



자바 가상기계 및 바이트코드 검증 (JVM and Bytecode Verification)

프로그래밍언어연구회 LiComR 겨울학교
2004년 2월 12일

숙명여대 컴퓨터과학과
창 병모


1



목차

- 자바 가상 기계 개요(Java virtual machine)
- 바이트코드(Bytecode)
- 자바 스택 및 메쏘드 호출
- 바이트코드 검증(Bytecode Verification)

2



자바 가상 기계 개요

3



자바 가상 기계

- 바이트코드 실행 환경
 - JVM에 대한 명세 [Sun95]
- 지금까지는 소프트웨어적 접근이 일반적임
 - 자바 프로그램과 하위 플랫폼 사이의 소프트웨어 계층
 - 바이트코드 명령어 실행을 위해 JVM의 몇 단계가 필요함

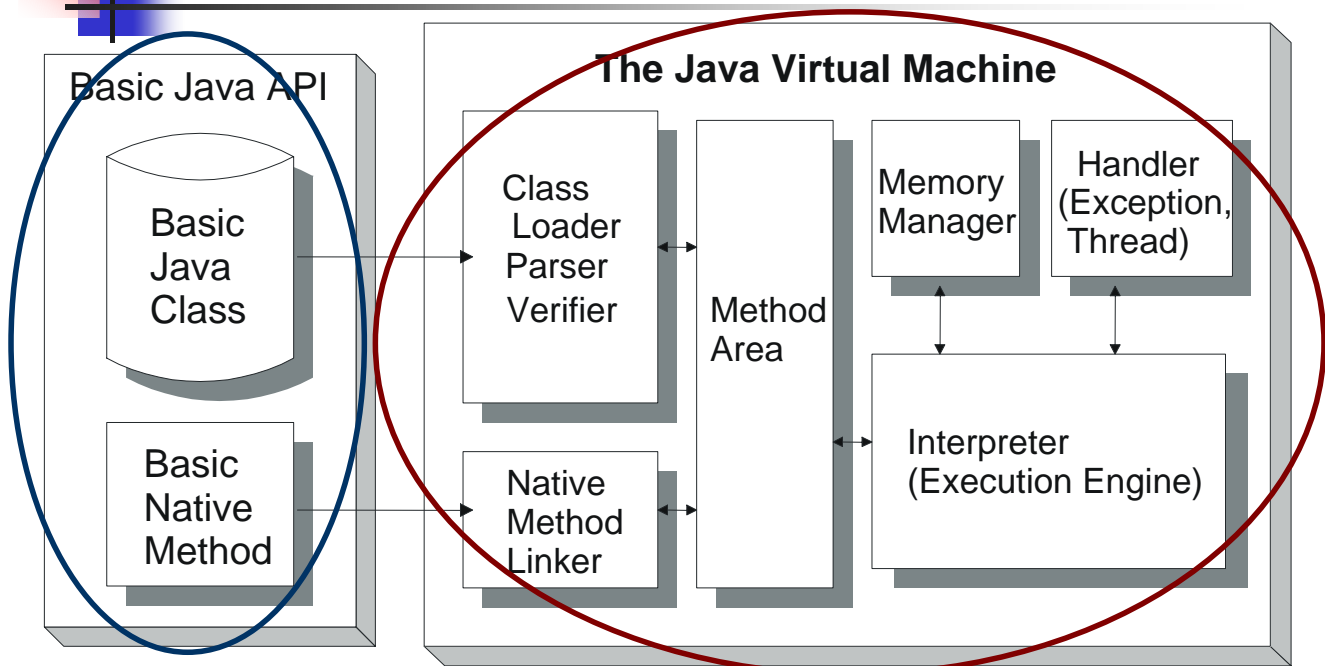
4

JVM 개요

- 스택 기계(Stack machine)
 - 스택 프레임
 - 지역 변수(Local variables) 혹은 레지스터(register)
 - 임시 기억장소, 반환 주소
 - 오퍼랜드 스택(Operand stack)
 - 계산을 위한 스택
- 타입이 있는 32-비트 값
- 다중 스레드 기계(Multi-threading machine)
- 객체-지향 지원
- 심볼 참조(Symbolic reference)
 - 실행 시간 바인딩

5

JVM 구조



6



JVM 시작

- **“java App hello everyone”**
 - App 클래스의 *main()* 메소드가 시작점
 - 나머지 인자들은 *main()*의 String 매개변수에 전달된다.
- 시작 과정
 - 실행 환경을 설정하고 초기화
 - 시스템으로부터 힙을 할당
 - “Object”, “Class”, “String”, “Thread” 등의 클래스 적재
 - 프로그램 시작 클래스 적재
 - *main()* 메소드 호출

7



클래스 로더(Class Loader)

- 기본 아이디어
 - 동적 클래스 적재(Dynamic, on-demand loading)
- 역할
 - 적재(loading)
 - 클래스가 처음 참조될 때 시스템에 적재된다
 - 클래스 파일을 메모리에 바이트 스트림으로 읽는다.
 - 연결(linking)
 - 적재된 클래스 내의 심볼 참조를 링크한다.
 - 플랫폼 독립을 위해 주소를 사용하지 않고 심볼 참조만 사용
 - 초기화(initialization)
 - 정적 변수 초기화

8



연결(Linking)

- 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 대체한다.

