자바에서 동적인 클래스 로딩과 링킹의 분석

김기태*, 고훈준, 조선문, 심현진, 강성관, 유원희
*인하대학교 전자계산공학과
e-mail:g2011493@inhavision.inha.ac.kr

Analysis of Dynamic Class Loading and Linking in Java

Ki-Tae Kim*, Hoon-Joon Kouh, Sun-Moon Jo, Hyun-Jin Sim, Sung-Kwan Kang, Weon-Hee Yoo
School of Computer Science, Inha University

요 약

자바의 동적 클래스 로딩은 실행 시간에 소프트웨어 컴포넌트를 동적으로 로딩하기 위한 강력한 메커니즘이다. 다른 시스템에서도 동적 로딩과 링킹을 제공하지만 지연 로딩, 타입 안전 링크, 사용자 정의 클래스 로딩 정책, 그리고 다중 이름 공간 등은 자바가 가진 중요한 특징이다. 자바에서 클래스 로딩의 핵심은 타입 안전에 대한 확신이다. 하지만 자바 가상 머신에서 타입 안전에 대한 메커니즘은 매우 복잡하고 또 정확성에 대한 접근이 명확하지 않아서 지금까지 많은 버그가 발생되었고 따라서 타입 안전에 문제가 되어왔다.

본 논문은 간단한 자바 소스 코드를 이용하여 동적인 클래스 로더의 동작을 분석하여 도식화하고, 연산적 의미론으로 추상화하여 이전에 제시되었던 타입 안전에 대한 문제를 분석한다.

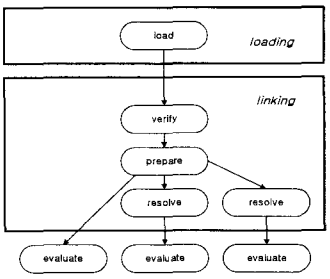
1. 서론

자바의 동적인 클래스 로딩은 자바 플랫폼에서 실행 시간에 소프트웨어 컴포넌트를 동적으로 로딩하기 위한 강력한 메커니즘이다[7]. 다른 시스템에서도 동적 로딩과 링킹을 제공하지만 지연 로딩, 타입 안전 링크, 사용자 정의 클래스 로딩 정책, 다중 이름 공간 등은 자바가 가진 중요한 특징이다[1,8]. 자바 가상 머신의 초기 버전의 클래스 로더는 실행 시간 객체이기 때문에 클래스에 대한 로딩이 단지 이름에 의해서만 이루어졌고, 그것을 로딩하는 클래스 로더는 포함되지 않았다. 따라서 클래스를 로딩할 때, 명확한 클래스의 이름이 요구되었다[5,6].

클래스 로딩에서 중요한 것은 타입 안전에 대한 확신이다. 하지만 자바 가상 머신에서 타입 안전에 대한 메커니즘은 매우 복잡하고, 정확성에 대한 접근이 명확하지 않아서 지금까지 많은 버그가 발생되었고 또한 타입 안전에 문제가 되어왔다[10,11]. 따라서 본 논문에서는 기존의 타입 안전 문제를 분석하고 이를 바탕으로 자바의 동적인 클래스 로딩을 연산적 의미론(*operational semantics*)을 통해서 설명한다.

2. 클래스 로딩 과정

클래스의 로딩 과정을 살펴보면 로딩(*loading*)과 링킹(*linking*)으로 나눌 수 있다[2,3,4,9]. 로딩은 클래스의 이름을 사용하여 클래스 파일 형태의 바이트들을 찾고, 이를 자바 가상 머신에게 알리는 작업을 수행한다. 링킹은 클래스가 기본적인 형태를 갖추고 있는가와 가상 머신의 보안과 관련된 제약 조건을 거스르지 않음을 보장할 수 있도록 클래스를 검증하는 단계를 수행하고, 그리고 정적 초기화를 수행하는 메소드를 호출한다.

