{C}_1 \quad e_2 \rhd_1 \mathcal{C}_2 \quad e_3 \rhd_1 \mathcal{C}_3}{\texttt{case}\ e_1\ c\ e_2\ e_3 \rhd_1 \{\mathcal{X}_e \supseteq \mathcal{X}_{e_2} \cup \mathcal{X}_{e_3}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

If this expression e appears in a function f, then e_1 and e_2 are also in f. So, we can transform this rule into:

$$\frac{e_1 \rhd_2 \mathcal{C}_1 \quad e_2 \rhd_2 \mathcal{C}_2 \quad e_3 \rhd_2 \mathcal{C}_3}{\texttt{case}\ e_1\ c\ e_2\ e_3 \rhd_2 \{\mathcal{X}_f \supseteq \mathcal{X}_f \cup \mathcal{X}_f\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

which can be simplified to:

$$\frac{e_1 \rhd_2 \mathcal{C}_1 \quad e_2 \rhd_2 \mathcal{C}_2 \quad e_3 \rhd_2 \mathcal{C}_3}{\texttt{case}\ e_1\ c\ e_2\ e_3 \rhd \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

Consider the construction rule for function application e in Fig. 1:

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

If this expression e appears in a function f, then so do e_1 and e_2. So, we can transform this rule to:

$$\frac{e_1 \rhd_2 \mathcal{C}_1 \quad e_2 \rhd_2 \mathcal{C}_2}{e_1\ e_2 \rhd_2 \{\mathcal{X}_f \supseteq app_\pi(\mathcal{X}_f, \mathcal{X}_f)\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

where app_π is a modified set-expression by applying π to app, and its semantics is defined as

$$\mathcal{I}\big(app_\pi(\mathcal{X}_1, \mathcal{X}_2)\big) = \big\{v \mid \lambda_g x.e \in \mathcal{I}(\mathcal{X}_1), v \in \mathcal{I}(\mathcal{X}_g), \mathcal{I}(\mathcal{X}_g) \supseteq \mathcal{I}(\mathcal{X}_2)\big\}.$$

If a function application $e_1 e_2$ is analyzed with this transformed rule, every function in the function f will be considered for this function application.

An analysis is designed by a set of construction rules. So, we can design a sparse version of an

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

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

$$R/\pi = \{r/\pi \mid r \in R\}.$$

To solve the constraints constructed by the transformed rule in Definition 2, we can use the same solving rules as in Fig. 1, but the solving rule for function applications must be modified by applying π as:

$$\frac{\mathcal{X} \supseteq app(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq \lambda x.e}{\mathcal{X} \supseteq \mathcal{X}_{\pi(e)} \quad \mathcal{X}_{\pi(x)} \supseteq \mathcal{X}_2}$$

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 rules by showing that the least model of the transformed constraints $R/\pi(p)$ is a sound approximation of that of the original constraints $R(p)$ for every program p. The proof is based on the observation in [4] 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 A. □

4. 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 an exception analysis and the second one deals with a side-effect analysis. We assume that these analyses are done after CFA and a safe call table *Lam* is available from CFA.

4.1. Exception analysis

We first extend the source language for handling exceptions as

$e ::= \ldots$		
	`raise` s	exception raise
|	`handle` s `as` e_1 `in` e_2	exception handle

For simple presentation, we consider exceptions as constants instead of introducing a data constructor to define exceptions. "`raise` s" raises a constant exception s. The handle-expression "`handle` s `as` e_1 `in` e_2", evaluates e_2 first, which is called a *handlee*-expression. If e_2's result is a normal value, then the value is returned. If e_2 raises some exception s', then it will catch the exception in case $s = s'$ and e_1 will be executed; the exception will be propagated in case $s \neq s'$.

The aim of exception analysis is to determine what exceptions might be resulted from evaluating each expression. To design an exception analysis at expression level, every expression e needs one set-variable $\mathcal{P}_e$ for uncaught exceptions such that:

$s \in Exn = \{s_1, \ldots, s_n\}$
exception names in program p
$s \in Packet = Exn$
raised exceptions
$\mathcal{P}_e \subseteq Packet$
set-variable for expression e