9



클래스 검증기(Class Verifier)

- 왜 클래스 검증이 필요한가?
 - 네트워크를 통한 신뢰할 수 없는 클래스로부터 시스템 보호
- 클래스 파일의 무결성 검증
 - 클래스 파일이 JVM 명세의 제약 조건을 만족하는지 검사
 - 스택과 변수 등의 비정상적 사용을 찾기 위해 실행 전에 실행 과정 시뮬레이션

10



4 단계 검증 과정

- 패스 1
 - 클래스 파일 포맷을 만족하는지 검사
- 패스 2
 - 바이트코드를 보지 않고 클래스 파일이 의미 있는지 검사
 - 모든 클래스는 슈퍼클래스가 있어야 한다.
 - 최종 클래스(final class)는 상속될 수 없다.
- 패스 3 (바이트코드 검증)
 - 타입-수준 자료흐름 분석으로 바이트코드의 적절한 수행 검사
 - 변수의 올바른 사용, 메소드의 올바른 호출
 - 바이트코드의 올바른 오퍼랜드 사용
 - 오퍼랜드 스택의 오버플로우/언더플로우 검사
- 패스 4 (동적 링크의 일부)
 - 명령어에 의한 참조(reference)가 유효한지 검사

11



메소드 영역(Method Area)

- 메소드 영역
 - Unix의 Code/text segment와 비슷한 역할
 - 적재된 클래스에 대한 모든 정보가 저장된다
 - 모든 쓰레드가 공유
- 실행시간 상수 풀(Constant pool)
 - 클래스의 심볼 테이블
 - 메소드 영역에 할당
 - 동적 연결의 핵심

12



메모리 관리자(Memory Manager)

- 메모리 구성
 - 메소드 영역(method area)
 - 힙(heap)
 - 스택(stack)
- 메모리 관리
 - 메모리 할당
 - 프로그램 요청(new)에 따라 힙에 객체 할당
 - JVM 실행에 따른 메모리 할당
 - 쓰레기 회수
 - 미사용 객체의 자동 회수

13



실행 시간 데이터 영역

- pc 레지스터
 - 바이트코드 명령어의 주소로
 - 쓰레드 당 하나의 pc
- 자바 스택
 - 쓰레드 당 하나의 자바 스택
 - 메소드 프레임을 위한 기억공간
- 힙
 - 객체와 배열을 위한 기억공간
 - 쓰레기 수집
 - 모든 쓰레드가 공유하는 기억 공간

14



실행 엔진(Execution Engine)

- JVM의 핵심부
 - 명령어의 꺼내오기(fetch), 해독(encode), 실행(execute)
 - 다른 부분의 동작을 촉발시킨다.
- 소프트웨어 혹은 하드웨어 구현
 - 소프트웨어
 - 빠른 해석기 구현
 - Just-in-Time 컴파일러
 - 하드웨어
 - Java 프로세서

15



예외 관리자(Exception Manager)

- 예외
 - Java의 예외 처리 메커니즘
 - 프로그래머가 예외 처리를 제어할 수 있다.
 - *try - catch - finally*
- 예외 테이블(Exception table)
 - 각 메소드는 catch 블록들을 나열하는 예외 테이블을 포함
- 예외 관리자
 - 예외가 발생하면 예외 관리자는 예외 테이블을 검색하여 해당 처리기로 제어를 이전한다.


16



네이티브 메소드 연결

- 네이티브 메소드(Native Method)
 - C나 C++ 같은 언어로 구현된 메소드
 - 하드웨어에 대한 접근
 - 성능 개선
 - e.g.) MPEG 해독
 - DLL 같은 라이브러리
- JNI (Java Native Interface)
 - Java 시스템과 네이티브 메소드 사이의 표준 인터페이스

17



바이트코드(Bytecode)

18



JVM의 데이터 타입

- 기초 타입(Primitive types)
 - 정수 관련 타입
 - **byte**(8 bits), **short**(16 bits), **int**(32 bits), **long**(64 bits), **char**(16 bits, UNICODE)
 - 부동소수점 : **float**(32 bits), **double**(64 bits)
 - **boolean** : *true* / *false*
 - **returnAddress** : 바이트코드 명령어 주소
- 참조 타입(Reference types)
 - 클래스(class)
 - 배열(array)
 - 인터페이스(interface)

19



바이트코드 명령어

- 8-비트 연산코드를 갖는 202개 명령어
- 스택 기반 실행
 - 오퍼랜드 스택을 이용
 - 명령어 실행 전 오퍼랜드들을 스택에 적재하고 결과 값도 스택에 적재
 - 레지스터는 없고 대신에 지역변수 사용
- 오퍼랜드 타입 명시
 - **iadd** : 정수 덧셈
- 복잡한 명령어들
 - 메모리 할당
 - 모니터/ 쓰레드 동기화
 - 메소드 호출

20

바이트코드 명령어 카테고리

Category	No.	Example
arithmetic operation	24	iadd, lsub, frem
logical operation	12	iand, lor, ishl
numeric conversion	15	int2short, f2l, d2l
pushing constant	20	bipush, sipush, ldc, iconst_0, fconst_1
stack manipulation	9	pop, pop2, dup, dup2
flow control instructions	28	goto, ifne, ifge, if_null, jsr, ret
managing local variables	52	astore, istore, aload, iload, aload_0
manipulating arrays	17	aastore, bastore, aaload, baload
creating objects and array	4	new, newarray, anewarray, multianewarray
object manipulation	6	getfield, putfield, getstatic, putstatic
method call and return	10	invokevirtual, invokestatic, areturn
miscellaneous	5	throw, monitorenter, breakpoint, nop

