Java 프로그램의 효율적인 분석을 위한 집합-기반 분석의 변환

(Transformation of Constraint-based Analyses for Efficient Analysis of Java Programs)

조 장 우 † 창 병 모 ††
(Jang-Wu Jo) (Byeong-Mo Chang)

요 약 본 논문에서는 Java 프로그램에 대한 집합기반 분석 설계 시, 효율적인 분석을 위해 분석의 단위를 조절하는 생성규칙 변환 방법을 제안한다. 기존의 생성규칙을 변환함으로서 기존의 분석보다는 정확성이 감소하지만 보다 효율적인 분석을 설계할 수 있다. 본 방법을 응용한 예로서 기존의 식 수준의 클래스 분석과 예외 상황 분석에 대해 생성규칙 변환을 사용하여 효율적인 클래스 분석과 예외상황 분석을 설계하였다. 클래스 분석에서는 메소드와 필드 변수 단위의 분석을 설계하였으며, 예외상황 분석에서는 메소드와 try 구문 단위의 분석을 설계하였다. 그리고 예외상황 분석에서는 식 수준의 분석과 변환된 메소드 수준의 분석이 각 메소드에 대해서 동등한 정보를 제공함을 보였다.

키워드 : 분석단위, 생성규칙변환, 분할함수, 제약기반분석

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

Key words : granularity of analysis, transformation of construction rules, partition function, constraint-based analysis

1. 서 론

집합-기반 분석은 정적분석을 위한 프레임워크로 함수형, 논리형 그리고 객체지향형과 같은 다양한 패러다임의 언어들에 적용될 수 있다[1, 2, 3, 4]. 집합-기반 분석은 3 단계로 구성된다. 첫 번째 단계인 설계 단계는 원하는 분석을 위한 집합-관계식을 유도하는 생성규칙들(construction rules)을 정의하는 것이다. 두 번째 단계는 생성규칙들을 이용하여 입력 프로그램에 대하여 집합-관계식을 생성한다. 세 번째 단계는 생성된 집합-관계식에 대한 해를 찾는 단계로 분석 결과를 제공한다.

집합-기반 분석에서 분석의 정확도는 집합-변수들의 인덱스 선택에 달려있다. 일반적으로 분석의 안전성 증명을 위해 이론적인 식-단위(expression-level)의 분석기를 정의한다. 식-단위의 분석에서는 각 식마다 하나의 집합-변수, 즉 인덱스를 갖게된다. 그러나 식-단위의 분석은 실용적인 큰 프로그램을 분석하는데는 효율적이지 못할 뿐만 아니라[5, 6], 부수-효과 분석(side-effect analysis)[7], 예외 분석(exception analysis)[8], 동시성 분석(concurrency analysis)[9] 등의 분석에서는 모든 식들을 분석할 필요가 없다. 따라서 이러한 경우에 모든 식에 대해서 집합-변수를 만드는 것은 낭비가 되

· 본 연구는 한국과학재단 목적기초연구(2000-1-30300-009-2) 지원으로 수행되었음.
· 이 논문은 2002년도 부산외국어대학교 학술연구조성비에 의해 연구되었음.
† 종신회원 : 부산외국어대학교 컴퓨터전자공학부 교수
jjw@taejo.pufs.ac.kr
†† 종신회원 : 숙명여자대학교 컴퓨터학과 교수
chang@cs.sookmyunng.ac.kr
논문접수 : 2001년 9월 10일
심사완료 : 2002년 5월 24일

며 따라서 분석 단위(analysis granularity)를 확대함으로써 분석의 전체 효율을 높일 필요가 있다.

본 논문에서는 확대된 분석 단위의 집합-기반 분석을 설계하는 생성규칙 변환을 제안한다. 생성규칙 변환을 이용하여 정확성이 감소할 수 있지만 효율적인 분석을 설계할 수 있다. 이 변환의 주요 아이디어는 다음과 같다. (1) 집합-변수들(set-variables)을 분할한다(partition). (2) 구성 규칙의 각 집합-변수들을 분할 블록(partition-block)으로 대치함으로써 새로운 구성 규칙을 유도한다.

본 논문에서 제안한 생성규칙 변환을 적용한 예로서 기존의 식 단위의 클래스 분석과 예외 상황 분석에 대해 생성규칙 변환을 적용하여 효율적인 클래스 분석과 예외상황 분석을 설계하였다. 클래스 분석에서는 메소드와 필드 변수 단위의 분석을 설계하였으며, 예외상황 분석에서는 메소드와 try 구문 단위의 분석을 설계하였다. 그리고 예외상황 분석에서는 식 단위의 분석과 변환된 메소드 단위의 분석이 각 메소드에 대해서 동등한 정보를 제공함을 보였다.

지금까지 분석 효율 향상을 위한 여러 연구들이 진행되어 왔다. [4, 10, 11, 12]에서는 생성규칙을 적용하여 집합-관계식을 생성한 후, 생성된 집합-관계식들을 단순화함으로서 분석 시간을 향상시켰다. 이 방법에서는 정확도를 잃지 않는 단순화 방법을 사용하였다. [5, 6, 8]에서는 제어흐름 분석, 예외상황 분석의 설계 시, 보다 큰 분석 단위의 분석을 직접 설계하였다. 이와 같은 방법들은 요약해석과 자료흐름 분석에도 또한 적용되었다 [13, 14, 15, 16].

본 논문의 목적은 보다 큰 분석-단위의 효율적인 분석을 (직접 설계하지 않고) 기존의 생성규칙들의 변환을 통하여 설계할 수 있는 체계적인 방법 혹은 다리를 제공하는 것이다.

본 논문의 구성은 다음과 같다. 2절에서는 Java의 추상구문을 정의하고 집합 기반 분석에 대해 소개하고, 3절에서는 Java의 클래스 분석에 대해 기술한다. 4절에서는 생성규칙 변환을 이용하여 분석을 설계하는 체계적인 방법을 기술한다. 5절에서는 생성규칙 변환을 적용한 예들을 소개하고, 6절에서는 관련 연구들을 기술하고, 7절에서 결론을 맺는다.

2. 기초

본 논문의 명료한 설명을 위해 간단한 자바 언어를 정의하였다. 본 언어는 예외 처리와 관련된 자바 언어의 구문들로 구성된 자바 언어의 부분 언어로 이의 추상구문(Abstract Syntax)은 그림 1과 같다. 프로그램은 클래스 정의들로 구성되고, 클래스 정의는 클래스 헤더와 클래스 본체로 구성된다. 클래스 본체는 변수 선언과

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

그림 1 Java 부분언어의 추상구문

메소드 정의로 구성되고, 메소드 정의는 메소드 이름, 매개변수 그리고 본체로 구성된다. 그리고 모든 식의 결과는 객체이다.배정식의 결과는 오른쪽 부분식의 결과이고, 연속된 식의 결과는 마지막 부분식의 결과이다. 메소드 호출의 결과는 호출된 메소드에서 반환되는 결과이다. try-catch 식은

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

형태이며 try-블록의 결과는 e_0 또는 e_1이다. e_0를 계산하면서 처리되지 않는 예외가 발생되지 않으면 e_0이고, 만약 처리되지 않는 예외가 발생하여 catch 블록에서 처리되면 e_1이다. 그리고 catch 블록에서 처리되지 않으면, try-catch의 결과는 없고, 제어는 메소드 호출 체인을 따라 예외처리기를 찾는다. 또한 하나의 try 구문에 대한 다중 catch 구문은 다음과 같이 중첩된 try 구문을 사용하여 표현 가능하다 :

$$\text{try } (\text{ try } e_0 \text{ catch } (\ c_1\ x_1\ e_1)) \text{ catch } (\ c_2\ x_2\ e_2)$$

예외 e_0는 throw e_0에 의해 발생된다. 그리고 프로그래머는 메소드 정의 시 메소드에서 처리되지 않는 예외들을 throws 절에 명시해야 한다. 예외는 발생되기 전에는 다른 객체들과 같은 특성을 가지므로 클래스로 정의 가능하고, 객체를 생성할 수 있고, 변수에 배정가능하며 매개변수로 전달될 수도 있다. 본 논문에서는 정형적인 의미구조가 [17, 18]의 의미구조와 유사함으로 생략한다.