We first define an exception analysis at expression-level by the rules in Fig. 2. Then, we design a sparse version of the exception analysis by rule transformation. In our new sparse version, only two groups of set-variables are considered: set-variables for functions (lambdas) and handlee-expressions. We assume that all functions and handlee-expressions are uniquely named as f, g, h, etc., and they are subscripted as λ_f, or e_g if necessary. The number of set-variables is thus proportional only to the number of functions and handlee-expressions, not to the number of expressions. For each function f, $\mathcal{X}_f$ is a set-variable for the uncaught exceptions inside the function f. The handlee-expression e_g in "`handle` s `as` e_1 `in` e_g" also has a set-variable $\mathcal{X}_g$, which is for uncaught excep-

Rules $e \rhd_1 \mathcal{C}$ for constructing constraints C from each expression e:

$$[\mathrm{VAR}_1]\ x \rhd_1 \emptyset \qquad [\mathrm{C}_1]\ \mathtt{c} \rhd_1 \emptyset \qquad [\mathrm{ABS}_1]\ \frac{e_1 \rhd_1 \mathcal{C}_1}{\lambda x.e_1 \rhd_1 \mathcal{C}_1}$$

$$[\mathrm{APP}_1]\ \frac{e_1 \rhd_1 \mathcal{C}_1 \qquad e_2 \rhd_1 \mathcal{C}_2}{e_1\ e_2 \rhd_1 \{\mathcal{P}_e \supseteq \bigcup_{\lambda x.e' \in Lam(e_1)} \mathcal{P}_{e'} \cup \mathcal{P}_{e_1} \cup \mathcal{P}_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\mathrm{CASE}_1]\ \frac{e_1 \rhd_1 \mathcal{C}_1 \qquad e_2 \rhd_1 \mathcal{C}_2 \qquad e_3 \rhd_1 \mathcal{C}_3}{\mathtt{case}\ e_1\ c\ e_2\ \ e_3 \rhd_1 \{\mathcal{P}_e \supseteq \mathcal{P}_{e_1} \cup \mathcal{P}_{e_2} \cup \mathcal{P}_{e_3}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

$$[\mathrm{RS}_1]\ \frac{e_1 \rhd_1 \mathcal{C}_1}{\mathtt{raise}\ s \rhd_1 \{\mathcal{P}_e \supseteq s\} \cup \mathcal{C}_1}$$

$$[\mathrm{HNDL}_1]\ \frac{e_1 \rhd_1 \mathcal{C}_1 \qquad e_2 \rhd_1 \mathcal{C}_2}{\mathtt{handle}\ s\ \mathtt{as}\ e_1\ \mathtt{in}\ e_2 \rhd_1 \{\mathcal{P}_e \supseteq \mathcal{P}_{e_1} \cup (\mathcal{P}_{e_2} - \{s\})\} \cup\ \mathcal{C}_1 \cup \mathcal{C}_2}$$

Fig. 2. Constructing exception constraints: $\rhd_1$.

Rules $e \rhd_2 \mathcal{C}$ for constructing constraints C from each expression e:

$$[\mathrm{VAR}_2]\ x \rhd_2 \emptyset \qquad [\mathrm{C}_2]\ \mathtt{c} \rhd_2 \emptyset \qquad [\mathrm{ABS}_2]\ \frac{e_1 \rhd_2 \mathcal{C}_1}{\rhd_2 \lambda_g x.e_1 \rhd_2 \mathcal{C}_1}$$

$$[\mathrm{APP}_2]\ \frac{e_1 \rhd_2 \mathcal{C}_1 \qquad e_2 \rhd_2 \mathcal{C}_2}{e_1\ e_2 \rhd_2 \{\mathcal{P}_f \supseteq \bigcup_{\lambda_g x.e' \in Lam(e_1)} \mathcal{P}_g\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\mathrm{CASE}_2]\ \frac{e_1 \rhd_2 \mathcal{C}_1 \qquad e_2 \rhd_2 \mathcal{C}_2 \qquad e_3 \rhd_2 \mathcal{C}_3}{\mathtt{case}\ e_1\ c\ e_2\ \mathtt{else}\ e_3 \rhd_2 \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

$$[\mathrm{RS}_2]\ \mathtt{raise}\ s \rhd_2 \{\mathcal{P}_f \supseteq \{s\}\}$$

$$[\mathrm{HNDL}_2]\ \frac{e_1 \rhd_2 \mathcal{C}_1 \qquad e_g \rhd_2 \mathcal{C}_2}{\mathtt{handle}\ s\ \mathtt{as}\ e_1\ \mathtt{in}\ e_g \rhd_2 \{\mathcal{P}_f \supseteq (\mathcal{P}_g - \{s\})\} \cup\ \mathcal{C}_1 \cup \mathcal{C}_2}$$

Fig. 3. Constructing sparse exception constraints: $\rhd_2$.

tions from e_g. This design decision can be represented by an index function as follows.