21

바이트코드 명령어 타입

- 타입 힌트
 - iadd, isub, istore,...
 - 실행 전 타입 검사 가능
 - 보다 안전한 시스템 가능
 - 타입 접두사 없는 명령어
 - pop, dup, invokevirtual, ...

type	code
int	i
long	l
float	f
double	d
byte	b
char	c
short	s
reference	a

22

예: 바이트코드

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

```
0: iconst_1      // push 1
1: istore_1     // store it in register 1 (변수 res)
2: iload_0      // push register 0 (매개변수 n)
3: ifle 14      // if negative or null, goto PC 14
6: iload_1      // push register 1 (res)
7: iload_0      // push register 0 (n)
8: imul         // multiply the two integers at top of stack
9: istore_1     // pop result and store it in register 1 (res)
10: iinc 0, 01  // decrement register 0 (n) by 1
11: goto 2      // goto PC 2
14: iload_1      // load register 1 (res)
15: ireturn     // return its value to caller
```

23

명령어 및 타입 접두사

	int	long	float	double	byte	char	short	reference
?2c	☐							
?2d	☐	☐	☐					
?2l	☐	☐	☐	☐				
?2f	☐	☐		☐				
?2i	☐	☐	☐	☐				
?2s	☐							
?add	☐	☐	☐	☐				
?aload	☐	☐	☐	☐	☐	☐	☐	☐
?and	☐	☐						
?astore	☐	☐	☐	☐	☐	☐	☐	☐
?cmp		☐						
?cmp{g l}			☐	☐				
?const <n>	☐	☐		☐				☐
?div	☐	☐	☐	☐				
?inc	☐							
?ipush					☐		☐	
?load	☐	☐	☐	☐				
?mul	☐	☐	☐	☐				
?neg	☐	☐	☐	☐				
?newarray								☐
?or	☐	☐						
?rem	☐	☐	☐	☐				☐
?return	☐	☐	☐	☐				☐
?shl	☐	☐						
?shr	☐	☐						
?store	☐	☐	☐	☐				☐
?sub	☐	☐	☐	☐				
?throw								☐
?ushr	☐	☐						
?xor	☐	☐						

24

자바 스택 및 메소드 호출 (Java Stack and Method Call)

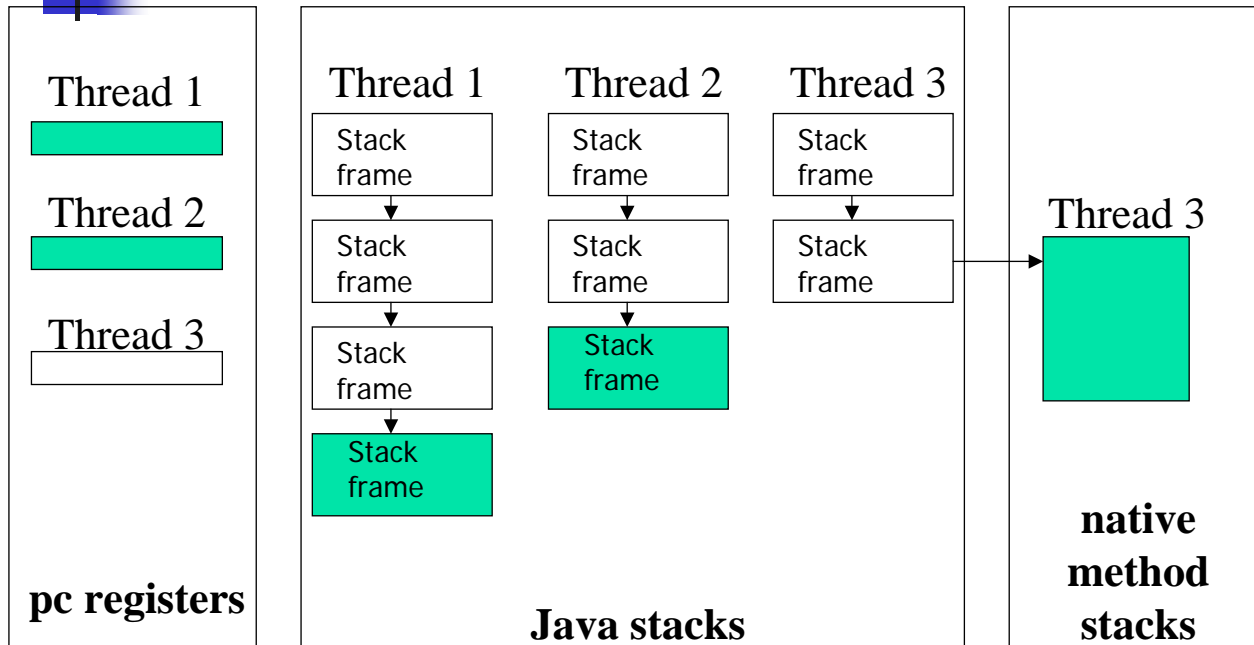
25

자바 스택 및 스택 프레임

- 쓰레드 당 하나의 자바 스택
 - 자바 스택 위에 스택 프레임들이 쌓인다.
- 메소드 호출 당 하나의 스택 프레임
 - 오퍼랜드 스택(Operand stack)
 - 지역 변수(Local variable)
 - 프레임 크기는 메소드가 컴파일될 때 결정된다.