집합-기반 분석을 이용한 프로그램의 분석은 집합-관계식(set-constraint)을 생성하는 단계와 집합-관계식의 해를 구하는 두 단계로 구성된다[1]. 집합관계식의 형식은 $X \supseteq se$ 이고 X는 집합 변수(set-variable)를 se는 집합 식을 나타낸다. 집합-관계식의 의미는 집합 변수 X

가 집합 식 se가 나타내는 집합을 반드시 포함한다는 것이다. 클래스 분석의 경우, 집합 식은 다음과 같은 형식이다 :

$se \rightarrow c \mid X \mid app(X_1, m, X_2) \mid se \cup se \mid se - \{c_1, ..., c_n\}$

여기서 c는 클래스 이름이다. 그리고 집합-관계식들 간에는 논리곱 관계이다. 집합-관계식들의 집합을 C라고 표기한다. 집합 식의 의미는 분석 대상이 되는 언어에 따라 정해진다. 예를 들어, $app(X_1, m, X_2)$는 집합 변수 X_1이 나타내는 클래스에서 정의된 메소드 m을 집합 변수 X_2의 값을 매개변수로 전달하여 수행한 결과를 의미한다. 집합 식들의 정형적인 의미구조는 그림 2에서와 같이 집합변수를 값들의 집합에 사상하는 해석(Interpretation) Int로 정의할 수 있다. 집합 C의 집합 관계식 $X \supseteq se$에 대해, $Int(X) \supseteq Int(se)$를 만족하는 해석 Int를 해(model)이라고 한다.

본 논문에서 정적 분석은 집합-관계식들의 최소 해로 정의된다. 프로그램에 대해 생성된 집합-관계식들의 최소해는 반드시 존재한다. 그 이유는 모든 연산자들이 단조 연산자들이고 집합-관계식의 왼쪽에는 하나의 집합 변수가 있기 때문이다[1]. 집합 C의 최소해를 $lm(C)$라고 표기한다.

집합 식의 구문 구조 :

se ::=	X_i	집합 변수
|	c	클래스 이름
|	$app(X_i, m, X_i)$	메소드 호출을 위한 집합 식
|	$se \cup se$	조건식을 위한 집합 식
|	$se - \{c_1, ..., c_n\}$	try-catch를 위한 집합 식
|	$\top$	전체 집합

집합 식의 의미 :

$$
\begin{aligned}
I\ nt(X_i) &\subseteq Val \\
Int(\top) &= Val \\
Int(c) &= \{c\} \\
Int(app(X_1, m, X_2)) &= \{v \mid c \in Int(X_i),\ m(x) \\
&= e_m \in c, \\
&\quad v \in Int(X_{c.m}), \\
Int(X_x) &\supseteq Int(X_2)\ \} \\
Int(se_1 \cup se_2) &= Int(se_1) \cup Int(se_2) \\
Int(se_1 - \{c_1, \ldots, c_n\}) &= Int(se_1) - \{c_1, \ldots, c_n\}
\end{aligned}
$$

그림 2 집합 식 : 구문 구조와 의미

$$
\begin{array}{ll}
[\text{New}_1] & \text{new } c \rhd_1 \{X_e \supseteq \{c\}\} \\
[\text{This}_1] & \dfrac{c \text{ is the enclosing class}}{\text{this} \rhd_1 \{X_e \supseteq \{c\}\}} \\
[\text{FieldAss}_1] & \dfrac{e_1 \rhd_1 C_1 \quad id \rhd_1 C_1}{id.x := e_1 \rhd_1 \{X_{c.x} \supseteq X_{e_1} \mid c \in X_{id}, x \in c\} \cup \{X_e \supseteq X_{e_1}\} \cup C_1} \\
[\text{ParamAss}_1] & \dfrac{e_1 \rhd_1 C_1}{x := e_1 \rhd_1 \{X_x \supseteq X_{e_1}, X_e \supseteq X_{e_1}\} \cup C_1} \\
[\text{Seq}_1] & \dfrac{e_1 \rhd_1 C_1 \quad e_2 \rhd_1 C_2}{e_1; e_2 \rhd_1 \{X_e \supseteq X_{e_2}\} \cup C_1 \cup C_2} \\
[\text{Cond}_1] & \dfrac{e_0 \rhd_1 C_0 \quad e_1 \rhd_1 C_1 \quad e_2 \rhd_1 C_2}{\text{if } e_0 \text{ then } e_1 \text{ else } e_2 \rhd_1 \{X_e \supseteq X_{e_1} \cup X_{e_2}\} \cup C_0 \cup C_1 \cup C_2} \\
[\text{FieldVar}_1] & \dfrac{id \rhd_1 C_{id}}{id.x \rhd_1 \{X_e \supseteq X_{c.x} \mid c \in X_{id}, x \in c\} \cup C_{id}} \\
[\text{Param}_1] & x \rhd_1 X_e \supseteq X_x \\
[\text{Throw}_1] & \dfrac{e_1 \rhd_1 C_1}{\text{throw } e_1 \rhd_1 C_1} \\
[\text{Try}_1] & \dfrac{e_0 \rhd_1 C_0 \quad e_1 \rhd_1 C_1}{\text{try } e_0 \text{ catch}(c_1 x_1 e_1) \rhd_1 \{X_e \supseteq X_{e_0} \cup X_{e_1}, X_{x_1} \supseteq \{c_1\}^*\} \cup C_0 \cup C_1} \\
[\text{MethCall}_1] & \dfrac{e_1 \rhd_1 C_1 \quad e_2 \rhd_1 C_2}{e_1.m(e_2) \rhd_1 \{X_e \supseteq app(X_{e_1}, m, X_{e_2})\} \cup C_1 \cup C_2} \\
[\text{MethDef}_1] & \dfrac{e_m \rhd_1 C}{m(x) = e_m \rhd_1 \{X_{c.m} \supseteq X_{e_m}\} \cup C} \\
[\text{ClassDef}_1] & \dfrac{m_i : C_i \quad i = 1, \ldots, n}{\text{class } c = \{\text{var } x_1, \ldots, x_k, m_1, \ldots, m_n\} \rhd_1 C_1 \cup \ldots \cup C_n} \\
[\text{Program}_1] & \dfrac{C_i \rhd_1 C_i \ i = 1, \ldots, n}{C_1, \ldots, C_n \rhd_1 C_1 \cup \ldots \cup C_n}
\end{array}
$$

그림 3 식 단위의 클래스 분석

3. 클래스 분석

먼저 집합기반 분석을 이용한 Java 언어에 대한 클래스 분석에 대해 기술한다[1]. 프로그램의 모든 식 e에 대해 $X_e \supseteq se$와 같은 집합-관계식을 생성한다. 집합 변수 X_e는 식 e가 나타내는 객체들의 클래스들을 의미하기 위한 변수이다. 그림 3은 식 e의 클래스들을 구하기 위한 집합-관계식을 생성하는 생성규칙이다. 집합 변수 X_e의 첨자 e는 생성 규칙이 적용되는 식을 의미하고, 첨자 $c.x$는 클래스 c의 필드 변수 x를 나타내고, $c.m$은 클래스 c의 메소드 m을 나타낸다. 관계 "$e \rhd_1 C$"는 "집합-관계식 C가 식 e로부터 생성된다"는 것을 의미한다.

