자바 언어에 대한 예외 분석 방법 비교
(Comparative Study of Java Exception Analyses)

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

요 약 JDK 자바 컴파일러의 예외 분석은 프로그래머의 throws 선언에 의존하는 프로시저-내 분석이다. 이 분석에서는, 실제 발생하지 않는 예외들이나 실제 발생하는 예외보다 광범위한 예외들이 throws 구문에 선언되어 있는 경우, 적절한 예외 처리를 하지 못하는 문제가 발생한다. 이러한 문제의 해결을 위해 [1,2,3]에서 프로그래머의 선언과 무관한 프로시저-간 예외 분석기가 제시되었다. 본 논문에서는 이 두 분석 방법을 설계 구현하고 실험을 통해서 비교하였다. 실제 사용되는 자바 프로그램에 실험 결과, 프로시져-간 예외분석이 처리되지 않는 예외정보에 대해 프로시져-내 분석보다 정확한 결과를 제공함을 알 수 있었다.

Abstract Exception analysis of JDK Java compiler is an intra-procedural analysis, which depends on programmers' specification on exceptions. This analysis fails to provide correct information on exception handling, in case programmers specify incorrect or broader information on exceptions. To overcome this situation, interprocedural exception analysis is suggested in [1,2,3], which is independent of programmers' specification. This paper designs and implements these two analyses and compares them by experiments. The experiments show that interprocedural exception analysis can detect uncaught exceptions more accurately than intraprocedural exception analysis for realistic Java programs.

1. 서 론

견고한 소프트웨어 개발을 용이하게 하기 위해서 현대의 대부분의 프로그래밍 언어에서는 명시적인 예외 처리 메커니즘을 지원한다. 인터넷 프로그래밍 언어인 자바에서도 예외처리 메커니즘을 지원하고 있다. 자바에서 예외는 Throwable 클래스와 이의 하위 클래스들로 제공되며 또한 프로그래머가 Throwable 클래스의 하위클래스로 새로운 예외 클래스를 정의할 수 있다[4].예외에 관련된 자바 구문으로는 예외를 발생시키는 throw 문장이 있고 발생된 예외를 처리하는 try-catch 문장이 있다. 그리고 메소드 선언 부분에서 처리되지 않는 예외를 선언하는 throws 구문이 있다.

· 본 연구는 한국과학재단 목적기초연구(2000-1-30300-009-2)지원으로 수행되었음.
· 본 연구는 2000년 부산외국어대학교 학술연구조성비 지원으로 수행되었음.
† 종신회원 : 부산외국어대학교 컴퓨터전자공학과 교수
jjw@taejo.pufs.ac.kr
†† 종신회원 숙명여자대학교 전산학과 교수
chang@cs.sookmyung.ac.kr
논문접수 : 2000년 11월 8일
심사완료 : 2001년 5월 7일

예외 메커니즘을 이용하여 프로그램의 안전에 허점이 발생하지 않기 위해서는, 프로그래머가 발생 가능한 예외에 대한 적절한 예외처리기를 설치해야 한다. 그렇지 못한 경우 컴파일러가 미리 분석해서 프로그램의 안전도를 증진시키는 것이 필요하다. JDK 자바 컴파일러는 프로그래머가 각 메소드에 대해 선언한 예외 명세를 기반으로 프로시저-내(intraprocedural) 예외 분석을 한다. 그러나 이 분석에서는, 발생 가능한 예외보다 광범위한 예외나 발생하지 않는 예외들이 throws 구문에 선언되어 있는 경우, 정확한 분석을 할 수 없다는 문제점을 가지고 있다. 이러한 문제점을 개선하기 위해서 [1,2,3]에서는 프로그래머의 선언과 무관한 프로시저-간(interprocedural) 예외 분석을 제시하였다. 이 분석은 선언을 이용하지 않고 예외를 분석함으로써 보다 정확한, 처리되지 않는 예외 정보를 줄 수 있을 뿐만 아니라 프로그래머가 선언한 예외 명세를 보다 정확히 검증할 수 있다.

