18-1-R02

# Java 언어의 타입 분석 및 응용에 관한 연구: 바이트코드 검증*

(A Study on Type Analyses and Applications for Java: Bytecode Verification)

창 병 모
숙명여자대학교 정보과학부
chang@sookmyung.ac.kr

요 약

본 논문에서는 Java 언어의 실행 환경인 자바 가상 기계 (Java virtual machine) 및 바이트코드(Bytecode)의 개요를 소개하고 지금까지 알려진 바이트코드 검증(Bytecode Verification) 기술에 대해서 타입 분석 기법이 적용된 예를 중심으로 설명한다.

## 1. 서론

최근 인터넷을 비롯한 여러 분야에서 널리 사용되고 있는 Java 언어의 큰 특징 중 하나로 플랫폼 독립성을 들 수 있다. Java 언어는 플랫폼 독립성을 유지하기 위해 기계어 코드로 번역하지 않고 Java 가상 기계(JVM) 코드인 바이트코드로 번역하여 실행한다.

JVM은 Java 프로그램과 하위 플랫폼 사이의 소프트웨어 계층으로 JVM에 대한 실행 환경은 [9]에 명세되어 있다. 지금까지 JVM은 대부분 소프트웨어로 구현되어 왔다. 바이트코드 명령어를 실행하기 위해서는 JVM 내에서 몇 단계가 필요하다. 그 중 하나가 바이트코드 검증으로 애플릿(applet)처럼 네트워크로부터 신뢰할 수 없는 코드를 내려받아 자동 실행하는 구조에서 핵심적인 보안 구성요소라고 할 수 있다. 이러한 상황은 애플릿뿐만 아니라 미들릿(midlet) 같은 모바일 코드, 자바 스마트카드 등에서도 역시 중요한 문제이다.

자바 보안의 기본 아이디어는 애플릿 같은 프로그램을 하드웨어에서 직접 수행하지 않고 모래상자(sandbox)라고 불리는 소프트웨어 층에서 실행한다는 점이다. 이를 통해서 다음과 같은 보안상의 문제점을 부분적으로 해결하고자 한다.

- 무결성(integrity): 중요한 데이터의 손상
- 비밀성(confidentiality): 민감한 정보의 누출
- 외부 공격: 트로이 목마 공격, 바이러스 같은 동작

본 논문에서는 Java 가상 기계 및 바이트코드(Bytecode)의 개요를 살펴보고 지금까지 알려진 바이트코드 검증(bytecode verification) 기술과 문제점을 정형 기법이

* 본 연구는 2003년 숙명여자대학교 교내연구비 지원에 의해 수행되었음.

적용된 예를 중심으로 소개한다.

## 2. 자바 가상 기계

자바 가상 기계는 Java 언어 실행 환경으로 기본적으로 스택 기계(stack machine)로 설계되었으며 다중 쓰레드 지원을 위해 실행 시각 쓰레드마다 하나의 스택이 할당된다. 하나의 스택에는 각 메쏘드마다 스택 프레임(stack frame)과 오퍼랜드 스택(operand stack)이 만들어 진다. 스택 프레임은 메쏘드 호출을 구현하기 위한 자료구조로 레지스터(register)라고 불리는 지역 변수(local variables)와 매개변수, 임시 기억장소, 반환 주소 등을 위한 기억 공간이다. 오퍼랜드 스택은 계산을 위한 스택으로 바이트코드 연산을 위한 오퍼랜드을 위한 기억공간이다.

자바 가상 기계는 그림 1과 같이 다음과 같은 서브시스템들로 구성된다.

- 클래스 적재기: 처음 사용되는 클래스를 동적으로 적재하고 연결한다.
- 메모리 관리자: 프로그램 요청 혹은 JVM 실행에 따라 필요 메모리를 할당하고 미사용 객체 즉 쓰레기를 자동 회수한다.
- 예외 및 쓰레드 관리자: 예외가 발생하면 예외 관리자는 예외 테이블을 검색하여 해당 처리기로 제어를 이전한다.
- 바이트코드 실행 엔진: JVM의 핵심부로 명령어의 꺼내오기(fetch), 해독(decode), 실행(execute) 등을 수행하며 다른 부분의 동작을 촉발시킨다.