그림 3의 생성규칙의 의미를 살펴보면 다음과 같다. new 식은 클래스 c의 객체를 생성하는 것이므로, $X_e \supseteq \{c\}$이다. 조건 식은 조건에 따라 e_1 또는 e_2 객체를 가지므로 $X_e \supseteq X_{e1} \cup X_{e2}$이다.

메소드 호출 식은 메소드 m에서 반환된 객체를 가진다. 메소드 $m(x) = e_m$은 객체 e_1의 클래스 $c \in X_{e1}$에서 정의된 메소드이다. 그러므로 해를 구하는 단계에서 그림 4의 해를 구하는 규칙에 의해 $X_e \supseteq X_{c.m}$과 $X_x \supseteq X_{e2}$이 추가된다.

$$\frac{e_1 \rhd_1 C_1 \quad e_2 \rhd_1 C_2}{e_1.m(e_2) \rhd_1 \{X_e \supseteq app(X_{e_1}, m, X_{e_2})\} \cup C_1 \cup C_2}$$

try 식은 객체 e_0 또는 e_1 객체를 가지므로 $X_e \supseteq X_{e0} \cup X_{e1}$이다. 변수 x_1은 try e_0에서 발생해서 catch(c_1 x_1 e_1)에서 처리되는 예외 객체를 나타내므로 클래스 c_1의 모든 하위클래스들 ($\{c_1\}^*$)로 근사할 수 있다.

$$\frac{e_0 \rhd C_0 \qquad e_1 \rhd C_1}{\begin{array}{l}\text{try } e_0 \text{ catch}(c_1\, x_1\, e_1) : \\ \{X_e \supseteq X_{e_0} \cup X_{e_1},\ X_{x_1} \supseteq \{c_1\}^*\} \cup C_0 \cup C_1\end{array}}$$

$$\frac{X \supseteq X_1 \cup X_2}{X \supseteq X_1} \qquad \frac{X \supseteq X_1 \cup X_2}{X \supseteq X_2}$$

$$\frac{X \supseteq app(X_1, m, X_2) \quad X_1 \supseteq c \quad m(x) = e \in c}{X \supseteq X_{c.m}} \qquad \frac{}{X_x \supseteq X_2}$$

$$\frac{X \supseteq \nu \qquad \nu \supseteq ae}{X \supseteq ae}$$

그림 4 해를 구하는 단계의 규칙 S

해를 구하는 단계에서 그림 4의 규칙 S을 집합-관계식 집합 C에 적용한다. 직관적으로, 집합관계식은 프로그램의 가능한 자료 흐름 경로를 따라 값들을 전달한다. 그러므로 규칙 S를 적용하여 복잡한 집합관계식을 단순한 집합관계식으로 분해한다. 메소드 호출 $X \supseteq app(X_1, m, X_2)$에서 호출되는 메소드 m이 클래스 c에 정의되어 있다면 $X \supseteq X_{c.m}$과 $X_x \supseteq X_2$이 추가된다.

$$\frac{X \supseteq app(X_1, m, X_2) \quad X_1 \supseteq c \quad m(x) = e \in c}{X \supseteq X_{c.m} \qquad X_x \supseteq X_2}$$

집합-관계식의 해는 집합-관계식의 고정점(fixpoint)를 구하는 과정으로 계산된다. 이 과정은 프로그램에 존재하는 클래스의 수가 유한하므로 반드시 종료한다. 그리고 분석의 정확성 증명 또한 [3]에서 유도된 연속함수에 대한 고정점을 구하는 계산으로 가능하다.

4. 생성규칙 변환

본 절에서는 생성규칙을 변환을 통해서 보다 큰 분석 단위의 분석을 설계하는 방법을 기술한다. 개략적으로 설명하면, 구문구조에 기반하여 새로운 분석을 위한 집합 변수의 인덱스를 결정하는 규칙을 설계한 후, 이 새로운 인덱스 규칙을 기존의 생성규칙에 적용하여 생성규칙을 변환하는 것이다.

인덱스 결정 규칙은 다음과 같이 인덱스 함수 I : $Expr \cup Name \rightarrow Index$로 표현될 수 있는데 $Expr$은 식들의 집합이고, $Name$ 은 변수와 메소드 이름들의 집합이고, $Index$는 인덱스들의 집합이다. 기존의 분석이 식 단위의 분석이라고 가정하면, 프로그램의 각 식과 이름에 대해 하나씩 고유의 인덱스(집합 변수)가 주어진다. 식 단위의 인덱스 결정 규칙은 인덱스 함수 I_E로 표현할 수 있는데 I_E : $Expr \cup Name \rightarrow Index$는 모든 식과 이름에 하나씩 고유의 인덱스를 사상한다. I_E는 일대일 사상이므로 $X_{IE(e)}$로 간단히 X_e표기한다.

큰 분석 단위의 분석을 설계하기 위해서는 집합변수의 인덱스를 결정하는 인덱스 함수를 정의해야 한다. 식 단위의 분석과 같이 각 식에 하나씩 고유의 인덱스를 부여하는 대신에 여러 식들에 동일한 인덱스를 부여할 수도 있다. 가장 간단한 예로 프로그램의 모든 식들에 동일한 단 하나의 인덱스만을 부여할 수 있다. 이 예는 인덱스 함수 I_P로 표현될 수 있다 I_P : $Expr \cup Name \rightarrow Index$는 각 식 또는 이름 e에 대해 $I_P(e) = 1$이다. 이러한 인덱스 함수는 RTA [19]에서 사용되었다.

또한 언어의 구문 구조에 기반하여 인덱스 함수를 정의할 수 있다. 예를 들어 프로그램의 각 클래스 내의 모든 식과 이름에 대해 동일한 하나의 인덱스를 부여하는 클래스 단위의 분석을 위한 인덱스 함수를 다음과 같이 정의할 수 있다.

예 1 클래스 단위 분석을 위한 인덱스 함수 I_C : $Expr \cup Name \rightarrow Index$는 다음과 같이 정의된다 :

$I_C(c.m) = c$ m이 클래스 c 내에서 정의된 메소드인 경우

$I_C(c.x) = c$ x가 클래스 c 내에서 정의된 필드 변수인 경우

$I_C(e) = c$ e가 클래스 c 내에서 사용된 식인 경우

$I_C(x) = c$ x가 클래스 c 내에서 정의된 메소드의 매개변수인 경우

여기서 c는 클래스 이름 또는 고유의 인덱스를 나타낸다. □

식 단위의 인덱스 함수 I_E는 각 식에 고유의 인덱스를 부여하는 반면, I_C는 한 클래스 내의 모든 식과 필드 변수들에 대해 동일한 하나의 인덱스를 부여한다. 이와 같은 개념을 다음과 같이 분할(partition)을 정의하므로서 일반화 할 수 있다.

정의 1 두 개의 인덱스 함수 I_1, I_2에 대해, $I_2 = \pi \circ I_1$를 만족하는 함수 π가 존재하면, I_2는 I_1의 분할이다. 그리고 π를 I_1에서 I_2로의 분할 함수라고 한다.