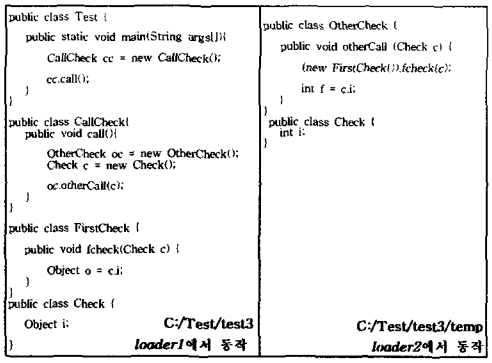


[그림 2-1] 로딩 과정

[그림2-1]은 가상 머신의 동작을 크게 로딩과 링킹으로 나누어 도식화 한 것이다. [그림 2-1]에서 링킹 과정은 세분화되어 표현되어진다. 로딩된 클래스는 검증 과정과 준비 단계, 그리고 결정 단계를 진행한 후 평가되어 진다.

3. 동적인 로딩 분석

자바의 클래스 로딩에서 핵심은 타입 안전에 대한 확신을 주는 것이다. 동적인 클래스 로딩의 동작과 타입 안전에 대한 확신을 높이기 위해 간단한 코드를 이용하여 동적인 클래스 로딩을 분석하여 도식화하고 형식적인 방법으로 접근한다.

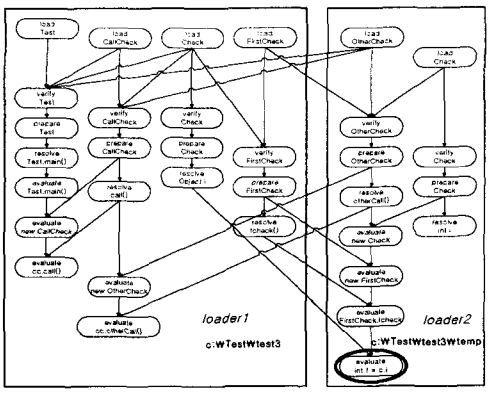


[그림 3-1] 동적인 클래스 로딩 예제

[그림 3-1]은 동적인 클래스 로딩을 표현한 간단한 예제이다. 클래스 *ClassLoader*는 $loader_1$과 $loader_2$ 라는 두 가지 다른 객체를 가지며, *Test, CallCheck, FirstCheck, OtherCheck,* $Check_1$, 그리고 $Check_2$ 등 6개의 클래스를 가진다고 가정한다. 클래스 *Test, CallCheck, FirstCheck,* $Check_1$는 $loader_1$을 정의 로더로 가지며, 클래스 *OtherCheck,* $Check_1$는 $loader_2$를 정의 로더로 가진다. 여기서 $Check_1$과 $Check_2$는 같은 이름을 가진 클래스이다. *OtherCheck*와 *Check* 클래스를 위해 초기화 로더처럼 $loader_1$을 사용하여 클래스 *OtherCheck*와 $Check_1$을 생산하고, 반면에 $loader_2$는 초기화 로더처럼 사용되어 *FirstCheck*와 *Check* 클래스를 이용하여 클래스 *FirstCheck*와 $Check_2$ 클래스를 생성한다. [그림 3-1]에서 $loader_1$은 클래스 *OtherCheck*를 로딩하는 것을 $loader_2$에게 위임한다. 그리고 $loader_2$는 클래스 *FirstCheck* 클래스의 로딩을 $loader_1$에게 위임한다.