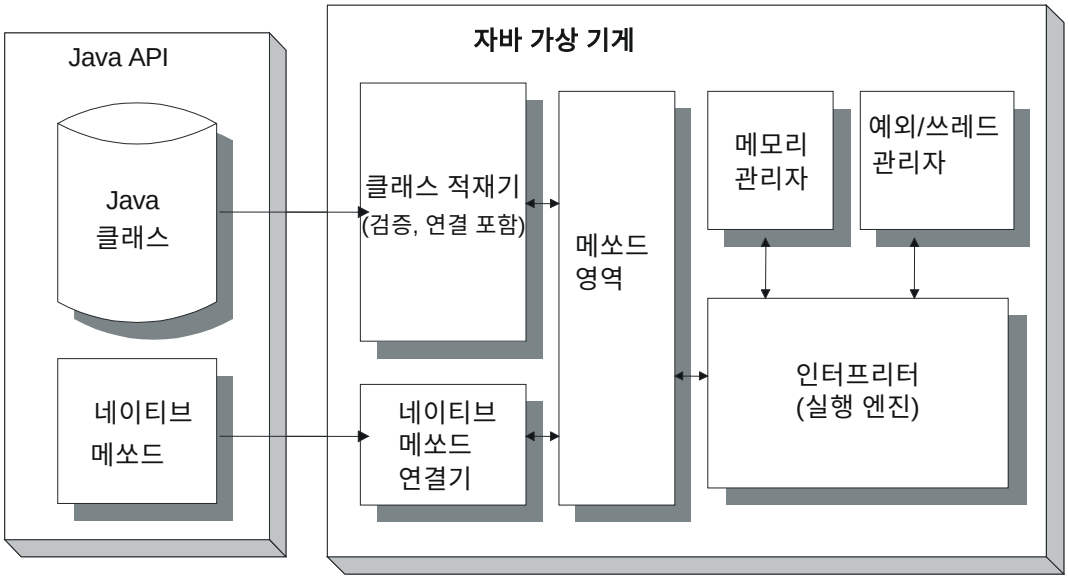


그림 1. 자바 가상 기계 구조

이외에도 자바 가상 기계의 특징을 들면 타입이 있는 32-비트 값을 다루고 객체-지향을 지원하고 심볼 참조(symbolic reference)를 이용하여 실행 시간 바인딩을 한다는 점이다.

클래스 로더 서브시스템(class loader subsystem)의 기본 아이디어는 동적 요구에 따른 클래스 적재(dynamic, on-demand class loading)라고 할 수 있으며 클래스 로더의 역할은 크게 3 가지로 다음과 같다.

- 적재(loading): 클래스가 처음 참조될 때 시스템에 적재되며 클래스 파일을 메모리에 바이트 스트림으로 읽는다.
- 연결(linking): 적재된 클래스 내의 심볼 참조를 링크한다. 플랫폼 독립을 위해 주소를 사용하지 않고 심볼참조만 사용한다.
- 초기화(initialization): 정적 변수 등을 초기화한다.

이 중 연결에서는 다음 순으로 참조된 심볼들을 연결한다.

- 검증(verification): 적재된 클래스의 구조적 정확성을 검사한다.
- 준비(preparation): 클래스의 정적 필드를 생성하고 초기화한다.
- 해결(resolution): 심볼 참조의 유효성을 검사하고 이들을 직접 참조(direct reference)로 대치한다.

이 중 본 논문에서는 주로 클래스 파일 검증에 대해서 논한다. 클래스 파일 검증이 필요한 이유는 네트워크를 통해 이동되는 신뢰할 수 없는 클래스로부터 시스템을 보호하기 위해서 이며 클래스 파일의 무결성 검증을 목표로 한다. 이를 위해 (1) 클래스 파일이 JVM 명세의 제약 조건을 만족하는지 검사하고 (2) 스택과 변수 등의 비정상적 사용을 실행 전에 실행 과정 시뮬레이션을 통해 검증한다.

JVM의 클래스 검증은 크게 4 단계로 구성되어 있으며 각 단계의 역할을 요약하면 다음과 같다.

- 1 단계: 클래스 파일 포맷을 만족하는지 구문을 검사한다.
- 2 단계: 바이트코드를 보지 않고 클래스 파일이 의미 있는지 검사한다. 예를 들면 모든 클래스는 부모 클래스가 있어야 하며 최종 클래스(final class)는 상속될 수 없다.
- 3 단계(바이트코드 검증): 타입-수준 자료흐름 분석으로 바이트코드가 적절히 수행 되는지 검사한다. 예를 들면 변수의 올바른 사용, 메쏘드의 올바른 호출, 바이트코드의 올바른 오퍼랜드 사용, 오퍼랜드 스택의 오버플로우/ 언더플로우 등을 검사한다.
- 4 단계: 명령어에 의한 참조(reference)가 유효한지 검사한다.