본 연구에서는 자바 예외상황 분석을 위한 위의 두 방법을 비교함을 목표로 한다. 이를 위해서 프로그래머의 선언에 의존하는 프로시저-내 예외 분석 및 선언과 무관한 프로시저-간 예외 분석을 설계 구현하였으며 실험을 통해

서 실제 프로그램에 적용시켜 봄으로써 두 분석 방법을 비교하였다.

본 연구에서는 Heintze가 [5]에서 제안한 집합-기반 분석(set-based analysis) 기법을 기반으로 분석기를 설계하였으며 구현에 있어서 [1,6,3]에서 제시된 것처럼 모든 식에 대해서 집합-변수를 만드는 대신 try-catch와 메소드 같은 예외에 관련된 구문에 대해서만 집합-변수를 정의함으로써 집합-변수의 수를 줄이고 이를 통해 분석 속도를 향상시켰다. 이렇게 구현된 분석기는 속도 면에서 보다 효율적이면서 각 메소드에 대해서 수식 수준에서 설계된 분석기와 동등한 예외 정보를 제공한다.

본 논문의 구성은 다음과 같다. 2절에서는 연구배경 및 동기를 설명하고, 3절에서는 예외 분석 방법의 설명을 위해 예외와 관련이 있는 자바언어의 구문으로 구성된 가상의 자바 언어를 정의하였고 4절에서는 예외 분석기 설계에 대해서 기술한다. 5절에서는 효율적인 분석기 구현을 위한 고려 사항과 구현 및 실험 결과를 기술하고 6절에서 결론을 맺는다.

2. 동기 및 연구 배경

프로그램 실행 중에 처리되지 않는 예외가 발생하면 프로그램의 실행이 비정상적으로 종료하므로 컴파일 과정에서 처리되지 않는 예외를 검증하는 것은 매우 중요하다. 자바는 메소드 내에서 처리되지 않는 예외들을 메소드 정의 시 throws 구문에 선언하도록 하고 있다. JDK의 자바 컴파일러는 프로그래머가 선언한 이 명세에 의존하는 프로시저-내 예외 분석을 한다. 이러한 조건은 프로그래머에 부담이 되어 필요 이상으로 광범위한 선언을 하거나 실제 발생하지 않는 예외들을 선언할 수 있다. 따라서 프로그래머가 불필요하거나 광범위한 예외들을 선언하는 경우 처리되지 않는 예외에 대한 정확한 분석 결과를 내지 못하며 이로 인해 정확한 예외 처리를 하지 못하는 문제점이 있다. 이와 같은 상황의 간단한 예가 그림 1의 프로그램이다.

```
class demo1{
  void demoProc1 ( ) throws Exception {
       try {
        demoProc2();
       } catch (Exception e) { ; }
  }

  void demoProc2 ( ) throws Exception {
       throw new IOException();
  }
}
```

그림 1 프로그램 예

그림 1의 메소드 demoProc1()에서는 실제 발생하지 않는 예외 Exception을 선언하므로 메소드 demoProc1 ()을 호출하는 코드에서는 불필요한 catch-절을 추가해야 한다. 그리고 메소드 demoProc2()에서는 실제 발생하는 예외 IOException 보다 상위의 예외 Exception을 선언하므로 메소드 demoProc2()을 호출하는 메소드에서는 정확한 예외 처리가 어려운 경우가 발생한다. 이러한 문제는 자바 컴파일러가 프로그래머의 선언에 의존하는 프로시저-내 분석을 수행하기 때문이다.