[그림 3-1]의 *public void otherCall (Check c)*에서 자바 가상 머신은 $Check_2$가 로드되기 전까지는 $Check_1$에 있는 *i*의 타입이 $Check_2$의 형식 인자 *c*와 매치 되는지 확인하지 않는다. 그래서 *otherCall (Check c)* 메소드가 수행될 때 예외가 발생하게 된다. 만약 이 메소드에 *c.i*가 존재하지 않는다면 자바 가상 머신은 예외를 발생시키지 않을 것이다. 타입 안전을 증명하는 보통의 접근 방법은 실행되는 동안 할당된 변수에 항상 같은 타입의 값이 있는가를 보이는 것이다. *JDK1.1* 버전에서는 *otherCall(Check c)* 메소드에서 *c.i* 필드에 대한 접근이 수행될 때, 클래스 $Check_1$과 $Check_2$의 동등성을 체크하지 못했다. 초기 버전의 자바 가상 머신에서는 같은 이름의 메소드가 어떤 변수에 접근할 때, 클래스의 동일성에 대한 체크를 하지 않았다. 그래서 이러한 필드의 접근은 예상할 수 없었고 잘못된 타입이 동작하기도 하였다.

[그림 3-2]는 [그림 3-1]의 동작을 그림으로 도식화한 것이다. [그림 3-2]에서 $loader_2$의 *OtherCheck* 클래스



[그림 3-2] 클래스의 동작 분석

에서 *c.i*를 평가할 때 *NoSuchFieldError*가 발생한다. 그 이유는 $loader_2$에 의해 로딩된 클래스 *OtherCheck*에서 *c.i*에 대한 평가가 이루어 질 때, 객체 *c*의 멤버 필드 *i*에 대한 접근을 시도하는데, 이때 접근되어야 할 멤버필드 *i*의 타입은 정수형이기를 기대하는데 실제로 평가된 것은 $loader_1$에 의해 로딩된 *Object* 타입의 *i*가 나타나기 때문에 *NoSuchFieldError*가 발생하게 된다.

[그림 3-1]에 대해 컴파일 시간에 *Test* 코드의 동작은 모든 클래스들이 오류 없이 로딩된다. 하지만 [그림 3-1]에 대해 실행시간에 *Test*코드의 동작을 보면 [그림 3-3]과 같이 예외가 발생한다.

```
[Loaded Test]
[Loaded CallCheck]
[Loaded OtherCheck]
[Loaded Check]
[Loaded FirstCheck]
[Loaded java.lang.NoSuchFieldError from C:\j2sdk1.4\jre\lib\rt.jar]
java.lang.NoSuchFieldError: i
[Loaded java.lang.StackTraceElement from C:\j2sdk1.4\jre\lib\rt.jar]
        at OtherCheck.otherCall(OtherCheck.java:12)
        at CallCheck.call(CallCheck.java:13)
        at Test.main(Test.java:12)
Exception in thread "main" Normal Termination
```

[그림 3-3] 실행시간에 *Test* 코드의 동작

[그림 3-3]에서는 모든 클래스의 로딩이 실행시간까지 지연되는 지연로딩의 결과를 볼 수 있다. [그림 3-3]의 예외는 [그림 3-2]에서 본 것과 마찬가지로 *i* 필드에 대한 접근 문제가 존재하기 때문에 발생된다. 초기 버전의 자바 가상 머신은 이와 같은 문제를 발견하지 못하였다. 그 이유는 같은 이름을 가진 클래스가 다른 로더에 의해 로딩되었을 때, 초기 버전의 가상 머신은 단지 이름만으로 클래스를 구별하였기 때문에 그와 같은 문제가 발생하였지만 최근 버전에서는 로드된 클래스 캐쉬(*loaded class cache*), 로딩 제한(*loading constraints*), 서브타입 제한(*subtype constraints*) 등을 제시하여 이러한 문제를 해결하였다.

본 논문에서 실행되는 자바 가상 머신의 상태는 로

드된 클래스 캐쉬, 로딩의 집합, 서브타입 제한, 객체를 저장하기 위한 힙으로 구성된 전역 상태로 구성된다. 상태는 프레임에 저장된 단일 스레드의 스택 수행을 위한 컴포넌트를 포함한다. 각 프레임은 클래스, 클래스의 메소드, 메소드의 프로그램 카운터, 그리고 지역 메모리(오퍼랜드 스택과 지역 변수)의 상태를 가진다.