I_P와 I_C가 I_E의 분할이라는 것은 정의로부터 자명하다. 분할 함수 π에 대해 $I = \pi \circ I_E$를 만족하는 인덱스 함수 I를 설계하였다면, 분할 함수 π를 기존의 인덱스들에 적용하여 기존의 생성 규칙을 변환할 수 있다. 생성규칙 변환의 기본적인 개념은 기존 생성 규칙의 인덱스 X_e를 새로운 인덱스 $X_{\pi(e)}$로 변환하는 것이다. 이러한 생성규칙 변환을 다음과 같이 정형화할 수 있다.

정의 2 I를 $I = \pi \circ I_E$를 만족하는 인덱스 함수이고, $e = \kappa(e_1, ..., e_n)$ 는 κ가 언어 구조를 나타내는 일반적인 식이라고 하자. 만약 r이 다음의 형태를 가지는 생성규칙이라고 하면,

$$\frac{e_1 \rhd C_1, \ldots, e_n \rhd C_n}{x(e_1, \ldots, e_n) \rhd \bigcup_{1 \le i \le n} C_i \cup \{X_e \supseteq se\}}$$

분할 함수 π를 적용하여 변환된 생성규칙 r/π는 다음과 같이 정의된다.

$$\frac{e_1 \rhd C_1, \ldots, e_n \rhd C_n}{x(e_1, \ldots, e_n) \rhd \bigcup_{1 \le i \le n} C_i \cup \{X_{\pi(e)} \supseteq se/\pi\}}$$

여기서 se/π는 se의 모든 집합 변수 $X_{e'}$를 $X_{\pi(e')}$로 대치하여 구할 수 있다.

예를 들어 그림 4에 있는 기존의 생성규칙을 변환하여, [5]의 클래스 단위의 클래스 분석을 설계할 수 있다.

예 2 I_C가 클래스 단위 분석의 인덱스 함수이고, π가 $I_C = \pi \circ I_E$인 분할 함수라고 하자. 그리고 π가 클래스 c'내의 식이고 생성규칙을 적용할 식이라면, 그림 3의 생성규칙에 π를 적용하여 클래스 단위의 클래스 분석을 다음과 같이 설계할 수 있다.

그림 3의 new-식 e는 :

$$\text{new c} \rhd_1 \{X_e \supseteq \{c\}\}$$

$\pi(e) = c'$이므로, π를 적용하면, 다음과 같이 변환된다 :

$$\text{new c} \rhd_2 \{X_{c'} \supseteq \{c\}\}$$

그림 3의 필드 변수 접근 식 e는 :

$$\frac{id \rhd_1 C_{id}}{id.x \rhd_1 \{X_e \supseteq X_{c.x} \mid c \in X_{id}\} \cup C_{id}}$$

$\pi(id.x) = \pi(id) = c'$ 이고 $\pi(c.x) = c$ 이므로, 다음과 같이 변환된다 :

$$\frac{id \rhd_2 C_{id}}{id.x \rhd_2 \{X_{c'} \supseteq X_c \mid c \in X_{c'},\ x \in c\} \cup C_{id}}$$

그림 3의 if-식 e는 :

$$\frac{e_0 \rhd_1 C_0 \quad e_1 \rhd_1 C_1 \quad e_2 \rhd_1 C_2}{\text{if } e_0 \text{ then } e_1 \text{ else } e_2 \rhd_2 \{X_e \supseteq X_{e_1} \cup X_{e_2}\} \cup C_0 \cup C_1 \cup C_2}$$

식 e가 클래스 c'에 존재하면 e_1 and e_2 또한 클래스 c'에 존재하므로 다음과 같이 변환된다:

$$\frac{e_0 \rhd_2 C_0 \quad e_1 \rhd_2 C_1 \quad e_2 \rhd_2 C_2}{\text{if } e_0 \text{ then } e_1 \text{ else } e_2 \rhd_2 \{X_{c'} \supseteq X_{c'} \cup X_{c'}\} \cup C_0 \cup C_1 \cup C_2}$$

위의 생성규칙은 다음과 같이 단순화 가능하다 :

$$\frac{e_0 \rhd_2 C_0 \quad e_1 \rhd_2 C_1 \quad e_2 \rhd_2 C_2}{\text{if } e_0 \text{ then else } e_2 \rhd_2 C_0 \cup C_1 \cup C_2}$$

그림 3의 메소드 호출 e는 :

$$\frac{e_1 \rhd_1 C_1 \qquad e_2 \rhd_1 C_2}{e_1.m(e_2) \rhd_1 \{X_e \supseteq app(X_{e_1}, m, X_{e_2})\} \cup C_1 \cup C_2}$$

$\pi(e) = \pi(e_1) = \pi(e_2) = c'$이므로, 다음과 같이 변환 가능하다 :

$$\frac{e_1 \rhd_2 C_1 \qquad \rhd e_2 \rhd_2 C_2}{e_1.m(e_2) \rhd_2 \{X_{c'} \supseteq app_\pi(X_{c'}, m, X_{c'})\} \cup C_1 \cup C_2}$$

여기서 app_π 는 app에 π를 적용한 수정된 집합 식으로 의미는 다음과 같이 정의된다.

$$Int(app_\pi(X_1,m,X_2)) = \{v \mid c \in Int(X_1), m(x) = e_m \in c, v \in Int(X_c), Int(X_c) \supseteq Int(X_2)\}$$ □

예 2의 메소드 호출에 대한 변환된 생성규칙을 $e_1.m(e_2)$에 적용하면, $e_1.m(e_2)$이 속한 클래스에서 사용 가능한 모든 클래스들의 메소드 m을 고려한다. 이와 같은 경우 타입 정보를 사용해서 정확도를 높일 수 있다 [5].

이제는 변환된 생성 규칙들의 집합으로 새로운 분석을 정의할 수 있다. 이의 개념을 다음과 같이 정형화할 수 있다.

정의 3 R이 생성 규칙들의 집합이면, 분할 함수 π를 적용하여 변환된 생성규칙들의 집합 R/π은 다음과 같이 정의된다.

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

프로그램에 대해 변환된 생성규칙으로 집합-관계식을 생성하는 방법은 다음과 같다. 집합-관계식을 생성하면서 식 또는 필드 변수를 만나면, 대응되는 집합 변수의 인덱스는 분할 함수를 적용하여 결정되므로 생성규칙을 생성할 수 있다.

예 2처럼 생성규칙 변환 후 해를 구하는 과정에서 *app*와 같은 분석함수가 새로운 집합-변수를 생성한다면, 분석함수의 의미구조가 변경되어야 한다.

이를 반영하기 위해서는 그림 4의 해를 구하는 규칙들에 분할함수를 적용하여 변경해야 한다. 만약 해를 구하는 규칙이 새로운 집합-변수를 생성한다면, 그 변수의 인덱스는 분할 함수를 적용하여 결정될 수 있다. 메소드 적용(method application)에 대한 해를 구하는 규칙은 새로 생성되는 집합변수들에 분할 함수 π를 적용하여 다음과 같이 변경된다 :

$$\frac{X \supseteq app_\pi(X_1, m, X_2) \quad X_1 \supseteq c \quad m(x) = e \in c}{X \supseteq X_{\pi(c.m)} \qquad X_{\pi(x)} \supseteq X_2}$$

예 2의 메소드 적용에 대한 해를 구하는 규칙은 다음과 같이 변경된다 :

$$\frac{X \supseteq app_\pi(X_1, m, X_2) \quad X_1 \supseteq c \quad m(x) = e \in c}{X \supseteq X_c \qquad X_c \supseteq X_2}$$

$R(p)$ (또는 $(R/\pi)(p)$)를 프로그램 p에 R (or R/π)의 생성 규칙들을 적용하여 생성된 집합관계식들의 집합이라고 할 때, 변환된 생성규칙의 안전성을 증명할 수 있다. 증명은 변환된 생성규칙들 $(R/\pi)(p)$의 최소 해가 기존의 생성 규칙들 $R(p)$의 안전한 근사(sound approximation)임을 보이고자 한다. 이 증명은 [3]의 C에서 유도된 연속함수 F의 최소 고정점은 $lm(C)$의 최소 해와 동일하다는 사실에 기반하여 증명할 수 있다.