Definition 4. An index function $I_1 : Expr \to Index$ is defined as follows:

$$I_1(e) = \begin{cases} g & \text{if } e \text{ is a handlee-expression } e_g \text{ or} \\ & \quad e\text{'s nearest enclosing handlee is } e_g, \\ f & \text{if } e\text{'s nearest enclosing function is } f. \end{cases}$$

Let π be a partition function such that $I_1 = \pi \circ I_E$. To design an sparse analysis, we transform the original construction rules by applying this partition function π to them. Fig. 3 shows the transformed rules for each expression e, assuming that each e appears in a function f, i.e., $\pi(e) = f$.

We first consider the rule for the expression `raise` s. The set-constraint $\mathcal{P}_e \supseteq s$ is simply transformed into $\mathcal{P}_f \supseteq s$, since $\pi(e) = f$. In case of a handle-expression "$e =$ `handle` s `as` e_1 `in` e_g", because

$\pi(e) = \pi(e_1) = f$ and $\pi(e_g) = g$, its original set-constraint $\mathcal{P}_e \supseteq \mathcal{P}_{e_1} \cup (\mathcal{P}_{e_g} - \{s\})$ is transformed into $\mathcal{P}_f \supseteq \mathcal{P}_f \cup (\mathcal{P}_g - \{s\})$, which can be simplified to $\mathcal{P}_f \supseteq \mathcal{P}_g - \{s\}$. In case of a function application, since $\pi(e) = \pi(e_1) = \pi(e_2) = f$, its set-constraint

$$\mathcal{P}_e \supseteq \bigcup_{\lambda_g x.e' \in Lam(e_1)} \mathcal{P}_{e'} \cup \mathcal{P}_{e_1} \cup \mathcal{P}_{e_2}$$

is transformed into

$$\mathcal{P}_f \supseteq \bigcup_{\lambda_g x.e' \in Lam(e_1)} \mathcal{P}_g \cup \mathcal{P}_f \cup \mathcal{P}_f,$$

which can be simplified to

$$\mathcal{P}_f \supseteq \bigcup_{\lambda_g x.e' \in Lam(e_1)} \mathcal{P}_g.$$

The two analyses designed by Figs. 3 and 2 are shown to give the same information on uncaught exceptions for every function and handlee-expression.

Theorem 2. *Let p be a program and π be a partition function such that $I_1 = \pi \circ I_E$. Let $\mathcal{C} = R(p)$ for the rules R in Fig. 2 and $\mathcal{C}_\pi = R/\pi(p)$. Then, $lm(\mathcal{C}_\pi)(\mathcal{X}_f) = lm(\mathcal{C})(\mathcal{X}_{e_f})$ for every function $\lambda_f x.e_f$ and handlee e_f.*

Proof. See Appendix A. □

4.2. Side-effect analysis

We first extend the source language for reference variables as:

$e ::= \cdots$
- $\mid$ $\texttt{new}_g\ x := e_1\ \texttt{in}\ e_2$ creating reference-variable
- $\mid$ $!x$ accesing reference-variable
- $\mid$ $x := e_0$ assignment

The expression "$\texttt{new}_g\ x := e_1\ \texttt{in}\ e_2$" creates a new reference-variable called x and initializes it to the value of e_1. The value of the reference-variable x can be obtained by $!x$ and it may be set to a new value by the assignment.

The aim of the side-effect analysis is to record, for each subexpression, which locations have been created, accessed and assigned [10]. In this analysis, a location will be represented by the program point where it could be created. As in [10], we shall define the annotations $\varphi \in Ann$ by:

$$\varphi ::= \{!\ell\} \mid \{\ell :=\} \mid \{\texttt{new}\ \ell\} \mid \varphi_1 \cup \varphi_2 \mid \emptyset.$$

The annotation $!\ell$ means that the value of a location created at ℓ is accessed, $\ell :=$ means that a location created at ℓ is assigned, and $\texttt{new}\ \ell$ that a new location has been created at ℓ.