26

쓰레드와 자바 스택



27

메소드 호출 명령어

- *invokevirtual*
 - 객체의 가상(virtual, instance) 메소드 호출
 - 목시적 매개변수 : *this*
- *invokeinterface*
 - 참조 타입이 인터페이스일 때 객체의 가상 메소드 호출
 - 목시적 매개변수 : *this*
- *invokespecial*
 - 실제 초기화 메소드, 전용 메소드, 슈퍼클래스 메소드 호출
 - 목시적 매개변수 : *this*
- *invokestatic*
 - 클래스의 정적 메소드(static method) 호출

28



메소드 호출과 this

- 메소드 호출의 구현
 - 대상 객체(target object)는 0-번째 매개변수로 전달
 - `x.m(...);` → `m(x, ...);`
- 메소드 선언의 구현
 - `this`는 0-번째 형식 매개변수 이름
 - `m(...){ ... }` → `m(this, ...){ ... }`

29

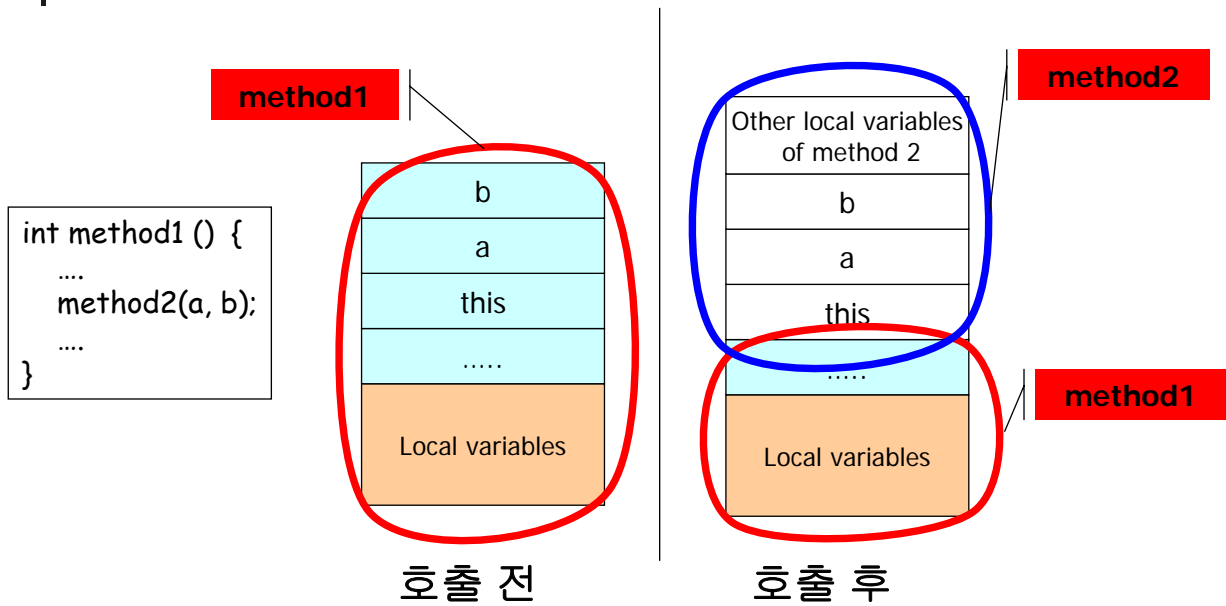


메소드 호출과 프레임

- 메소드 *m* 호출 전
 - 호출의 대상 객체(target object) 주소를 스택에 적재
 - 실 매개변수 값들을 스택에 적재
- 메소드 *m* 호출 후
 - *m*을 위한 새로운 프레임을 자바 스택에 적재
 - 호출자의 오퍼랜드 스택으로부터 *m*의 실 매개변수를 제거
 - *m*의 지역변수(형식 매개변수)에 실 매개변수 값 복사
 - *m*의 다른 지역 변수 초기화

30

메소드 호출 예



31

invokevirtual

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

```
Method int add12and13
0  aload_0           // Push local variable 0 (this)
1  bipush 12        // Push int constant 12
3  bipush 13        // Push int constant 13
5  invokevirtual #4 // Method Example.addtwo(II)I
8  ireturn          // Return int on top of operand stack
                        // it is the int result of addTwo()
```

32



invokestatic

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

```
Method int add12and13  
0 bipush 12  
2 bipush 13  
4 invokestatic #3 // Method Example.addTwoStatic(II)I  
7 ireturn
```

33



메소드 테이블(Method Table)

- 클래스 당 자료 구조
 - 클래스의 메소드들에 대한 포인터를 갖는 테이블
 - 메소드 호출을 위한 자료구조
 - 각 객체는 해당 클래스의 메소드 테이블 포인터를 갖는다.
- 메소드 호출 구현
 - 객체로부터 메소드 테이블 포인터를 얻는다.
 - 메소드 아이디를 이용하여 해당 메소드의 주소를 얻는다.
 - 그 주소로 점프한다.
- 상속과 재정의
 - 서브클래스는 슈퍼클래스의 메소드 테이블을 상속 받는다.
 - 메소드가 재정의되면 상응하는 메소드 테이블 엔트리도 갱신된다.