정리 1 p는 프로그램, R은 생성규칙들의 집합, 그리고 π는 분할 함수이다. 그리고 $C = R(p)$ 이고 $C_\pi = R/\pi(p)$라고 하면, 모든 식 e에 대해 $lm(C_\pi)(X_{\pi(e)}) \supseteq lm(C)(X_e)$가 성립한다.

증명 부록 참조

5. 응 용

생성 규칙변환 방법의 유용성을 보이기 위해 이 방법에 의한 분석기 설계의 예를 두 가지 제시한다. 첫 번째는 메소드 단위의 클래스 분석을 설계하고 두 번째는 예외 분석을 다룬다. 둘 다 기본적으로 메소드-단위를 기반으로 하고 있으며 새로 설계될 예외 분석은 각 메소드에 대해서 기존 분석과 같은 정보를 제공한다.

5.1 클래스 분석

여기서는 변환을 이용하여 분석-단위가 큰 클래스 분석을 설계하는데 메소드들과 필드 변수들을 위한 두 종류의 집합-변수들만을 고려한다. 따라서 집합-변수의 수는 식들의 수가 아니라 메소드들과 필드들의 수에 비례하게 된다. 이러한 설계 결정은 다음과 같은 인덱스 함수로 표현할 수 있다.

정의 4 메소드-단위 분석을 위한 인덱스 함수 I_M : $Expr \cup Name \rightarrow Index$는 다음과 같이 정의된다 :

- $I_M(c.m) = c.m$ 이 클래스 c 내에 정의된 메소드인 경우
- $I_M(c.x) = c.x$ x가 클래스 c 내에 정의된 필드 변수인 경우
- $I_M(x) = c.m$ x가 클래스 c 내에 정의된 메소드 m의 매개변수인 경우
- $I_M(e) = c.m$ e가 클래스 c 내에 정의된 메소드 m에 나타나는 식인 경우

여기서 $c.m$과 $c.x$는 클래스 c의 메소드 m과 필드 x를 나타낸다.

I_M를 메소드-단위 분석을 위한 인덱스 함수이고 π를 $I_M = \pi \circ I_E$인 분할 함수라고 하자. 대상 식 e가 메소드 m'에 나타난다고 즉 $\pi(e) = c'.m'$이라고 가정하면 π를 그림 3의 기존 규칙들에 적용하여 그림 5의 메소드-단위의 클래스 분석을 설계할 수 있다.

그림 3의 필드 변수 사용의 경우 생성 규칙 :

$$\frac{id \rhd_1 C_{id}}{id.x \rhd_1 \{X_e \supseteq X_{c.x} \mid c \in X_{id}\} \cup C_{id}}$$

은 $\pi(id.x) = c'.m'$이고 $\pi(c.x) = c.x$이므로 다음과 같이 변환된다 :

$$\frac{id \rhd_3 C_{id}}{id.x \rhd_3 \{X_{c',m'} \supseteq X_{c.x} \mid c \in X_{c.m}, x \in c\} \cup C_{id}}$$

그림 3의 메소드호출 e의 경우 생성 규칙

$$\frac{e_1 \rhd_1 C_1 \quad e_2 \rhd_1 C_2}{e_1.m(e_2) \rhd_1 \{X_e \supseteq app(X_{e_1}, m, X_{e_2})\} \cup C_1 \cup C_2}$$

$\pi(e) = \pi(e_1) = \pi(e_2) = c'.m'$이므로 다음과 같이 변환된다 :

$$\frac{e_1 \rhd_3 C_1 \quad e_2 \rhd_3 C_2}{e_1.m(e_2) \rhd_3 \{X_{c',m'} \supseteq app_\pi(X_{c',m'}, m, X_{c',m'})\} \cup C_1 \cup C_2}$$

여기서 app_π는 변경된 분석 함수로 그 의미는 π를 적용하여 다음과 같이 정의된다 :

$$Int(app_\pi(X_1,m,X_2)) = \{v \mid c \in Int(X_1),\ m(x) = e_m \in c,\ v \in Int(X_{c.m}),\ Int(X_{c.m}) \supseteq Int(X_2)\}$$

이 분석의 정확도는 [5, 8]에서처럼 타입 정보를 이용함으로써 향상될 수 있다.

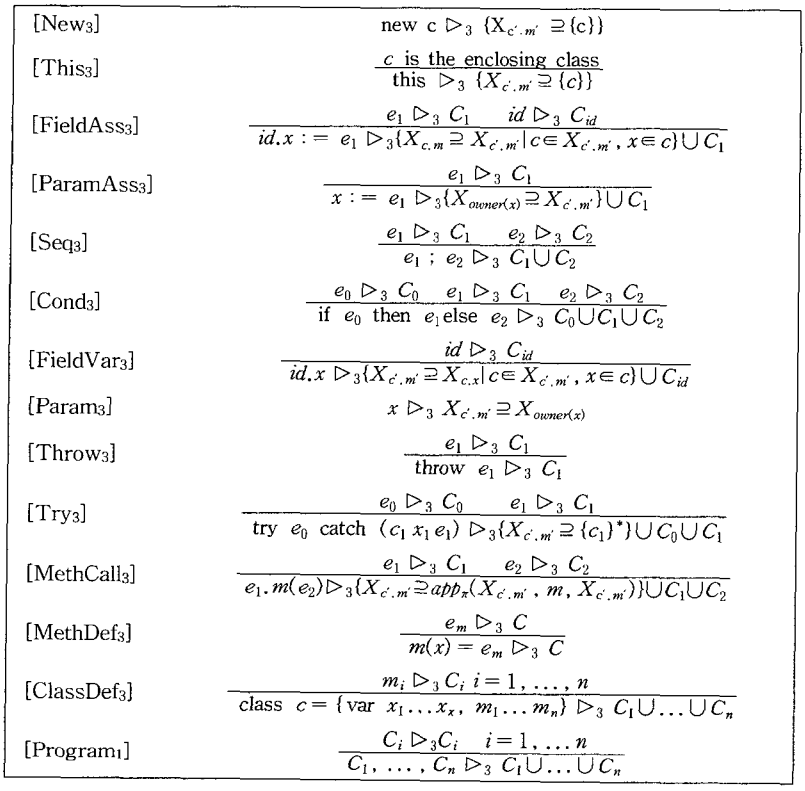

그림 5 메소드 단위의 클래스 분석

5.2 예외 분석

여기서는 먼저 각 식에 대해서 처리되지 않는 예외를 분석하는 집합-기반 분석을 제시한다. 이 분석은 불필요한 예외 처리기를 알려주며 보다 정확한 예외 처리를 위한 정보를 제공한다. 여기서는 이 분석이 클래스 분석[20] 혹은 타입 추론[17, 18]후에 이루어지는 것으로 가정한다. 또한 각 식 e에 대해서 클래스 분석 정보 $Class(e)$가 이미 있다고 가정한다. Java에서는 예외 클래스 역시 다른 클래스와 같은 정상 클래스임을 주의해야 한다.