이러한 문제를 해결하기 위해서 [1,2,3]에서 프로그래머의 선언과 무관한 프로시져-간 예외 분석을 제안하였다. 프로시져-간 예외 분석의 경우, 선언을 이용하지 않고 예외를 분석함으로서 보다 정확한 처리되지 않는 예외 정보를 줄 수 있을 뿐만 아니라 프로그래머가 선언한 예외 명세를 보다 정확히 검증할 수 있다. 그림 1에서 프로시져-간 예외 분석을 수행하면, demoProc2()에서는 IOException이 처리되지 않는 예외이고, demoProc1()에서는 처리되지 않는 예외가 없다는 결과를 제공한다. 또한 demoProc1()의 try 구문에서 실제 처리되지 않는 예외는 IOException이므로, catch 구문에서 광범위한 예외 처리를 했다는 정보를 제공한다.

따라서 본 연구에서는 프로시져-내 예외분석과 프로시저-간 예외 분석을 설계 구현하고 실제 프로그램에 적용시켜 봄으로서 두 분석 방법을 비교하고자 한다.

3. 입력 언어

본 연구에서 예외 분석의 명료한 설명을 위해 예외 처

P	::=	C^*	program
C	::=	class c ext c {var x^* M^*}	class definition
M	::=	$m(x)$[throws c']= e	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 2 ExnJava의 추상구문

리와 관련된 자바 언어의 구문들로 구성된 자바 언어의 부분 언어인 가상의 자바 언어 ExnJava를 정의하였다. 그림 2는 ExnJava에 대한 추상 구문(abstract syntax)이다. 프로그램은 클래스 정의들로 구성되고, 클래스 정의는 클래스 헤더와 클래스 본체로 구성된다. 클래스 본체는 변수 선언과 메소드 정의로 구성되고, 메소드 정의는 메소드 이름, 매개변수 그리고 본체로 구성된다. try e catch(c x e) 구문에서 try e는 try 블록을 의미하고 catch(c x e)에서 c, x, e는 각각 처리할 예외클래스 이름, 처리되는 예외가 바인딩되는 변수, 그리고 예외처리 코드를 의미한다. 또한 자바 언어에서 하나의 try 구문에 대한 다중 catch 구문은 다음과 같이 중첩된 try 구분을 사용하여 표현 가능하다 : try try e_0 catch (c_1 x_1 e_1) catch(c_2 x_2 e_2).

4. 예외분석기의 설계

본 절에서는 집합-기반 분석 기법을 사용해서 프로시져-내 예외 분석기와 프로시져-간 예외 분석기를 설계하였다. 예외 분석을 위해서는 클래스 분석(class analysis) 정보가 필요하다. 클래스 분석이란 어떤 식이 가리키는 객체의 타입(클래스)을 정적으로 유추하는 분석이다[1,7,8,9,10]. 자바에 대한 집합-기반 클래스 분석은 [1,8,9,10]에 기술되어 있다. 본 절에서는 클래스 분석 정보가 존재한다고 가정하고 예외 분석기를 설계한다.

예외 분석의 목표는 각 식에 대해서 실행 중 발생될 수 있는 예외들을 분석을 통해서 구하는 것이다. 따라서 이 분석에서는 모든 식 e 에 대해 처리되지 않는 예외들을 위한 집합 변수 X_e를 정의하고 $X_e \supseteq se$ 형태의 집합-관계식(set-constraints)을 구성한다. 집합 변수 X_e는 식 e에서 발생되었으나 처리되지 않는 예외의 클래스 이름들을 포함하게 된다. 집합-관계식 $X_e \supseteq se$는 집합 변수 X_e가 se가 나타내는 집합을 포함한다는 것을 의미한다.

4.1 프로시저-내 예외 분석

집합-기반 분석은 집합-관계식을 생성하는 생성 규칙들(derivation rules)을 정의함으로써 설계한다. 그림 3은 예외 분석을 위해 각 식 e에 대해 집합-관계식을 생성하는 생성 규칙이다. 그림 3의 "$\triangleright e : C$" 는 집합-관계식 C 는 수식 e에서 생성된다는 것을 의미한다. 그리고 **Class**(e_1)는 식 e_1에 대한 클래스 분석의 결과로 식 e_1이 가리키는 객체의 클래스 집합을 나타낸다.