## 3. 바이트코드

바이트코드는 Java 가상 기계의 코드로 Java 프로그램은 컴파일러에 의해 바이트코드 프로그램으로 컴파일된다[9]. 바이트코드에서 다루는 자료 타입은 크게 기초 타입(primitive type)과 참조 타입(reference type)으로 나눌 수 있다. 기초 타입을 요약정리하면 다음과 같다.

- 정수 관련 타입: byte(8 비트), short(16 비트), int(32 비트), long(64 비트), char(16 비트, UNICODE)
- 부동소수점 관련 타입 : float(32 비트), double(64 비트)
- boolean : true 혹은 false
- returnAddress : 바이트코드 명령어 주소

JVM의 참조 타입(reference type)은 다음과 있다.

- 클래스(class)
- 배열(array)
- 인터페이스(interface)

바이트코드 8-비트 연산코드를 갖는 202개 명령어로 구성되어 있으며 스택을 기반으로 실행된다. 즉 실행을 위해 오퍼랜드 스택을

이용하고 명령어 실행 전 오퍼랜드들을 스택에 적재하고 결과 값도 스택에 적재한다. 레지스터는 없고 대신에 지역변수를 레지스터처럼 사용한다. 대부분의 바이트코드 명령어는 오퍼랜드 타입을 명시한다. 예를 들면 iadd 명령어는 정수 덧셈을 의미한다. 비슷하게 다른 명령어 isub, iload, istore 등에도 타입을 명시한다. 이러한 바이트코드 명령어의 타입 힌트는 실행 전 타입 검사를 가능하게 하며 이로 인해 보다 안전한 시스템이 가능하다. 복잡한 바이트코드 명령어로는 메모리 할당, 모니터/ 쓰레드 동기화, 메쏘드 호출 등을 위한 명령어를 들 수 있다.

예를 들어 다음 메쏘드를 위한 바이트코드를 살펴보자

```
static int factorial(int n) {
   int res
   for (res = 1; n > 0; n--)
            res = res * n;
   return res
}
```

```
0: iconst_1 // 1 적재
1: istore_1 // 레지스터 1(변수 res)에 저장
2: iload_0  // 레지스터 0(매개변수 n) 적재
3: ifle 14  // 음수 혹은 0이면 PC 14로 간다.
6: iload_1  // 레지스터 1 (변수 res) 적재
7: iload_0  // 레지스터 0 (변수 n) 적재
8: imul     // 스택 꼭대기의 두 정수 곱
9: istore_1 // 결과 값을 팝하여 레지스터 1
            // (변수 res)에 저장
10: iinc 0, 01 // 레지스터 0(변수 n) 1 감소
11: goto 2   // PC 2로 간다.
14: iload_1  // 레지스터 1 (변수 res) 적재
15: ireturn  // 값을 호출자에 반환
```

메쏘드 호출을 위한 명령어는 다음과 같이 4 개가 있다.

- invokevirtual : 객체의 가상 메쏘드 (virtual method) 호출에 사용된다.
- invokeinterface: 참조 타입이 인터페이스일 때 객체의 가상 메쏘드 호출에 사용된다.
- invokespecial: 실체 초기화 메쏘드, 전용 메쏘드, 수퍼클래스 메쏘드 호출 등에 사용된다.
- invokestatic: 클래스의 정적 메쏘드 (static method) 호출에 사용된다.

가상 메쏘드 호출 시 대상 객체(target object)는 0-번째 매개변수로 전달된다.

x.m(...); ⇨ m(x, ... );

또한 메쏘드 선언은 구현 시 this가 0-번째 형식 매개변수 이름으로 사용된다.

m(...) {...} ⇨ m(this,... ) {...}

따라서 가상 메쏘드 m을 호출하려면 먼저 호출 대상 객체(target object)를 스택에 적재하고 실 매개변수 값들을 스택에 적재한 후에 해당 메쏘드를 호출한다.

예를 들어 다음 메쏘드와 바이트코드를 살펴보자.

```
int add12and13() {
    return addTwo(12, 13);
}
```

```
0: aload_0    // 지역변수 0(this) 적재
1: bipush 12  // 정수 12 적재
3: bipush 13  // 정수 13 적재
5: invokevirtual #4 // addtwo() 호출
8: ireturn // addTwo()의 결과인 오퍼랜드
           //  스택 꼭대기의 정수 값 반환
```

예외의 처리 과정을 살펴보자. 예외 관리자는 예외가 발생하면 예외 발생 지점의 문맥(context)을 저장하고 발생된 예외를 처리할 수 있는 가장 가까운 catch 절을 찾는다. 이때 발생된 예외 타입과 예외 테

이블의 catch 절에 선언된 타입을 비교하여 해당 catch 절을 찾는다. 해당 catch 절을 찾으면 예외(사실은 예외에 대한 참조)를 오퍼랜드 스택에 넣고 catch 절을 실행한다. 찾지 못하면 해당 자바 쓰레드를 종료한다.