첫 번째로 그림 6의 각 식을 위한 집합-제약식을 생성하는 규칙들을 살펴보자. 예외 분석을 위해 프로그램의 각 식 e는 $P_e \supseteq se$와 같은 제약식을 갖는다. P_e는 e의 처리되지 않는 예외들의 클래스 이름들을 위한 집합-변수이다. throw 식을 위한 규칙을 살펴보자 :

$$\frac{e_1 \triangleright_1 C_1}{\text{throw } e_1 \triangleright_1 \{P_e \supseteq Class(e_1)\} \cup P_{e_1}\} \cup C_1}$$

이 식은 예외 e_1를 발생시키거나 그 이전에 e_1 내부로부터 처리되지 않는 예외들이 있을 수 있다.

try 식을 위한 규칙을 살펴보자 :

$$\frac{e_0 \triangleright_1 C_0 \quad \triangleright e_1 \quad \triangleright_2 C_1}{\text{try } e_0 \text{ catch } (c_1 x_1 e_1) \triangleright_1 \{P_e \supseteq (P_{e_0} - \{c_1\}^*) \cup P_{e_1}\} \cup C_0 \cup C_1}$$

e_0부터 발생된 예외들은 그들의 클래스가 c_1에 속하면 처리되어 변수 x_1에 바인딩된다. 이 후에 e_1 내부에서도 예외들이 발생할 수 있다. 따라서 $P_e \supseteq (P_{e_0} - \{c_1\}^*) \cup P_{e_1}$이 되며 여기서 $\{c_1\}^*$클래스 c_1의 모든 서브클래스들의 집합을 나타낸다.

메소드 호출을 위한 규칙을 살펴보자 :

$$\frac{e_1 \triangleright_1 C_1 \quad e_2 \triangleright_1 C_2}{e_1.m(e_2) \triangleright_1 \{P_e \supseteq P_{c.m} \mid c \in Class(e_1),\ m(x) = e_m \in c\} \cup \{P_e \supseteq P_{e_1} \cup P_{e_2}\} \cup C_1 \cup C_2}$$

호출 식의 처리되지 않는 예외는 부분식 e_1과 e_2로부터의 것들을 포함한다 : $P_e \supseteq P_{e1} \cup P_{e2}$.

메소드 $m(x) = e_m$은 e_1의 객체의 클래스 즉 $c \in Class(e_1)$에 정의된 것이다. 따라서 $P_e \supseteq P_{c.m}$이 된다.

이제 메소드-단위에서 처리되지 않는 예외들을 분석

$$[\text{New}_1] \quad \text{new } c \rhd_1 \emptyset$$

$$[\text{This}_1] \quad \text{this} \rhd_1 \emptyset$$

$$[\text{FieldAss}_1] \quad \frac{e_1 \rhd_1 C_1}{id.x := e_1 \rhd_1 \{P_e \supseteq P_{e_1}\} \cup C_1}$$

$$[\text{ParamAss}_1] \quad \frac{e_1 \rhd_1 C_1}{x := e_1 \rhd_1 \{P_e \supseteq P_{e_1}\} \cup C_1}$$

$$[\text{Seq}_1] \quad \frac{e_1 \rhd_1 C_1 \quad e_2 \rhd_1 C_2}{e_1 \; ; \; e_2 \rhd_1 \{P_e \supseteq P_{e_1} \cup P_{e_2}\} \cup C_1 \cup C_2}$$

$$[\text{Cond}_1] \quad \frac{e_0 \rhd_1 C_0 \quad e_1 \rhd_1 C_1 \quad e_2 \rhd_1 C_2}{\text{if } e_0 \text{ then } e_1 \text{else } e_2 \rhd_1 \{P_e \supseteq P_{e_0} \cup P_{e_1} \cup P_{e_2}\} \cup C_0 \cup C_1 \cup C_2}$$

$$[\text{FieldVar}_1] \quad \frac{id \rhd_1 C_1}{id.x \rhd_1 C_1}$$

$$[\text{Throw}_1] \quad \frac{e_1 \rhd_1 C_1}{\text{throw } e_1 \rhd_1 \{P_e \supseteq Class(e_1) \cup P_{e_1}\} \cup C_1}$$

$$[\text{Try}_1] \quad \frac{e_g \rhd_1 C_g \quad e_1 \rhd_1 C_1}{\text{try } e_g \text{ catch } (c_1 \, x_1 \, e_1) \rhd_1 \{P_e \supseteq P_{e_g} - \{c_1\}^* \cup P_{e_1}\} \cup C_g \cup C_1}$$

$$[\text{MethCall}_1] \quad \frac{e_1 \rhd_1 C_1 \quad e_2 \rhd_1 C_2}{e_1.m(e_2) \rhd_1 \{P_e \supseteq P_{c.m} | c \in Class(e_1), m(x) = e_m \in c\} \cup \{P_e \supseteq P_{e_1} \cup P_{e_2}\} \cup C_1 \cup C_2}$$

$$[\text{MethDef}_1] \quad \frac{e_m \rhd_1 C_m}{m(x) = e_m \rhd_1 \{P_{c.m} \supseteq P_{e_m}\} \cup C_m}$$

$$[\text{ClassDef}_1] \quad \frac{M_i \rhd_1 C_i \; i = 1, \ldots, m}{\text{class } c = \{\text{var } x_1 \ldots x_n \, M_1 \ldots M_m\} \rhd_1 C_1 \cup \ldots \cup C_m}$$

$$[\text{Program}_1] \quad \frac{C_i \rhd_1 C_i \quad i = 1, \ldots n}{C_1 \ldots C_n \rhd_1 C_1 \cup \ldots \cup C_n}$$

그림 6 식 단위의 예외 분석

하는 분석을 생성 규칙 변환에 의해 설계해보자. 새로운 분석에서는 메소드들과 try-블록들을 위한 두 종류의 집합 변수들만을 고려한다. 각 메소드 m을 위한 집합-변수는 m에 대한 호출 동안에 처리되지 않는 예외들의 클래스 이름들을 위한 집합이다. 식 try e_g catch(c_1 x_1 e_1) 내의 try-블록 e_g는 또한 집합-변수를 갖는데 이는 e_g 내의 처리되지 않는 예외들의 클래스들을 위한 것이다. 따라서 이 분석을 위한 집합 변수들의 수는 식의 수가 아닌 메소드들과 try-블록들의 수에 비례한다. 이러한 설계 결정 사항은 다음과 같이 인덱스 함수로 표현될 수 있다.

정의 5 인덱스 함수 I_X : $Expr \cup MethodName \rightarrow Index$는 다음과 같이 정의된다 :

$I_X(e) = c.g$ e가 클래스 c 내의 try-블록 e_g이거나 그 내부에 나타난 경우

$I_X(e) = c.m$ e가 클래스 c의 메소드 m에 나타나는 경우

$I_X(c.m) = c.m$ m이 클래스 c 내에서 정의되는 경우

π를 $I_X = \pi \circ I_E$인 분할 함수라고 하자. 메소드 단위의 분석을 설계하기 위해서는 기존 규칙을 이 분할 함수를 적용하여 변환해야 한다. 그림 7은 각 대상 식 e에 대해서 $\pi(e) = c'.m'$라고 가정할 때 변환된 규칙을 보여주고 있다.