그림 3의 주요 수식에 대한 생성규칙의 의미는 다음과 같다. throw-식에서 처리되지 않는 예외들은 식 e_1이 나타내는 객체의 클래스와 식 e_1에서 처리되지 않는 예외들이다.

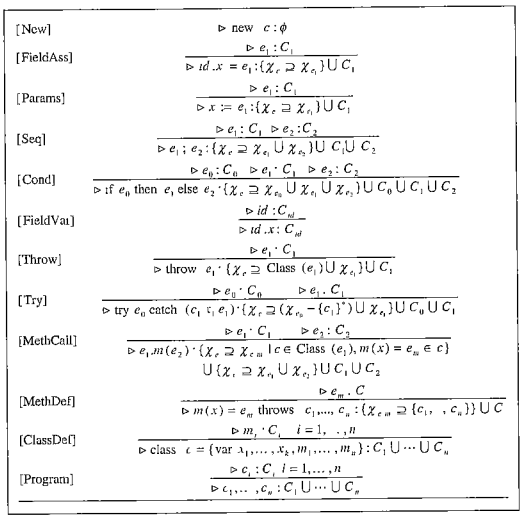

그림 3 프로시져-내 예외분석기의 설계

$$[\text{Throw}] \quad \frac{\triangleright e_1 : C_1}{\triangleright \text{throw } e_1 : \{\chi_e \supseteq \text{Class}(e_1)\} \cup \chi_{e_1}\} \cup C_1}$$

try-catch 식의 처리되지 않는 예외들은 식 e_0 에서 발생하는 예외들 중에서 처리되지 않는 예외들과 예외처리기 e_1 에서 발생하는 예외들이다. e_0 에서 발생된 예외 중에서 예외처리기에서 명시된 c_1 또는 c_1 의 하위클래스의 객체인 경우 처리되어 진다. 따라서 c_1 과 c_1 의 하위클래스들로 구성된 집합을 $(c_1)^*$로 표기하였다.

$$[\text{Try}] \quad \frac{\triangleright e_0 : C_0 \qquad \triangleright e_1 : C_1}{\triangleright \text{try} e_0 \text{ catch}(c_1\, x_1\, e_1) : \{\chi_e \supseteq (\chi_{e_0} - \{c_1\}^*) \cup \chi_{e_1}\} \cup C_0 \cup C_1}$$

메소드 호출식 $e_1.m(e_2)$ 에서 처리되지 않는 예외는 식 e_1및 e_2에서 처리되지 않는 예외들과 식 e_1이 나타내는 객체의 클래스 내의 메소드 m의 throws 구문에 명시된 예외들을 포함한다. 집합 변수 $X_{c.m}$은 그림 3의 [MethodDef] 생성 규칙에서와 같이 클래스 c의 메소드 m에서 throws 구문에 명시된 예외들을 의미한다.

$$[\text{MethCall}] \quad \frac{\triangleright e_1 : C_1 \qquad \triangleright e_2 : C_2}{\triangleright e_1.m(e_2) : \{\chi_e \supseteq \chi_{c.m} \mid c \in \text{Class}(e_1),\ m(x) = e_m \in c\} \cup \{\chi_e \supseteq \chi_{e_1} \cup \chi_{e_2}\} \cup C_1 \cup C_2}$$

4.2 프로시저-간 예외 분석

프로시져-간 예외 분석에서 호출된 메소드의 예외 정보는 메소드의 throws 구문에 의존하지 않고 호출된 메소드에 대한 예외 분석을 수행함으로서 구한다. 따라서 메소드 호출 $e_1.m(e_2)$ 에 대한 처리되지 않는 예외들은