34

클래스와 메소드 테이블

```
class A {
    int foo() {...}
    void bar() {...}
};

class B extends A {
    int foo() {...}
    float boo() {...}
};
```

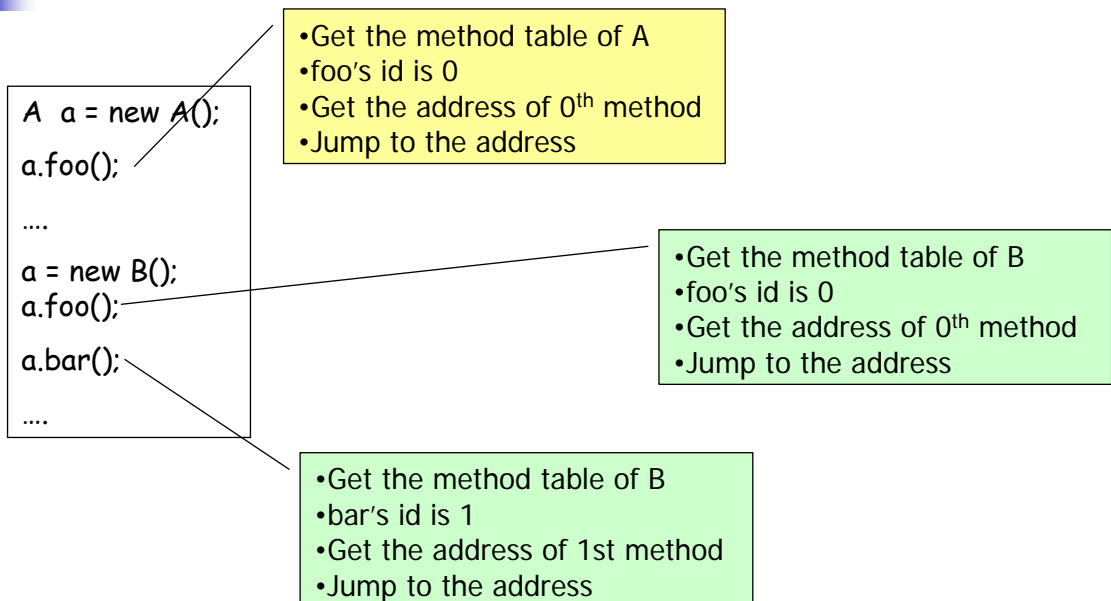
address of foo()	0
address of bar()	1

Method table of A

address of foo()	0
address of bar()	1
address of boo()	2

Method table of B

메소드 호출 예



예외 처리

- 예외 관리자
 - 예외 발생 지점의 문맥(context)를 저장한다.
 - 예외를 처리할 수 있는 가장 가까운 *catch* 절을 찾는다.
 - 발생한 예외의 타입과 예외 테이블의 *catch* 절에 선언된 타입을 비교한다.
 - 찾으면 예외(에 대한 참조)를 오퍼랜드 스택에 넣고 *catch* 절을 실행한다.
 - 찾지 못하면 해당 자바 스레드를 종료한다.
- 정상적인 실행 흐름은 *catch* 절의 예외 처리기와 분리된다.

37

JVM의 *try-catch*

```
void catchOne() {
    try {
        tryItOut();
    } catch (TestExc e) {
        handleExc(e);
    }
}
```

Method void catchOne()

```
0 aload_0 // Beginning of try block
1 invokevirtual #6 // Method Example.tryItOut()V
4 return // End of try block; normal return
```

```
5 astore_1 // Store thrown value in local variable 1
6 aload_0 // Push this
7 aload_1 // Push thrown value
8 invokevirtual #5 // Invoke handler method:
// Example.handleExc(LTestExc;)V
11 return // Return after handling TestExc
```

Exception table:

From To Target Type

```
0 4 5 Class TestExc
```

38



JVM의 도전적인 주제

- 실행 속도
 - 해석 오버헤드(Interpretation overhead)
 - 가상 호출 대 직접 호출(Virtual call vs. direct call)
 - 쓰레드 동기화(Thread synchronization)
 - 예외 관리(Exception management)
- 메모리 관리
 - 메모리 관리의 안전성
 - 정교한 쓰레기 수집

39



JVM 구현 사례들

- Kaffe
 - Most famous open-source JVM
 - Support multiple platforms
 - Poor performance
- SUN HotSpot JVM
 - Pioneer of feedback-directed dynamic compiler
 - Based on SELF compiler from Stanford
- SNU LaTTe JVM with classical JIT compiler
 - Outperform SUN HotSpot
 - Being integrated with Kaffe
- IBM Jalapeno JVM and Jikes RVM
 - Research JVM implemented in Java
 - Feedback-directed dynamic compiler
 - JVM itself is dynamically translated.

40



자바 바이트코드 검증

[Leroy01]

41



개요

- 바이트코드 검증
 - 신뢰할 수 없는 코드를 내려받기 하는 구조에서 핵심적인 보안 구성요소
- 내용
 - 알려진 검증 기술과 문제점
 - 정형 기법 적용 예

42



웹 애플릿: 90년대의 바이러스?

- 웹 애플릿
 - 민감한 컴퓨터에서 신뢰할 수 없는 코드의 자동 실행
 - 모바일 코드, 서블릿, 자바 스마트카드 등도 역시
- 보안상의 문제점
 - 무결성(integrity): 중요한 데이터의 손상
 - 비밀성(Confidentiality): 민감한 정보의 누출
 - 트로이 목마 공격, 바이러스 같은 동작

43



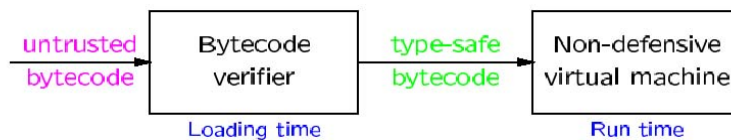
모래상자 모델(Sandbox model)