Rules $e \rhd_1 \mathcal{C}$ for constructing constraints C from each expression e:

$$[\text{VAR}_1]\ x \rhd_1 \emptyset \qquad [\text{C}_1]\ \texttt{c} \rhd_1 \emptyset \qquad [\text{ABS}_1]\ \frac{e_1 : \mathcal{C}_1}{\lambda x.e_1 \rhd_1 \mathcal{C}_1}$$

$$[\text{APP}_1]\ \frac{e_1 \rhd_1 \mathcal{C}_1 \qquad e_2 \rhd_1 \mathcal{C}_2}{e_1\ e_2 \rhd_1 \{\mathcal{Z}_e \supseteq \bigcup_{\lambda x.e' \in Lam(e_1)} \mathcal{Z}_{e'} \cup \mathcal{Z}_{e_1} \cup \mathcal{Z}_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\text{CASE}_1]\ \frac{e_1 \rhd_1 \mathcal{C}_1 \qquad e_2 \rhd_1 \mathcal{C}_2 \qquad e_3 \rhd_1 \mathcal{C}_3}{\texttt{case}\ e_1\ c\ e_2\ \ e_3 \rhd_1 \{\mathcal{Z}_e \supseteq \mathcal{Z}_{e_1} \cup \mathcal{Z}_{e_2} \cup \mathcal{Z}_{e_3}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

$$[\text{NEW}_1]\ \frac{e_1 \rhd_1 \mathcal{C}_1 \qquad e_2 \rhd_1 \mathcal{C}_2}{\texttt{new}_g\ x := e_1\ \texttt{in}\ e_2 \rhd_1 \{\mathcal{Z}_e \supseteq \{\texttt{new}\ g\} \cup \mathcal{Z}_{e_1} \cup \mathcal{Z}_{e_2}, \mathcal{Z}_x \supseteq \{g\}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\text{DEREF}_1]\ !x \rhd_1 \{\mathcal{Z}_e \supseteq \{!g \mid g \in \mathcal{Z}_x\}\}$$

$$[\text{ASS}_1]\ \frac{e_1 \rhd_1 \mathcal{C}_1}{x := e_1 \rhd_1 \{\mathcal{Z}_e \supseteq \{g := \mid g \in \mathcal{Z}_x\}\} \cup \mathcal{C}_1}$$

Fig. 4. Construction rules for side-effects: $\rhd_1$.

Rules $e \rhd_2 \mathcal{C}$ for constructing constraints C from each expression e:

$[\mathrm{VAR}_2]\ x \rhd_2 \emptyset$ $\qquad$ $[\mathrm{C}_2]\ \mathtt{c} : \emptyset$ $\qquad$ $[\mathrm{ABS}_2]\ \dfrac{e_1 \rhd_2 \mathcal{C}_1}{\lambda_g x.e_1 \rhd_2 \mathcal{C}_1}$

$$[\mathrm{APP}_2]\quad \frac{e_1 \rhd_2 \mathcal{C}_1 \qquad e_2 \rhd_2 \mathcal{C}_2}{e_1\ e_2 \rhd_2 \{\mathcal{Z}_f \supseteq \bigcup_{\lambda_g x.e' \in Lam(e_1)} \mathcal{Z}_g\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\mathrm{CASE}_2]\quad \frac{e_1 \rhd_2 \mathcal{C}_1 \qquad e_2 \rhd_2 \mathcal{C}_2 \qquad e_3 \rhd_2 \mathcal{C}_3}{\mathtt{case}\ e_1\ c\ e_2\ \ e_3 \rhd_2 \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

$$[\mathrm{NEW}_2]\quad \frac{e_1 \rhd_2 \mathcal{C}_1 \qquad e_2 \rhd_2 \mathcal{C}_2}{\mathtt{new}_g\ x := e_1\ \mathtt{in}\ e_2 \rhd_2 \{\mathcal{Z}_f \supseteq \{\mathtt{new}\ g\}, \mathcal{Z}_x \supseteq \{g\}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\mathrm{DEREF}_2]\quad !x \rhd_2 \{\mathcal{Z}_f \supseteq \{!g \mid g \in \mathcal{Z}_x\}\}$$

$$[\mathrm{ASS}_2]\quad \frac{e_1 \rhd_2 \mathcal{C}_1}{x := e_1 \rhd_2 \{\mathcal{Z}_f \supseteq \{g := \mid g \in \mathcal{Z}_x\}\} \cup \mathcal{C}_1}$$

Fig. 5. Constructing sparse constraints for side-effects: $\rhd_2$.

For the analysis at expression-level, every expression e needs one set-variable $\mathcal{Z}_e$ for side-effects of an expression e and every reference variable x needs one set-variable $\mathcal{Z}_x$ for program points where its location is created. The construction rules for this analysis is shown in Fig. 4.