예외를 처리하는 예를 살펴보면 다음과 같다.

```
void m() {
    try {
        doSomething();
    } catch (E e) {
        handleExc(e);
    }
}
```

```
Method void m()
0 aload_0  // try 블록의 시작
1 invokevirtual #6
          // 메쏘드doSomething() 호출
4 return   // try 블록 끝; 정상적 반환
5 astore_1
        // 던져진 값을 지역변수 1에 저장
6 aload_0  // this를 적재
7 aload_1  // 던져진 값 적재
8 invokevirtual #5
           // 메쏘드 handleExc() 호출
11 return  // E 처리 후에 반환
Exception table:
From To Target Type
 0      4      5      Class E
```

## 4. 바이트코드 검증

바이트코드 검증은 애플릿이나 미들릿처럼 신뢰할 수 없는 코드를 내려받는 구조에서 실행 전에 코드의 안전성을 검증하기 위한 것으로 JVM의 핵심적인 보안 구성요소이다[8,9]. 그 기본 아이디어는 애플릿을 하드웨어에서 직접 수행하지 않고 모래상자(sandbox)라 불리는 소프트웨어 층에서 실행한다는 것이다.

모래상자 모델의 핵심 내용은 접근 제어와 타입 안전성을 보장하는 가상 기계라고 할 수 있다. 접근 제어를 위해서는 접근 제어를 제공하는 안전한 API만 사용할 수 있으며 파일과 네트워크 접근을 제한한다. 타입 안전성을 보장하는 가상 기계는 바이트코드 검증을 통해서 데이터 무결성, 코드 무결성, 가시성 조정자의 올바른 사용 등을 검증한다.

타입 안전성을 보장하는 방어적 가상 기계(defensive virtual machine)는 바이트코드를 실행하면서 동적 타입 검사를 한다. 따라서 실행 속도가 상당히 떨어진다.

따라서 현재 대부분의 JVM에서는 적재 시 바이트코드 검증 특히 정적 데이터 흐름 분석(static dataflow analysis)을 통하여 타입 안정성을 확보하고 방어적 가상 기계 사용으로 인한 실행 속도 저하를 방지한다.

바이트코드 검증기의 역할은 그림 2.와 같다.

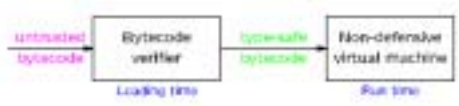


그림 2. 바이트코드 검증기 역할

바이트코드 검증을 통해서 검증하는 내용을 정리하면 다음과 같다.

(1) 코드의 형식: 예를 들어 명령어 중간으로 점프할 수 없다.

(2) 올바른 타입의 오퍼랜드에 명령어 적용: 예를 들어 getfield C.f 는 클래스 C 혹은 그 서브클래스의 객체에 대한 참조만을 받는다

(3) 스택 오버플로우/언더플로우: 연산 중에 스택 오버플로우 혹은 언더플로우 검사.

(4) 지역변수( 레지스터)의 초기화: 예를 들어 초기화되지 않는 레지스터를 사용할 수 없다.

(5) 객체의 초기화 : new C()의 경우 C의 구성자를 호출한 후에 사용해야 한다.

(6) 가시성 조정자: 예를 들어 클래스 밖에서 전용 멤버를 접근할 수 없다.

바이트코드 검증의 기본아이디어는 방어적 가상 기계를 타입-수준에서 요약 해석(abstract interpretation)하는 것이다[8]. 이 경우 요약 도메인(abstract domain)은 오퍼랜드 스택과 레지스터(지역변수) 집합을 타입으로 요약한 것이며 요약 연산자(abstract operator)는 각 바이트코드 명령어에 대해서 오퍼랜드 타입을 검사하고 결과 타입을 계산하는 것이다.

예제 바이트코드 프로그램에 대한 타입-수준의 요약해석의 예가 그림3.에 있다.

```
class C {
        int x;
        void move(int delta) {
          int oldx = x;
          x += delta;
          D.draw(oldx,x);
        }
}
```

여기서 this는 r0로 delta는 r1로 oldx는 r2로 표시된다.