- 기본 아이디어
 - 애플릿을 하드웨어에서 직접 수행하지 않고 소프트웨어 층(모래상자)에서 실행한다.
- 접근 제어
 - 접근 제어를 제공하는 안전한 API만 사용
 - 파일 접근 제어
 - 네트워크 접근 제어
- 타입 안전성을 보장하는 가상 기계
 - 바이트코드 검증
 - 데이터 무결성, 코드 무결성, 가시성 조정자

44

가상 기계에서 타입 안정성

- 방어적 가상 기계(Defensive virtual machine)
 - 바이트코드 실행하면서 동적 타입 검사
 - 실행 속도가 상당히 떨어진다.
- 적재 시 바이트코드 검증
 - 정적 데이터 흐름 분석을 이용한 타입 안정성 확보
 - 방어적 가상 기계 사용으로 인한 실행 속도 저하 방지



45

바이트코드 검증 내용

- 코드의 형식
 - 예: 명령어 중간으로 점프 금지
- 올바른 타입의 오퍼랜드에 명령어 적용
 - 예: `getfield C.f`는 클래스 `C` 혹은 그 서브클래스의 객체에 대한 참조를 받는다
- 스택 오버플로우/언더플로우
- 지역변수(레지스터)의 초기화
 - 예: 초기화되지 않는 레지스터의 데이터를 사용할 수 없다.
- 객체의 초기화
 - `new C`, `C`의 구성자 호출 후 사용
- 가시성 조정자
 - 예: 클래스 밖에서 전용 멤버 접근 금지

46



바이트코드 검증: 자료 흐름 분석

47



검증의 기본 아이디어

- 기본 아이디어
 - 방어적 가상 기계의 타입-수준 요약 해석
 - (Type-level abstract interpretation of a defensive virtual machine)
- 요약 도메인(Abstract Domain)
 - 오퍼랜드 스택과 레지스터(지역변수) 집합을 타입으로 요약
- 요약 연산자(Abstract Operator)
 - 명령어에 대해서 오퍼랜드 타입을 검사하고 결과 타입을 계산한다.

48

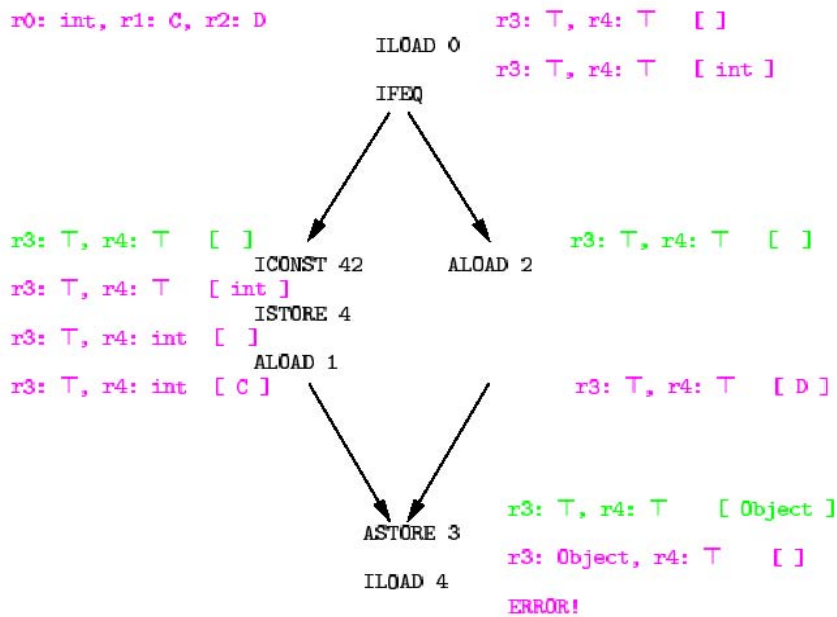
예

<code>class C {</code>	<code>ALOAD 0</code>	<code>r0: C, r1: int, r2: T</code>	<code>[]</code>
<code> int x;</code>		<code>r0: C, r1: int, r2: T</code>	<code>[C]</code>
<code> void move(int delta) {</code>	<code>GETFIELD C.x : int</code>	<code>r0: C, r1: int, r2: T</code>	<code>[int]</code>
<code> int oldx = x;</code>	<code>DUP</code>	<code>r0: C, r1: int, r2: T</code>	<code>[int ; int]</code>
<code> x += delta;</code>	<code>ISTORE 2</code>		
<code> D.draw(oldx, x);</code>		<code>r0: C, r1: int, r2: int</code>	<code>[int]</code>
<code> }</code>	<code>ILOAD 1</code>	<code>r0: C, r1: int, r2: int</code>	<code>[int ; int]</code>
<code>}</code>	<code>IADD</code>	<code>r0: C, r1: int, r2: int</code>	<code>[int]</code>
<code>this: r0</code>	<code>ALOAD 0</code>	<code>r0: C, r1: int, r2: int</code>	<code>[int ; C]</code>
<code>delta: r1</code>	<code>SETFIELD C.x : int</code>	<code>r0: C, r1: int, r2: int</code>	<code>[]</code>
<code>oldx: r2</code>	<code>ILOAD 2</code>	<code>r0: C, r1: int, r2: int</code>	<code>[int]</code>
	<code>ALOAD 0</code>	<code>r0: C, r1: int, r2: int</code>	<code>[int ; C]</code>
	<code>GETFIELD C.x : int</code>	<code>r0: C, r1: int, r2: int</code>	<code>[int ; int]</code>
	<code>INVOKESTATIC D.draw : void(int,int)</code>	<code>r0: C, r1: int, r2: int</code>	<code>[]</code>
	<code>RETURN</code>		

분기 처리

- 분기는 일반적인 자료 흐름 분석처럼 처리한다.
- 분기 지점
 - 타입을 모든 successors에게 전달한다.
- 조인 지점
 - 모든 predecessor의 타입들의 lub를 취한다.
- 반복
 - 타입이 더 이상 변하지 않을 때까지 반복한다.