예를 들어 try-식을 위한 규칙을 살펴보자 :

$$\frac{e_g \rhd_1 C_g \quad e_1 \rhd_1 C_1}{\text{try } e_g \text{catch } (c_1 x_1 e_1) \rhd_1 \{P_e \supseteq P_{e_g} - \{c_1\}^* \cup P_{e_1}\} \cup C_g \cup C_1}$$

대상 식 e가 메소드 $c.m$에 나타나면 e_1역시 그렇다. 따라서 이 규칙은 다음과 같이 단순화될 수 있다.

$$\frac{e_g \rhd_2 C_g \quad e_1 \rhd_2 C_1}{\text{try } e_g \text{catch } (c_1 x_1 e_1) \rhd_2 \{P_{c'.m'} \supseteq P_{c.g} - \{c_1\}^*\} \cup C_g \cup C_1}$$

기존 분석과 새로운 분석은 각 메소드와 try-블록에 대해서 똑 같은 처리되지 않는 정보를 제공한다. 그림 7의 새로운 분석은 메소드와 try-블록에 대해서 그림 6의 분석과 동일함을 다음과 같이 보일 수 있다 :

정리 2 p는 프로그램이고 π는 $I_X = \pi \circ I_E$인 분할 함수라고 하자. 그림 6의 규칙들 R에 대해서 $C = R(p)$이고 $C_\pi = R/\pi(p)$라고 하자. 클래스 c내의 각 메소드와 try-블록에 대해서 다음이 성립한다.

$$lm(C_\pi)(P_{c.f}) = lm(C)(P_{e_f})$$

증명 부록 참조

$$[\mathrm{New}_2] \quad \text{new c} \rhd_2 \emptyset$$

$$[\mathrm{This}_2] \quad \text{this} \rhd_2 \emptyset$$

$$[\mathrm{FieldAss}_2] \quad \frac{e_1 \rhd_2 C_1}{id.x := e_1 \rhd_2 C_1}$$

$$[\mathrm{ParamAss}_2] \quad \frac{e_1 \rhd_2 C_1}{x := e_1 \rhd_2 C_1}$$

$$[\mathrm{Seq2}] \quad \frac{e_1 \rhd_2 C_1 \quad e_2 \rhd_2 C_2}{e_1 ; e_2 \rhd_2 C_1 \cup C_2}$$

$$[\mathrm{Cond}_2] \quad \frac{e_0 \rhd_2 C_0 \quad e_1 \rhd_2 C_1 \quad e_2 \rhd_2 C_2}{\text{if } e_0 \text{ then } e_1 \text{else } e_2 \rhd_2 C_0 \cup C_1 \cup C_2}$$

$$[\mathrm{FieldVar}_2] \quad \frac{id \rhd_2 C_1}{id.x \rhd_2 C_1}$$

$$[\mathrm{Throw}_2] \quad \frac{e_1 \rhd_2 C_1}{\text{throw } e_1 \rhd_2 \{P_{c',m'} \supseteq Class(e_1)\} \cup C_1}$$

$$[\mathrm{Try}_2] \quad \frac{e_g \rhd_2 C_g \quad e_1 \rhd_2 C_1}{\text{try } e_g \text{ catch } (c_1\, x_1\, e_1) \rhd_2 \{P_{c',m'} \supseteq P_{c',g} - \{c_1\}^*\} \cup C_g \cup C_1}$$

$$[\mathrm{MethCall}_2] \quad \frac{e_1 \rhd_2 C_1 \quad e_2 \rhd_2 C_2}{e_1.m(e_2) \rhd_2 \{P_{c',m'} \supseteq P_{c,m} | c \in Class(e_1), m(x) = e_m \in c\} \cup C_1 \cup C_2}$$

$$[\mathrm{MethDef}_2] \quad \frac{e_m \rhd_2 C_1}{m(x) = e_m \rhd_2 C_1}$$

$$[\mathrm{ClassDef}_2] \quad \frac{M_i \rhd_2 C_i \quad i = 1, \ldots, m}{\text{class } c = \{\text{var } x_1 \ldots x_n\, M_1 \ldots M_m\} \rhd_2 C_1 \cup \ldots \cup C_m}$$

$$[\mathrm{Program}_2] \quad \frac{C_i \rhd_2 C_i \quad i = 1, \ldots n}{C_1 \ldots C_n \rhd_2 C_1 \cup \ldots \cup C_n}$$

그림 7 메소드 단위의 예외 분석

6. 토의 및 관련 연구

클래스 분석은 n이 프로그램 내의 식들과 변수들의 수라고 할 때 $O(n^3)$의 시간 복잡도를 갖고 있다. 예제 4의 메소드-단위 분석을 고려하더라도 시간 복잡도 자체는 변하지 않는다. 그러나 집합 변수의 수 n이 메소드들과 필드들의 수로 훨씬 작아진다. 예제 5에 의한 예외 분석의 경우에 집합 변수의 수는 메소드들과 try-블록들의 수에 비례하며 식의 수보다는 훨씬 작다. 일반적으로 I가 새로운 분석을 위한 인덱스 함수이면 이 분석을 위한 집합 변수들의 수는 $I(Expr \cup Name)$와 같다.

집합-기반 분석의 효율을 향상시키기 위한 여러 연구 방향들이 제시되어 왔다. 첫 번째는 제약식들을 완전히 구성한 후에 이들을 단순화하여 분석 속도를 높이는 것이다 [4, 10, 11, 12]. 이 방법들은 보통 원래 분석의 정확도를 잃지 않으면서 집합-관계식을 단순화한다. [10]의 기본 아이디어는 동일(idempotence) 관계와 공동 부분식(common subexpression) 관계를 기반으로 집합 변수들을 분할하는 것이다. [4]의 집합-기반 분석은 분할을 위해 [10]에 더 많은 관계를 도입하였다.

두 번째 방향은 더 큰 분석 단위에서 분석을 직접 설계하는 것이다. 이러한 분석들이 ML과 Java에 대해서 설계되었다[6, 8]. [6]에서 ML를 위한 함수-단위의 예외 분석이 설계되었으며 실험을 통해서 속도와 정확도 면에서 경쟁력이 있음을 보였다. 또한 [8]에서는 Java를 위한 예외 분석과 클래스 분석이 메소드-단위로 설계되었으며 [21]에서 수식-단위 분석과 메소드 단위 분석이 각 메소드에 대해서 같은 정보를 제공함을 이론적으로 증명하였다. 또한 Java 언어를 대상으로 XTA, CTA, MTA 및 RTA라고 하는 0-CFA보다 큰 분석-단위의 분석들이 설계되었다[5]. 이들은 메소드, 필드 혹은 클래스를 위한 집합 변수를 만들어서 클래스 분석의 분석-단위를 조정하였다. 또한 실험적으로 이들이 실용적인 큰 프로그램에 대해서 빠르며 상대적으로 정확한 정보를 제공함을 보였다. 이들 역시 분석의 정확도를 개선하기 위하여[6, 8] 정적인 타입 정보를 이용하였다. 큰 분석-단위의 분석기 설계의 아이디어는 데이타 흐름 분석과 요약 해석 분야에도 적용되었다[13, 14, 15, 16].

기존의 연구들과 본 논문의 차이점은 보다 큰 분석-단위의 효율적인 분석을 (직접 설계하지 않고) 기존의 생

성규칙들의 변환을 통하여 설계할 수 있는 체계적인 방법 혹은 다리를 제공하는 것이다.