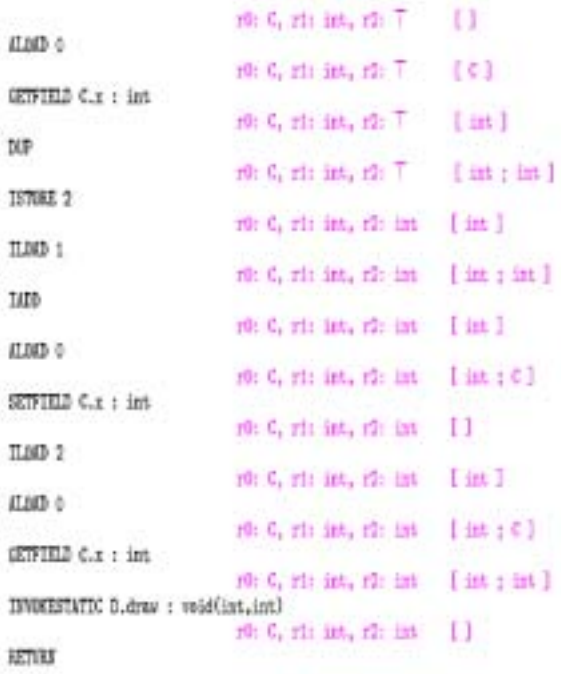


그림 3. 요약 해석의 예

이 요약 해석에서 그림 4.의 예처럼 분기는 일반적인 자료 흐름 분석에서처럼 처리한다. 즉 분기 지점에서 타입 정보는 모든 후행자들(successors)에게 전달한다. 조인 지점에서는 모든 선행자들(predecessors)의 타입들의 lub를 취한다. 반복의 경우는 타입이 더 이상 변하지 않을 때까지 반복한다.

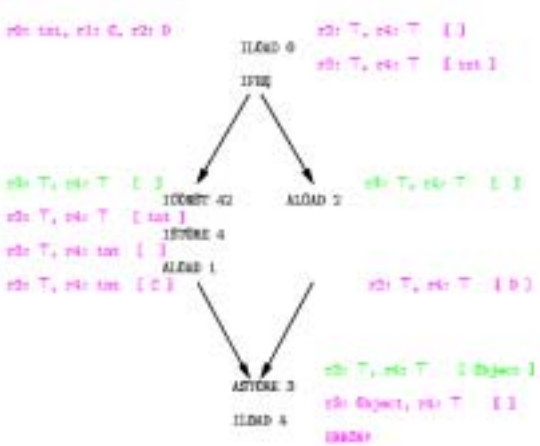

그림 4. 분기 및 조인의 자료 흐름 분석

타입-수준 요약해석을 위한 각 명령어의 전이 관계(transition relation)는 다음과 같이 정의될 수 있다.

$$instr : (\tau_{reg}, \tau_{stack}) \rightarrow (\tau'_{reg}, \tau'_{stack})$$

e.g. $\texttt{iadd} : (r,\ \texttt{int.int}.s) \rightarrow (r,\ \texttt{int}.s)$

각 바이트코드 명령어에 대한 전이 관계는 그림 6.에 있다.

이 전이 관계를 이용하여 바이트코드 프로그램을 위한 데이터 흐름 분석을 위한 데이터 흐름 등식(dataflow equation)을 일반적으로 다음과 같이 설정할 수 있다.

$$i : in(i) \rightarrow out(i)$$
$$in(i) = lub\{out(j) \mid j \text{ predecessor of } i\}$$
$$in(i_{start}) = ((P_0, \ldots, P_{n-1}, \top, \ldots, \top), \varepsilon)$$

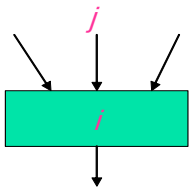


그림 5. 명령어 i의 선행자와 후행자

이러한 데이터 흐름 등식이 설정된 후에 표준 고정점 반복(standard fixpoint iteration)으로 해를 계산할 수 있다. 이 해의 안전성(soundness)은 [8]에서 증명되었다.

바이트코드 검증과 관련해서 몇 가지 문제점들이 있다.

(1) 인터페이스(interface): 인터페이스를 포함하는 경우 서브타입 관계가 세미-래티스가 아니다. 이 문제에 대한 해결책으로 Sun에서는 실행 시간 타입 검사를 사용한다 [9]. 다른 방법으로는 Dedekind completion을 사용하여 새로운 타입을 추가함으로 이 문제점을 극복할 수도 있다[6].