예



정형화

- 타입-수준 VM의 전이 관계(transition relation) 정의

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

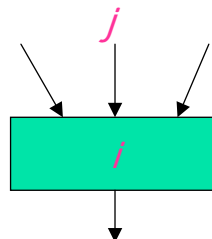
e.g. $iadd : (r, \text{int.int.s}) \rightarrow (r, \text{int.s})$

- Dataflow equation 설정

$$i : in(i) \rightarrow out(i)$$

$$in(i) = lub\{out(j) \mid j \text{ predecessor of } i\}$$

$$in(i_{start}) = ((P_0, \dots, P_{n-1}, \top, \dots, \top), \epsilon)$$



- 표준 고정점 계산(standard fixpoint iteration)으로 해 계산
- 안전성(Correctness) 증명됨



요약 연산자

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

53



바이트코드 검증: 세부 사항

54



문제점

- 인터페이스(Interface)
 - 서브타입 관계가 세미-라티스가 아님
 - 실행 시간 타입 검사 사용[Sun의 방법]
 - Dedekind completion 사용[Knoblock & Rehof]
- 객체 초기화(Object initialization)
 - must-alias 분석 필요
 - 초기화되지 않은 객체에 대한 두 개의 참조가 스택에 있는 경우
- 서브루틴(Subroutine)

55

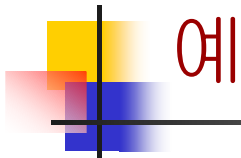


JVM에서 서브루틴

- try ... finally를 컴파일할 때 코드 중복을 피하기 위해 사용.

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

56



예

Without subroutines

```

...
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

```

With subroutines

```

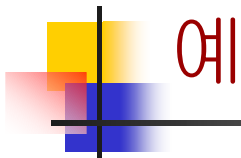
...
iload cond
ifne Early_return
...
jsr Subroutine
...

Early_return:
compute e
istore 2
jsr Subroutine
iload 2
ireturn

Exception_handler:
astore 2
jsr Subroutine
aload 2
athrow

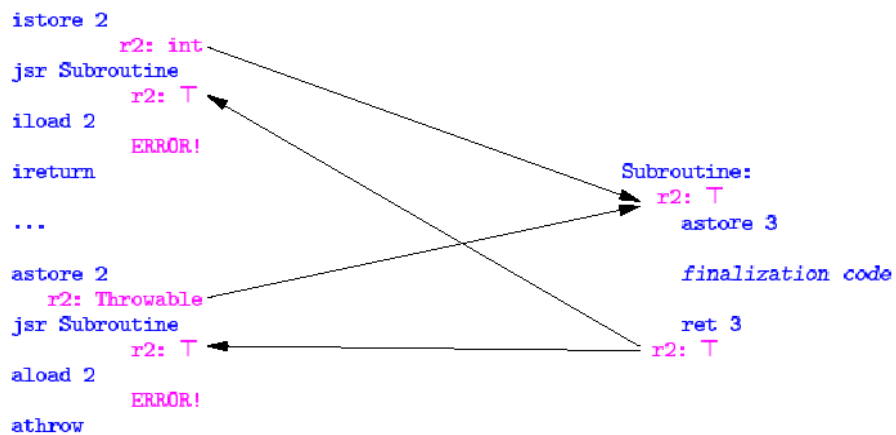
Subroutine:
astore 3
finalization code
ret 3

```



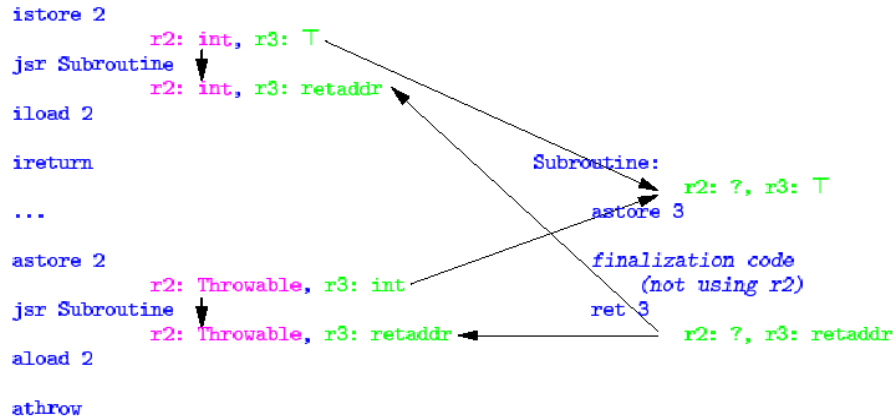
예

- jsr과 ret를 분기처럼 취급하면 지나친 타입 합병이 일어난다.



솔루션 1: Sun 방법

- 서브루틴 내에서 사용되지 않는 레지스터들은 jsr 후에도 자기 타입을 유지한다.