We design a sparse version of this side-effect analysis by transforming the original construction rules in Fig. 4. Instead of making one set-variable for each expression, we make one set-variable $\mathcal{Z}_f$ for each function f, and one set-variable $\mathcal{Z}_x$ for each reference-variable x. This design decision is represented by an index function as follows:

Definition 5. An index function

$I_1 : \mathit{Expr} \cup \mathit{ReferenceVar} \to \mathit{Index}$

is defined as follows:

$I_2(x) = x$ if x is a reference-variable,

$I_2(e) = f$ if e appears in a function f.

Let π be a partition function such that $I_2 = \pi \circ I_E$. Then, we can transform the original construction rules in Fig. 4 into the new construction rules in Fig. 5 by applying the partition function π. We assume $\pi(e) = f$ for each expression e.

We show that the sparse analysis in Fig. 5 gives the same information for every function as the expression-level analysis in Fig. 4.

Theorem 3. *Let p be a program and π be the partition function such that $I_2 = \pi \circ I_E$. Let $\mathcal{C} = R(p)$ for the rules R in Fig. 4 and $\mathcal{C}_\pi = R/\pi(p)$. Then, $lm(\mathcal{C}_{\pi_1})(\mathcal{X}_f) = lm(\mathcal{C})(\mathcal{X}_{e_f})$ for every function $\lambda_f x.e_f$.*

Proof. See Appendix A. □

5. Discussion

In case of 0-CFA, the 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 function-level analysis in Example 2, the order of time complexity may not change, but the number of set-variables to be constructed is the same as the size of $I_F(\mathit{Expr} \cup \mathit{Var})$, which is the number of functions and usually much smaller. In case of the sparse exception analysis, the number of set-variables is proportional only to the number of functions and handlee-expressions, not to 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 $I(Expr \cup Var)$.

There have been several research directions to improve efficiency of set-based analysis. The first direction is to improve analysis time by simplifying set-constraints after constructing the whole constraints [5–7,12]. They usually simplify set-constraints without losing the precision of the original analysis. Basic idea of congruence partitioning in [5] is to partition set-variables based on idempotence and common subexpression relation. Componential set-based analysis has added more relations for partitioning over congruence partitioning [7].

The second direction is to design analyses at a larger granularity. Sparse exception analyses, called function-level analyses, were *manually* designed for SML and Java, respectively [14,13]. The function-level analysis for ML is shown to be competitive in speed and precision by experiments [14]. Recently, several sparse versions of 0-CFA have been designed for Java [11]. They make analysis scalable by making set-variables for methods, fields, or classes. The basic idea of designing analyses at a larger granularity has also been applied in data flow analysis [9], where syntactic tokens are used to group execution traces and coalesce the memory states associated with them, and abstract interpretation [2,3], where a semantic function for every control point is approximated by partitioning control points and defining a new semantic function over it.

In this paper, we assume 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. 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.

Appendix A. Proofs

Proof of Theorem 1. As in [4], 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 $\mathcal{D} = \Delta \to \wp(Val)$ be the domain of interpretations $\mathcal{I}$ and $\mathcal{D}_\pi = \Delta_\pi \to \wp(Val)$ be the domain of partitioned interpretations $\mathcal{I}_\pi$. For every interpretation $\mathcal{I}$, we define $\alpha(\mathcal{I}) = \mathcal{I}_\pi$ where $\mathcal{I}_\pi : \Delta_\pi \to \wp(Val)$ is defined as $(\mathcal{I}_\pi)(\mathcal{X}_m) = \bigcup_{e \in f} \mathcal{I}(X_e)$ for every function $f \in \Delta_\pi$. 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* $\gamma \circ \mathcal{F}_\pi(\mathcal{I}_\pi) \supseteq \mathcal{F} \circ \gamma(\mathcal{I}_\pi)$: Note that the sparse derivation rules are obtained by replacing every set-variable $\mathcal{X}_e$ by $\mathcal{X}_{\pi(e)}$ in the corresponding original rules. So, if there is a constraint $\mathcal{X}_e \supseteq se$ constructed by an original rule, then there must be a constraint $\mathcal{X}_{\pi(e)} \supseteq se/\pi$. 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. □

Proof of Theorem 2. As in the soundness proof, the continuous functions $\mathcal{F}$ and $\mathcal{F}_\pi$ can be defined. We prove this theorem by showing that $lfp(\mathcal{F}_\pi)(\mathcal{X}_f) = lfp(\mathcal{F})(\mathcal{X}_{e_f})$ for every function $\lambda_f x.e_f$ and handlee e_f. By the soundness theorem, $lfp(\mathcal{F}_\pi)(\mathcal{X}_f) \supseteq lfp(\mathcal{F})(\mathcal{X}_{e_f})$. So, we just prove that $lfp(\mathcal{F}_\pi)(\mathcal{X}_f) \subseteq lfp(\mathcal{F})(\mathcal{X}_{e_f})$.