식 e_1및 e_2에서 처리되지 않는 예외들과 호출된 메소드 m의 예외 분석의 결과인 $X_{c.m}$을 포함하게 된다. 그러므로 프로시저-간 예외 분석의 설계에서는 그림 3의 생성 규칙 중 $X_{c.m}$을 구하는 [MethDef] 규칙만을 다음과 같이 변경하면 된다.

$$[\text{MethDef}]\quad \frac{\triangleright e_m : C_m}{\triangleright m(x) = e_m \text{ throws} c_1,\ldots,c_n : \{X_{c.m} \supseteq X_{e_m}\} \cup C_m}$$

그림 3의 생성 규칙을 이용하여 입력 자바 프로그램에 대한 집합 관계식을 구성하고 구성된 집합-관계식들의 해를 구하면 각 식에서 처리되지 않는 예외들을 구할 수 있다. 이 정보를 이용해서 throws 구문에 광범위한 선언과 필요하지 않은 선언을 검증할 수 있고 또한 광범위한 예외 처리기와 불필요한 예외 처리기를 검증할 수 있다.

5. 구현 및 실험

본 절에서는 집합-기반 분석을 이용한 예외 분석기의 효율적인 분석기를 구현하기 위한 고려사항과 구현 환경 그리고 실험에 대해 기술한다.

5.1 효율적인 분석기 구현을 위한 고려 사항

식-단위의 예외 분석은 이론적으로는 정확한 분석 결과를 제공하지만 집합 변수의 수가 식의 수 만큼 필요하므로 분석 속도 면에서 실용적이지 못하다. 집합-기반 분석의 시간 복잡도는 집합 변수의 수가 N 이라고 할 때 N^3 이므로 실제적인 자바 프로그램의 분석에는 적절하지 않다. 이러한 문제를 해결하기 위해서 분석의 단위를 조절하는 연구가 진행되어 왔다[2,3,7,10]. 따라서 본 연구에서는 분석 속도 면에서 실용적인 분석을 위해서 메소드와 try 블록에만 집합-변수를 정의하여 집합-변수들의 수를 줄임으로서 효율적인 예외 분석기를 구현한다. 이와 같이 분석의 단위를 조절하는 기법은 [1,6]의 예외 분석에 성공적으로 적용된 바 있다.

다음은 예외와 관련된 식에 대한 생성 규칙이다. $f \triangleright e$ 는 식 e 가 메소드 f 내의 식이란 것을 의미한다. X_f 는 메소드 f 내에서 발생되었으나 처리되지 않는 예외들을 위한 집합-변수이며 X_{e_g} 는 try-블록 e_g에서 발생되었으나 처리되지 않는 예외들을 위한 집합-변수이다.

throw e_1이 메소드 f의 구문일 때 메소드 f에 대한 집합 변수 X_f 는 처리되지 않는 예외 클래스 **Class** (e_1)를 다음과 같이 포함한다.

$$[\text{Throw}]_m \quad \frac{f \triangleright e_1 : C_1}{f \triangleright \text{throw } e_1 : \{\chi_f \supseteq \text{Class}(e_1)\} \cup C_1}$$

try 블록 e_g에서 처리되지 않는 예외들은 집합 변수 X_{e_g}가 포함하고 있으므로 try-cath 구문에서 처리되지 않은 예외들은 다음 규칙처럼 e_g에서 발생한 예외들 중에서 클래스 c_1의 하위클래스들을 제외한 예외들이다. e_1에서 처리되지 않은 예외들은 가정에 의해서 이미 X_f에 포함되어 있다.

$$[\text{Try}]_m \quad \frac{f \triangleright e_g : C_g \qquad f \triangleright e_1 : C_1}{f \triangleright \text{try} e_g \text{ catch}(c_1 x_1 e_1) : \{\chi_f \supseteq \chi_{e_g} - \{c_1\}\} \cup C_g \cup C_1}$$