$\texttt{iconst}\ n\ :\ (S, R) \rightarrow (\texttt{int}.S, R)$ if $|S| < M_{stack}$
$\texttt{ineg}\ :\ (\texttt{int}.S, R) \rightarrow (\texttt{int}.S, R)$
$\texttt{iadd}\ :\ (\texttt{int.int}.S, R) \rightarrow (\texttt{int}.S, R)$
$\texttt{iload}\ n\ :\ (S, R) \rightarrow (\texttt{int}.S, R)$
  if $0 \le n < M_{reg}$ and $R(n) = \texttt{int}$ and $|S| < M_{stack}$
$\texttt{istore}\ n\ :\ (\texttt{int}.S, R) \rightarrow (S, R\{n \leftarrow \texttt{int}\})$ if $0 \le n < M_{reg}$
$\texttt{aconst\_null}\ :\ (S, R) \rightarrow (\texttt{null}.S, R)$ if $|S| < M_{stack}$
$\texttt{aload}\ n\ :\ (S, R) \rightarrow (R(n).S, R)$
  if $0 \le n < M_{reg}$ and $R(n) <: \texttt{Object}$ and $|S| < M_{stack}$
$\texttt{astore}\ n\ :\ (\tau.S, R) \rightarrow (S, R\{n \leftarrow \tau\})$
  if $0 \le n < M_{reg}$ and $\tau <: \texttt{Object}$
$\texttt{getfield}\ C.f.\tau\ :\ (\tau'.S, R) \rightarrow (\tau.S, R)$ if $\tau' <: C$
$\texttt{putfield}\ C.f.\tau\ :\ (\tau_1.\tau_2.S, R) \rightarrow (S, R)$ if $\tau_1 <: \tau$ and $\tau_2 <: C$
$\texttt{invokestatic}\ C.m.\sigma\ :\ (\tau'_n \ldots \tau'_1.S, R) \rightarrow (\tau.S, R)$
  if $\sigma = \tau(\tau_1, \ldots, \tau_n), \tau'_i <: \tau_i$ for $i = 1 \ldots n$, and $|\tau.S| \le M_{stack}$
$\texttt{invokevirtual}\ C.m.\sigma\ :\ (\tau'_n \ldots \tau'_1.\tau'.S, R) \rightarrow (\tau.S, R)$
  if $\sigma = \tau(\tau_1, \ldots, \tau_n), \tau' <: C, \tau'_i <: \tau_i$ for $i = 1 \ldots n$, $|\tau.S| \le M_{stack}$

그림 6. 바이트코드 명령어의 전이관계

(2) 객체 초기화(object initialization): 이것을 검사하기 위해서는 must-alias 분석이 필요하다. 이는 초기화되지 않은 객체에 대한 두 개의 참조가 스택에 있는 경우가 가능하기 때문이다 [4,8].

(3) 서브루틴(subroutine): 서브루틴에 대한 분석에서 여러 문제점이 발생할 수 있다 [3,5,12].

여기에서는 바이트코드 검증을 위한 데이터 흐름 분석에서 서브루틴 관련 문제점들과 그 해결책을 소개한다.

서브루틴은 try-finally를 컴파일할 때 코드 중복을 피하기 위해 사용된다. 예를 들어 다음과 같은 문장을 컴파일하는 경우에

```
try {
        ...
        if (cond) { return e; }
} finally {
        // finalization code
```

```
}
```

서브루틴을 사용하지 않는 경우와 서브루틴을 사용하는 경우에 컴파일된 코드는 각각 그림 7.과 그림 8.에 있다.

```
...
iload cond
ifne Early_return
...
finalization code
...
Early_return:
compute e
istore 2
finalization code
iload 2
ireturn
Exception_handler;
astore 2
finalization code
aload 2
athrow
```

그림 7. 서브루틴을 사용하지 않는 경우

```
...
iload cond
ifne Early_return
...
jsr Subroutine
...
Early_return:
compute e
istore 2
jsr Subroutine
iload 2
ireturn
Exception_handler;
astore 2
jsr Subroutine
aload 2
athrow
Subroutine
astore 3
finalization code
ret 3
```

그림 8. 서브루틴을 사용하는 경우

바이트코드 검증을 위한 데이터 흐름 분석에서 jsr과 ret를 분기처럼 취급하여 분석할 수 있다[3,5,12]. 그러나 이 경우 문제점은 지나친 타입 합병이 일어난다는 점이다. 이러한 상황은 그림 9.에 잘 나타나 있다.