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

Induction hypothesis: Suppose $\mathcal{I}_\pi(\mathcal{X}_f) \subseteq \mathcal{I}(\mathcal{X}_{e_f})$ for every function and handlee f.

Induction step: Let $\mathcal{I}'_\pi = \mathcal{F}_\pi(\mathcal{I}_\pi)$. Then there exists $\mathcal{I}'$ such that $\mathcal{I}' = \mathcal{F}^i(\mathcal{I})$ for some i and $\mathcal{I}'_\pi(\mathcal{X}_f) \subseteq \mathcal{I}'(\mathcal{X}_{e_f})$ for every f.

(1) For every set-variable $\mathcal{X}_f$, suppose $\mathcal{I}'_\pi(\mathcal{X}_f) = \mathcal{I}_\pi(\mathcal{X}_f) \cup \alpha$.
(2) Then, α must be added by some of the rules $[\mathrm{RS}_2]$, $[\mathrm{HNDL}_2]$, and $[\mathrm{APP}_2]$ in Fig. 3.
(3) There must be the corresponding rules $[\mathrm{RS}_1]$, $[\mathrm{HNDL}_1]$, and $[\mathrm{APP}_1]$ in Fig. 3.
(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 $\mathcal{X}_{e_f}$ in some more iterations $\mathcal{F}^i(\mathcal{I})$ by the original rules in Fig. 3, because e is in f. □

Proof of Theorem 3. As in the proof of Theorem 2, this theorem can be proved by induction on the number of iterations in computing $lfp(\mathcal{F}_\pi)$. □

References

[1] D.F. Bacon, P.F. Sweeney, Fast static analysis of C++ virtual function calls, in: Proceedings of ACM Conference on OOPSLA, October 1996.
[2] F. Bourdoncle, Abstract interpretation by dyanamic partitioning, J. Funct. Programming 2 (4) (1992) 407–435.
[3] P. Cousot, R. Cousot, Abstract interpretation and application to logic programs, J. Logic Programming 13 (2–3) (1992) 103–179.
[4] P. Cousot, R. Cousot, Formal language, grammars and set-constraint-based program analysis by abstract interpretation, in: Proceedings of '95 Conference on Funtional Programming Languages and Computer Architecture, June 1995, pp. 25–28.
[5] E. Duesterwald, R. Gupta, M.L. Soffa, Reducing the cost of data flow analysis by congruence partitioning, in: Proceedings of International Conference on Compiler Construction, April 1994.
[6] M. Fahndrich, J.S. Foster, Z. Su, A. Aiken, Partial online cycle elimination in inclusion constraint graphs, in: ACM SIGPLAN Conference on PLDI, June 1998.
[7] C. Flanagan, M. Felleisen, Componential set-based analysis, in: Proceedings of ACM Symposium on Principles of Programming Languages, January 1997.
[8] N. Heintze, Set-based program analysis, Ph.D Thesis, School of Computer Science, Carneige Mellon University, Pittsburgh, PA, 1992.
[9] N.D. Jones, 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.
[10] F. Nielson, H. Nielson, C. Hankin, Principles of Program Analysis, Springer, Berlin, December 1999.
[11] F. Tip, J. Palsberg, Scalable propagation-based call graph construction algorithms, in: Proceedings of ACM Conference on OOPSLA, October 2000.
[12] Z. Su, M. Fahndrich, A. Aiken, Projection merging: Reducing redundancies in inclusion constraint graphs, in: Proceedings of ACM Symposium on Principles of Programming Languages, January 2000.
[13] K. Yi, B.-M. Chang, Exception analysis for Java, in: Proceedings of 1999 ECOOP Workshop on Formal Techniques for Java Programs, Lisbon, Portugal, June 1999.
[14] K. Yi, S. Ryu, A Cost-effective estimation of uncaught exceptions in Standard ML programs, Theoret. Comput. Sci. 237 (1) (2000).