메소드 호출 $e_1.m(e_2)$에서 처리되지 않은 예외들은 다음 규칙처럼 호출된 메소드 m에서 처리되지 않은 예외들이다.

$$[\text{MethCall}]_m \quad \frac{f \triangleright e_1 : C_1 \qquad f \triangleright e_2 : C_2}{f \triangleright e_1.m(e_2) : \{\chi_f \supseteq \chi_{c.m} \mid c \in \text{Class}(e_1), m(x) = e_m \in c\} \cup C_1 \cup C_2}$$

프로시저-내 분석에서는 호출된 메소드 m에서 처리되지 않은 예외 $X_{c.m}$은 4절과 같이 메소드 m 의 throws 구문에 명시된 예외들을 나타내고 프로시저-간 분석에서는 메소드 m에서 처리되지 않은 예외 $X_{c.m}$은 m의 내부의 식에서 실제 발생 가능한 예외들 중에서 처리되지 않는 예외들을 포함하게 된다.

이와 같이 구현된 분석기가 각 메소드에 대해서 4절의 분석기와 동등한 예외 정보를 제공함을 다음 정리를 통해 보인다. 프로그램 pgm 으로부터 4절의 생성 규칙을 적용하여 만들어진 집합 관계식들은 C이고 5절의 생성 규칙을 적용하여 만들어진 집합 관계식들은 C_π라고 하자. 집합-관계식들 C의 최소 해(least model)는 $lm(C)$로 표시한다. 다음 정리는 각 메소드와 try-블록에 대해서 집합 관계식 C_π는 집합 관계식 C와 동일한 해를 제공함을 나타낸다.

정리 1(동등성) 메소드와 try-블록을 나타내는 임의의 집합-변수 X_f에 대해, $lm(C_\pi)(X_f) = lm(C)(X_f)$ 이다.

증명. [2]

5.2 구현 및 실험

예외 분석기는 2-패스로 구현된다. 첫 번째 패스에서는 입력 프로그램에 대해 집합-관계식을 구성한다. lex와 yacc을 이용하여 예외 분석을 위한 집합-관계식 구성 규칙을 yacc 의 시맨틱 액션 형태로 기술한다. 두 번째 패스에서는 구성된 집합-관계식의 해를 계산한다. 집합-관계식의 해는 집합-관계식의 고정점(fixpoint)를 구하는 과정으로 계산된다. 이 과정은 프로그램에 존재하는 예외 클래스의 수가 유한하므로 반드시 종료하게 된다.

표 1 대상 프로그램의 간략한 설명

프로그램	간략한 설명	클래스 수	메소드 수	라인 수
Statistician	클래스의 메소드에 대한 간단한 통계	1	1	387
JavaBinHex	BinHex(.hqx) 압축복원	1	3	300
JHLZIP	ZIP 압축	2	11	425
JHLUNZIP	ZIP 압축복원	1	3	287
com.ice.tar	유닉스의 tar 압축 및 복원	10	141	4045
Jess-Rete	Jess의 Reasoning Engine	1	98	1667

표 2 대상 프로그램의 예외에 관련된 구문적 특징

프로그램	발생하는 예외의 종류	throws 구문 수	catch 구문 수	처리되지 않는 예외 수
Statistician	3	1	24	1
JavaBinHex	2	0	8	0
JHLZIP	3	4	8	4
JHLUNZIP	2	1	3	1
com.ice.tar	6	40	21	41
Jess-Rete	7	38	10	33

표 3 광범위한 throws 구문과 불필요한 throws 구문

프로그램	광범위한 throws		불필요한 throws	
	프로시저-간	프로시저-내	프로시저-간	프로시저-내
Statistician	0	0	0	0
JavaBinHex	0	0	0	0
JHLZIP	2	1	0	0
JHLUNZIP	1	0	0	0
com.ice.tar	3	1	4	0
Jess-Rete	1	1	6	0