59

이 방법의 문제점

- 매개변수 다형성과 유사

Subroutine: $\forall \alpha. \{r2: \alpha; r3: T\} \rightarrow \{r2: \alpha; r3: \text{retaddr}\}$

- 문제점

- jsr과 대응되는 ret 찾기
- 각 서브루틴이 사용하는 레지스터 결정

- 서브루틴 구조 분석이 필요

- 서브루틴 구조가 구문적으로 정해져 있지 않음
- [Abadi & Stata]: 예외를 고려하지 않음
- [Qian]: Sun의 구현과 거의 비슷함

60

솔루션2: 문맥-민감 분석

- 문맥-민감 분석(Context-sensitive analysis)
 - 서브루틴을 각 호출 사이트에 대해서 한 번씩 분석
 - 호출 문맥: (스택 타입, 레지스터 타입)
- 호출 문맥(Calling context)

ϵ outside of any subroutine
 L in a subroutine called from PC L
 $L_1.L_2$ in a subroutine called from L_2 ,
itself in a subroutine called from L_1

61

예

```
istore 2
A: jsr Subroutine
    $\epsilon$ : r2: int, r3: T
    $\epsilon$ : r2: int, r3: retaddr
iload 2
ireturn
...
astore 2
    $\epsilon$ : r2: Throwable, r3: int
B: jsr Subroutine
    $\epsilon$ : r2: Throwable, r3: retaddr
aload 2
athrow
```

Subroutine:
A: r2: int, r3: T
B: r2: Throwable, r3: int
astore 3
finalization code
ret 3
A: r2: int, r3: retaddr
B: r2: Throwable,
r3: retaddr

62

솔루션 3: 모델 검사

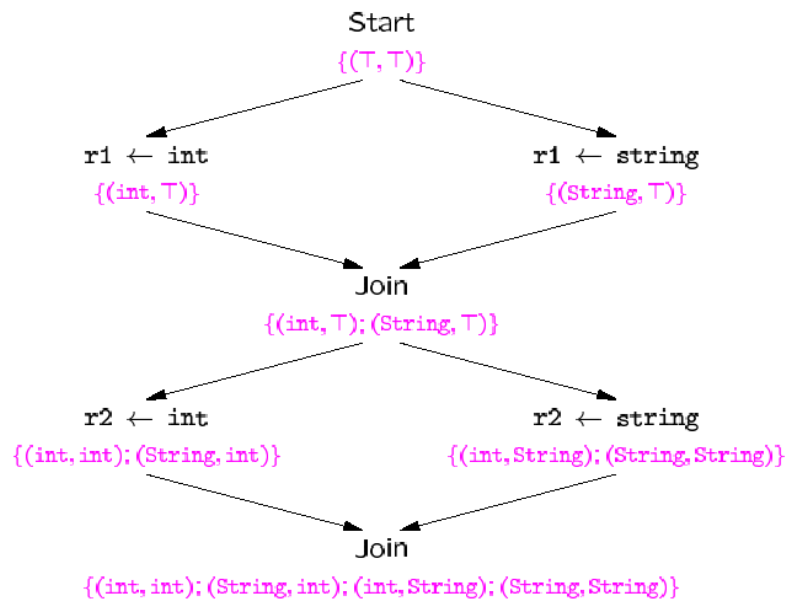
- 타입-수준 VM를 전이 관계로 모델링

$$(PC_{instr}, \tau_{regs}, \tau_{stack}) \rightarrow (PC_{succ}, \tau'_{regs}, \tau'_{stack})$$

- 모든 가능한 상태들을 검사한다.
- 타입 합병을 전혀 하지 않음
- 계산 비용은 많지만 다른 분석들의 안전성 증명에 유용하다.

63

예



64

경량 바이트코드 검증

65

Java-가능 스마트 카드

- Java 카드는 애플릿의 다운로드를 지원하고 엄격한 보안이 필요하다.



- 자바 카드는 제한된 자원만을 제공한다.
 - 2 킬로 바이트 RAM
 - 16 킬로 바이트 EEPROM
 - 느린 8 비트 프로세서

66



On-card 검증

- 자바 카드 상에서 바이트코드 검증
 - 시간: 고정점 계산은 복잡한 계산
 - 기억공간: Sun 알고리즘의 필요 메모리

$$3 \times (M_{stack} + M_{regs}) \times N_{branch}$$

(각 분기 지점에서 유도된 타입 정보 저장)

- 예 $M_{stack} = 5, M_{regs} = 15, N_{branch} = 50 \Rightarrow 3450 \text{ bytes.}$

67



솔루션: 경량 바이트코드 검증

- [Rose & Rose] 알고리즘
 - 일종의 Proof Carrying Code
 - 코드와 함께 각 분기 지점의 스택과 레지스터 타입 전달
 - 검증기는 이 정보를 이용하여 단지 검사만 한다.
- 장점
 - 고정점 계산은 피할 수 있고 한 패스면 충분하다.
 - Certificate는 읽기만 하면 되므로 EEPROM에 저장 가능
- 한계
 - 서브루틴 처리 방법
 - Certificate가 상당히 크다(코드의 20 ~ 100%)

68



결론

- 정형 기법을 이용한 바이트코드 검증
 - 산업체 요구가 많은 분야 (특히 스마트 카드)
 - 작고 잘 정의된 문제
 - 복잡한 알고리즘

- 향후 연구 분야
 - 바이트코드가 아닌 실제 기계 코드 검증
 - 타입 어셈블리어(Typed Assembly Language)
 - 타입 안전성 이상의 검증
 - 자원 사용 (메모리, 시간, 반응성)
 - 접근 제어(access control), 정보 흐름(information flow)
 - 자바 API 검증
 - API의 보안 관련 성질들을 정형적으로 검증