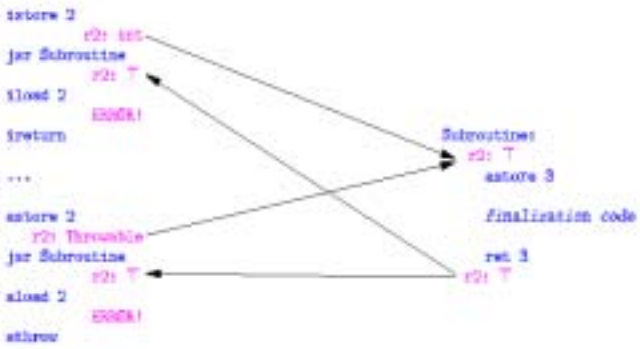


그림 9. 서브루틴 분석에서 타입 합병

이 문제에 대해서는 기금까지 3종류의 해결책이 제시되었다.

첫 번째 방법은 Sun에서 제시한 방법으로 서브루틴 내에서 사용되지 않는 레지스터들은 jsr 후에도 자기 타입을 유지하게 하는 것이다[9].

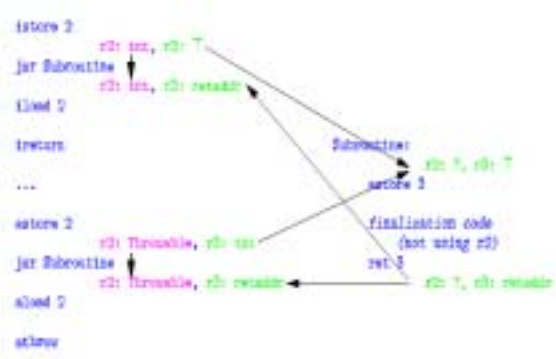

그림 10. Sun의 방법 예

이 방법을 적용하기 위해서는 먼저 jsr과 대응되는 ret 찾아야 하고 각 서브루틴이 사용하는 레지스터 결정해야 한다. 바이트코드 프로그램은 서브루틴 구조가 구문적으로 정해져 있지 않으므로 이를 위해 서브루틴 구조 분석이 필요하다. [12]에서는 예외를 고려하지 않은 상태에서 이러한 연구를 하였으며 [13]의 연구는 Sun의 구현과 거의 비슷하다.

이 문제에 대한 두 번째 해결책은 문맥-민감 분석(context-sensitive analysis)으로 서브루틴를 각 호출 사이트에 대해서 한 번씩 분석하는 것이다. 여기서 호출 문맥(calling context)은 각 호출 지점에서의 상태 즉 (스택 타입, 레지스터 타입)이 될 것이다. 그림 11.은 예제 프로그램에 대한 문맥-민감 분석의 예를 보여주고 있다.

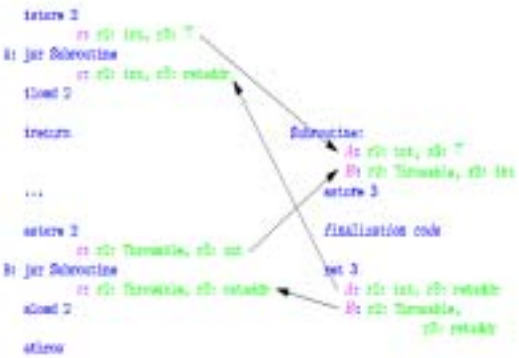

그림 11. 문맥-민감 분석 예

이 문제에 대한 세 번째 해결책은 모델 검사(model checking) 기술을 이용하여 타입-수준 VM를 다음과 같이 전이 관계로 모델링하고 모든 가능한 상태들을 검사한다[1].

$$(PC_{instr}, r_{regs}, r_{stack}) \rightarrow (PC_{succ}, r'_{regs}, r'_{stack})$$

이 경우 타입 합병을 전혀 일어나지 않으며 계산 비용은 많지만 다른 분석들의 안전성 증명에 유용하다. 그림 12.는 모델 검사를 이용한 간단한 예를 보여주고 있다.

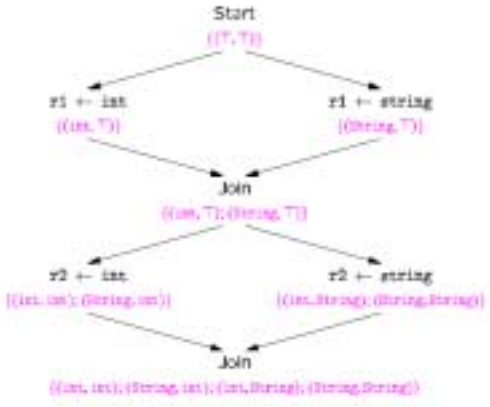


그림 12. 모델 검사 예

## 5. 경량 바이트코드 검증

Java 카드는 애플릿의 다운로딩을 지원하고 엄격한 보안이 필요하다[2]. 자바 카드는 2 KB RAM, 16 KB EEPROM, 8 비트 프로세서 등과 같은 제한된 자원만을 제공한다.