7. 결 론

본 연구에서는 기존 분석이 식-수준에서 설계되었으며 인덱스 결정함수 역시 식을 기준으로 정의되었다고 가정하였다. 그러나 본 연구의 아이디어가 식에만 한정될 필요는 없으며 기존 분석이 임의의 수준에서 정의되었다고 가정할 수 있다. 예를 들어 기존 분석이 k-CFA 분석에서처럼 식과 문맥(context)를 기준으로 정의될 수 있으며 이 경우에 0-CFA 분석을 위한 규칙들을 k-CFA의 규칙들 변환하여 구할 수 있다. 본 연구에서는 Java 언어를 기반으로 변환을 위한 틀을 제시하고 있으나 제약-기반 분석이 설계될 수만 있다면 특정 언어나 분석에 한정되지 않는다.

또 하나의 향후 연구 과제는 분석 정보의 동등성에 관한 것이다. 예외 분석처럼 큰 분석단위의 분석이 함수와 같은 구문 구조에 대해서 기존 분석과 같은 정보를 제공할 수 있다. 이러한 동등성에 대한 일반적인 조건을 찾는 것은 흥미로운 남겨진 문제이다. 또한 규칙 변환에 의해 동시성 분석 혹은 보안 분석을 위한 큰 분석 단위의 분석을 설계하는 것도 흥미로운 주제가 될 것이다.

참 고 문 헌

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

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

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

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

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

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

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

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

[10] 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.

[11] 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.

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

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

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

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

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

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

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

[19] D.F. Bacon and P.F. Sweeney, Fast static analysis of C++ virtual function calls. In *Proceedings of*

ACM Conference of Object-Oriented Programming Systems, languages, and Applications, October 1996.

[20] 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.

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

부록. 증명

정리 1 증명 [3]에서처럼 C로부터 연속함수 F를 유도할 수 있으며 같은 방식으로 C_π로부터 연속함수 F_π를 유도할 수 있다. 따라서 이 정리를 $\gamma \circ lfp(F_\pi) \supseteq lfp(F)$임을 보임으로써 증명한다.

이것은 다음을 보임으로써 증명할 수 있다 :

(1) Galois insertion: $\Delta = Vars(C)$이고 $\Delta_\pi = Vars(C_\pi)$라고 하자. $D = \Delta \rightarrow \wp(Val)$를 해석 Int의 도메인이라고 하고 $D_\pi = \Delta_\pi \rightarrow \wp(Val)$를 분할된 해석 Int_π의 도메인이라고 하자. 각 해석 Int에 대해서 $\alpha(Int) = Int_\pi$로 정의되며 $Int_\pi : \Delta_\pi \rightarrow \wp(Val)$는 $m \in \Delta_\pi$에 대해서 $Int_\pi(X_m) = \cup_{e \in m} Int(X_e)$로 정의된다.

각 $m \in \Delta_\pi$에 대해서 $Int'(X_e) = Int_\pi(X_{\pi(e)})$인 $\gamma(Int_\pi) = Int'$를 정의한다. 그러면 $\gamma(Int_\pi) = Int'$이므로 $(D, \alpha, D_\pi, \gamma)$는 Galois insertion이 된다.

(2) $\gamma \circ F_\pi(Int_\pi) \supseteq F \circ \gamma(Int_\pi)$의 안전성: 이 증명을 위해서는 그림 7의 유도 규칙들은 그림 6의 대응되는 유도규칙에서 X_e를 $X_{\pi(e)}$로 대치하여 구할 수 있다는 점을 유의해야 한다. 따라서 집합-관계식 $X_e \supseteq se$가 그림 6의 유도 규칙에 있다면 그림 7의 대응되는 유도 규칙에 $X_{\pi(e)} \supseteq se/\pi$ 가 반드시 있다. F는 각 집합-변수 $X_3 \in \Delta$에 대해서 $X_e = se$ 형태의 등식으로 정의된 함수이고 F_π는 각 집합-변수 $X_{\pi(e)} \in \pi(\Delta)$에 대해서 $X_{\pi(e)} \supseteq se/\pi$ 형태의 등식으로 정의된 함수이다. 각 집합 변수 $X_{e'} \in se$에 대해서 γ의 정의에 의해서 $\gamma(Int_\pi)(X_{e'}) = Int_\pi(X_{\pi(e')}) = S$이다. 또한 F_π의 $X_{\pi(e)} = se/\pi$에서 $X_{e'}$는 $X_{\pi(e')}$로 대치되어 있고 모든 집합 식은 단조 증가함으로 모든 집합-변수 X_e에 대해서 $F_\pi(Int_\pi)(X_{\pi(e)}) \supseteq F \circ \gamma(Int_\pi)(X_e)$이 성립하고 따라서 γ의 정의에 의해 $\gamma \circ F_\pi(Int_\pi) \supseteq F \circ \gamma(Int_\pi)$이 성립한다.

정리 2 증명 안전성 증명에서처럼 F와 F_π를 정의한다. 각 메소드와 try-블록에 대해서 $lfp(F_\pi)(X_{cf}) = lfp(F)(X_{ef})$임을 증명한다. 안전성 정리에 의해서 $lfp(F_\pi)(X_{cf}) \supseteq lfp(F)(X_{ef})$는 증명되었으므로 $lfp(F_\pi)(X_{cf}) \subseteq lfp(F)(X_{ef})$ 만을 증명한다.

이 증명은 $lfp(F_\pi)$를 계산하는 과정에서 반복 횟수에 대한 수학적 귀납법으로 한다.

가정 : 각 메소드와 try-블록에 대해서 $Int_\pi(X_{cf}) \subseteq Int(X_{cf})$라고 가정한다.

단계 : $Int'_\pi = F_\pi(Int_\pi)$이면 어떤 i에 대해서 $Int' = F^i(Int)$이고 $Int'_\pi(X_{cf}) \subseteq Int'(X_{ef})$이 성립하는 Int'이 존재한다.

(1) 각 X_{cf}에 대해서, $Int'_\pi(X_{cf}) = Int_\pi(X_{cf}) \cup \alpha$라고 가정한다.

(2) α는 반드시 그림 7에 있는 $[\text{Throw}_2]$, $[\text{Try}_2]$, $[\text{MethodCall}_2]$에 의해서 추가되었을 것이다.

(3) 그림 6에 대응되는 규칙 $[\text{Throw}_1]$, $[\text{Try}_1]$, $[\text{MethodCall}_1]$이 존재한다.

(4) (3)과 귀납 가정에 의해서 e가 e_f에 나타나고 $F(Int)(X_e) \supseteq \alpha$인 X_e가 반드시 존재함으로 그림 6의 규칙에 의해서 몇 번의 반복 $F^i(Int)$을 거쳐서 X_{ef}에 포함된다.

조 장 우

1992년 서울대학교 계산통계학과 졸업(이학사). 1994년 서울대학교 대학원 전산과학과(이학석사). 1994년 ~ 현재 한국과학기술원 전산학과 박사과정. 1997년 ~ 현재 부산외국어대학교 컴퓨터전자공학부 조교수. 관심분야는 프로그램 분석, 최적화 컴파일러, 모바일 프로그래밍, 개발환경

창 병 모

1988년 서울대학교 컴퓨터공학과 졸업(공학사). 1990년 한국과학기술원 전산학과(공학석사). 1994년 한국과학기술원 전산학과(공학박사). 1994년 ~ 1995년 한국전자통신연구소 박사후 연수 연구원. 1995년 ~ 현재 숙명여자대학교 컴퓨터과학과 부교수. 관심분야는 컴파일러 구성론(정적분석, 코드 최적화), 논리 프로그래밍, 모바일 프로그래밍