표 4 광범위한 예외처리기와 불필요한 예외처리기

프로그램	광범위한 catch		불필요한 catch	
	프로시저-간	프로시저-내	프로시저-간	프로시저-내
Statistician	0	0	1	1
JavaBinHex	1	1	0	0
JHLZIP	1	1	0	0
JHLUNZIP	1	0	0	0
com.ice.tar	4	2	0	0
Jess-Rete	3	2	0	0

실험은 소스 코드가 공개되고 자주 사용되는 자바 응용 프로그램과 자바 애플릿을 대상으로 했다. 여기서 사용된 자바 프로그램들은 [11,12]에서 구할 수 있다. 대상 프로그램은 300에서 4000 라인 정도의 프로그램들로 간략한 설명과 구문적인 특징들은 표 1과 표 2에 기술되어 있다.

대상 프로그램에 대해 프로시저-내와 프로시저-간 예외 분석을 적용한 결과는 표 3과 표 4와 같다. 표 3에서 불필요한 throws는 발생하지 않는 예외들을 throws 구문에 선언한 경우이고 광범위한 throws 는 발생 가능한 예외 클래스보다 상위의 클래스들을 throws 구문에 선언한 경우이다. 그리고 표 4에서 불필요한 catch는 발생하지 않는 예외에 대해 catch 구문을 작성한 경우이고, 광범위한 catch는 발생하는 예외보다 상위의 예외 클래스로 catch 구문을 작성한 경우이다. 실험 결과 프로시저-간 분석이 프로시저-내 분석에 비해 보다 정확한 throws 와 catch 구문의 사용에 대한 정보를 제공함을 알 수 있었다. 6개의 대상 프로그램의 84개의 throws 구문 중에서 프로시저-간 분석의 경우 7개의 광범위한 throws 구문과 10개의 불필요한 throws 구문을 발견하였고, 프로시저-내 예외분석의 경우 3개의 광범위한 throws 구문과 0개의 불필요한 throws 구문을 발견하였다. 또한 74개의 catch 구문 중에서 프로시저-간 예외분석은 10개의 광범위한 catch 구문을 발견하였고, 프로시저-내 분석의 경우 6 개의 광범위한 catch 구문을 발견하였다.

이와 같이 자바 컴파일러와 같이 프로시저-내 분석을 하면 불필요하고 광범위한 throws 구문과 catch 구문을 검증하지 못해 적절한 예외처리가 어려운 경우가 발생하는 것을 알 수 있다.

표 5 분석속도(단위:초)

프로그램	프로시저-내			프로시저-간			속도비
	생성시간 (패스-1)	해를 구하는시간 (패스-2)	전체시간(A)	생성시간 (패스-1)	해를 구하는시간 (패스-2)	전체시간(B)	B/A
Statistician	0.02	0.01	0.03	0.02	0.01	0.03	1
JavaBinHex	0.02	0.01	0.03	0.02	0.01	0.03	1
JHLZIP	0.02	0.01	0.03	0.02	0.01	0.03	1
JHLUNZIP	0.01	0.01	0.02	0.01	0.01	0.02	1
com.ice.tar	0.17	0.04	0.21	0.18	0.05	0.23	1.09
Jess-Rete	0.05	0.02	0.07	0.05	0.02	0.07	1

그리고 표 5는 분석에 소요되는 시간을 비교하였다. 분석에 사용된 컴퓨터의 사양은 Sun UltraSparc Enterprise 450 모델로, CPU는 480 MHz Ultrasparc-II 이고 RAM은 512 메가이다. 그리고 시간을 측정하기 위해 gcc의 time 라이브러리를 사용하였다. 표 5에서 집합-관계식을 생성하는 단계인 패스-1의 시간이 유사한데, 그 이유는 생성되는 집합관계식의 수가 같기 때문이다. 그리고 패스-2의 시간도 거의 차이가 없다. 패스-2는 집합관계식의 고정점을 반복을 통해서 구하는 과정이다. 프로시저-내 분석에서는 두 번 반복으로 해를 구한다. 프로시저-간 분석에서는 처리되지 않는 예외가 메

소드의 호출 체인에서 n 단계 전달된다면 n+1 번 반복으로 해를 구할 수 있다. 본 실험에서 패스-2의 속도 차이가 거의 없는 이유는 일반적인 프로그램에서 처리되지 않는 예외가 메소드 호출 체인에서 전달되는 단계가 작기 때문이다.