자바 카드 상에서 바이트코드 검증은 검증 시간 면에서 고정점 계산이 복잡하고 필요 기억 공간 역시 자바 카드에게는 너무 크다[8].

이러한 문제점을 해결하기위한 방법들이 제안되어 왔는데 그 중 대표적인 것이 [11] 알고리즘이다. 이 방법은 일종의 증명 휴대 코드(Proof Carrying Code)[11]로 코드와 함께 미리 계산된 각 분기 지점의 스택과 레지스터 타입을 전달하며 검증기는 이 정보를 이용하여 단지 검사만 한다.

이 방법은 고정점 계산은 피할 수 있고 한 패스로 계산이 가능하며 증명 정보는 읽

기만 하면 되므로 EEPROM에 저장 가능하다는 장점이 있다. 반면 서브루틴 처리 방법이 복잡하고 증명 정보의 크기가 상당히 크다(코드의 20 ~ 100%).

## 5. 결론 및 향후 연구

본 논문에서는 타입 분석 기법을 이용한 바이트코드 검증에 대해서 소개하였다. 이 문제는 정형 기법이 스마트 카드와 같은 산업체의 실용적 분야에 적용된 대표적인 예로 작고 잘 정의된 문제이며 이를 타입 분석 기법을 적용하여 해결한 대표적인 예라고 할 수 있다[2,7].

앞으로 바이트코드 검증에 관한 연구 결과를 실제 기계어 혹은 어셈블리 코드를 검증하는데 적용할 수 있을 것이다. 또한 타입 분석 대신에 보다 간단한 타입 추론을 이용한 타입 어셈블리어(Typed Assembly Language)에 대한 연구가 활발히 진행 중이다[4,12]. 또한 자원 사용 ( 메모리, 시간, 반응성) 등과 같은 타입 안전성 이상의 성질들을 검증하는 연구가 필요할 것이다.

## 참고문헌


[1] D. Basin, S. Friedrich and M. Gawkowski. Bytecode verification by model checking. Journal of Automated Reasoning. Special issume on bytecode verification.

[2] Z. Chen. Java Card Technology for Smart Cards: Architecture and Programmers's Guide. Addison-Wesley, 2000

[3] A. Coglio. Simple verification technique for complex Java bytecode subroutines. In 4th ECOOP Workshop on Formal Techniques for Java-like Programs, 2002

[4] S. N. Freund and J. C. Mitchell. A formal framework for the Java bytecode language and verifier. In ACM OOPSLA 1999, pages 147-166, 1999.

[5] M. Hagiya and A. Tozawa. on a new method for dataflow analysis of Java virtual machine subroutines. SAS'98, volume 1503 of LNCS, pages 17-32. Springer-Verlag, 1998.

[6] T. Knoblock and J. Rehof. Type elaboration and sybtype completion for Java bytecode; In 27th Symposium on Principles and Programming Languages, pages 228-242. ACM Press, 2000.

[7] X. Leroy. Bytecode verification for java smart card. Software Practice and Experience, 32:319-340, 2002.

[8] X. Leroy. Java bytecode verification: algorithms and formalizations, Journal of Automated Reasoning, Special issume on bytecode verification.

[9] Lindholm and F. Yellin. The Java Virtual Machine Specification. Addison-Wesley, 1999. Second edition.

[10] G. C. Necula. Proof-carrying code. In 24th Symposium on Principles of Programming Languages, pages 106-119. ACM Press, 1997.

[11] E. Rose and K. Rose. Lightweight bytecode verification. In OOPSLA Workshop on Formal Underpinnings of Java, 1998.

[12] R. Stata and M. Abadi. A type system for java bytecode subroutines. ACM Transactions on Programming Languages and Systems, 21(1):90-137, 1999.

[13] Z. Qian. Standard fixpoint iteration for Java bytecode verification. ACM Transactions on Programming Languages and Systems, 22(4):638-672, 2000.



창 병 모

1988년 서울대학교 컴퓨터공학과(학사)

1990년 한국과학기술원 전산학과(공학석사)

1994년 한국과학기술원 전산학과(공학박사)

1995년부터 현재까지 숙명여자대학교 컴퓨터과학과 교수

관심분야 : 프로그램 분석 및 검증, 자바 및 모바일 컴퓨팅, 자동 소프트웨어 공학