6. 결 론

본 연구에서는 프로그래머의 선언과 무관한 프로시저-간 예외 분석을 설계하고 이를 구현하였다. 또한 두 분석 방법의 비교를 위해 프로그래머의 선언에 의존하는 자바 컴파일러 방식의 예외 분석도 설계 구현하였으며 실험을 통해서 실제 프로그램에 적용시켜 봄으로써 두 분석 방법을 비교하였다. 프로시져-간 예외 분석은 선언을 이용하지 않고 예외를 분석함으로써 프로시져-내 예외 분석보다 정확한 예외 정보를 줄 수 있을 뿐만 아니라 프로그래머가 선언한 예외 명세를 보다 정확히 검증할 수 있다.

참 고 문 헌

[1] B.-M. Chang, K. Yi, and J. Jo, "Constraint-based analysis for Java," SSGRR2000 Computer and e-business Conference, August 2000, L'Aquila, Italy.

[2] B.-M. Chang, J. Jo, K. Yi, and K. Choe, "Interproceduaral Exception Analysis for Java," In Proceedings of 2001 ACM Symposium on Applied Computing, pages 620-625, Mar. 2001.

[3] K. Yi and B.-M. Chang, "Exception analysis for Java," 'ECOOP'99 Workshop on Formal Techniques for Java Programs, June 1999.

[4] K. Arnold and J. Gosling, "The Java Programming Languages, Second Edition," Addison-Wesley, 1997

[5] N. Heintze, "Set-based program analysis," Ph.D thesis, Carnegie Mellon University, Oct, 1992.

[6] K. Yi and S. Ryu, "Towards a cost-effective estimation of uncaught exceptions in SML programs," In Lecture Note in Computer Science, volume 1302, pages 98-113, Springer-Verlag, Proceedings of 4th Static Analysis Symposium, Sep, 1997.

[7] D. Bacon and P. Sweeney, "Fast Static Analysis of C++ Virtual Function Calls," In Proceedings of 11th Anual Conference on Objected-Oriented Programming Systems, Languages, and Applications, pages 324-341, Jan 1994.

[8] J. Dean, D. Grove, and C. Chambers, "Optimization of objected-oriented program using static class hierarchy," In Proceedings of 9th European Conference on Objected-Oriented Programming, pages 77-101, Aug. 1995

[9] G. Defouw, D. Grove, and C. Chambers, "Fast interprocedural class analysis," in Proceedings of 25th ACM SIGPLAN Symposium on Principles of Programming Languages, pages 222-236, Jan. 1998.

[10] F. Tip and J. Parlsberg, "Scalable Propagation-Based Call Graph Construction Algorithms," In Proceedings of 15th Annual Conference on Objected-Oriented Programming Systems, Languages, and Applications, pages 281-293, Oct. 2000.

[11] http://www.jars.com

[12] http://www.gamelan.com

조 장 우

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

창 병 모

1988년 서울대학교 컴퓨터공학과 졸업 (학사). 1990년 한국과학기술원 전산학과에서 공학석사 학위 취득. 1994년 한국과학기술원 전산학과에서 공학박사 학위 취득. 1994년 ~ 1995년 한국전자통신연구소 박사후 연수 연구원. 1995년 ~ 현재 숙명여자대학교 전산학과 조교수. 관심분야는 컴파일러 구성론(정적 분석, 코드 최적화), 논리 프로그래밍, 연역 데이타베이스