[그림 3-1]의 과정을 정규화 표현으로 나타내기 내기 위해서는 몇 가지 정의가 필요하다. 우선 클래스와 클래스 로더에 대해 정의하면 [표 3-1]과 같다.

[표 3-1] 클래스 정의

$cls\_nm : ClassFile \rightarrow Class$
$sup\_nm : ClassFile \rightharpoonup Class$
$cls\_mem : ClassFile \rightarrow Package(Member)$
$refs : ClassFile \rightarrow Package(Class)$
$cls\_f : ClassFile$
$\forall m \in member(clsf).\ ClassMeber(m) = cls\_nm(cls\_f)$
$sys\_cls\_ld \in LoadedClass$
$cls\_ld \in LoadedClass,\ cls\_f \in ClassFile \Rightarrow$
$(cls\_f,\ cls\_ld) \in LoadedClass$
$\rightharpoonup$: partial function

[표 3-1]은 *cls_nm, sup_nm, cls_mem, refs, cls_f* 등 클래스와 관련된 정보를 표현한 것이다. *cls_nm*은 클래스 파일로부터 클래스 이름을 가져오는 것이고 *sup_nm*은 상위 클래스 파일의 이름을 가져오는 것이다. 예를 들면 *sup_nm(cls_f(Check))*라 하면 클래스 파일인 *Check*의 상위 클래스를 찾는데 [그림 3-1]에서 *Check* 클래스의 상위 클래스는 *Object* 클래스이기 때문에 *sup_nm(cls_f(Check)) = java.lang.Object* 라고 나타낸다. 그리고 *sup_nm : ClassFile ⇀ Class* 에서 ⇀ 은 부분 함수(*partial function*)를 표현하는데 이유는 *Object*클래스에서는 그 상위 클래스가 없기 때문에 →이 아닌 ⇀ 인 부분함수로 표현한다.

*sys_cls_ld*는 시스템 클래스 로더를 표현하는데 *classpath*에 있는 디렉토리 리스트로부터 클래스를 로딩 할 수 있다. 논문에서는 */test3*와 */test3/temp*는 같은 클래스 패스에 있다. *cls_ld*는 클래스 로더를 표현하고, *cls_f*은 클래스 파일을 표현한다. *c.cls_f*은 로드된 클래스 *c*의 *class file*을 표현하는 것이다. *c.cls_ld*는 로드된 클래스 *c*의 클래스로더를 표현한다. 그리고 *LoadedClass*는 로드된 클래스의 집합을 의미한다.

[표 3-2] 서브클래스 관계

S : set S of loaded classes
$subR \subseteq LoadedClass \times LoadedClass$
$c,\ c' \in S$
$sup\_nm(cls\_f(c)) = cls\_nm(cls\_f(c'))$

$$\frac{c.cls\_ld = c'.cls\_ld}{S \rightleftharpoons c\ subR\ c'} \quad (1)$$

$$\frac{}{S \rightleftharpoons c\ subR^{*}\ c'} \quad (2)$$

$c,\ c',\ c'' \in S$

$$\frac{S \rightleftharpoons c\ subR\ c'' \qquad S \rightleftharpoons c''\ subR^{*}\ c'}{S \rightleftharpoons c\ subR^{*}\ c'} \quad (3)$$

서브클래스는 [표 3-2]와 같이 표현할 수 있다. [표 3-2]에서 S는 로드된 클래스들의 집합을 표현한다. $S \rightleftharpoons c\ subR\ c'$는 서브 클래스의 관계를 표현한다. c와 c'이 로드된 클래스들의 집합에 속하고 $c.cls_f$의 상위 클래스의 이름이 $c'.cls_f$과 같고 두 클래스 파일을 로딩한 로더가 동일하다면 클래스 c는 c'의 서브 클래스라고 할 수 있다. 또한 서브 클래스의 관계인 $subR^{*}$는 $subR$에 대해 반사와 전사의 관계를 가진다. 자바 가상 머신의 동작 중 로딩은 [표 3-3]과 같이 표현할 수 있다.

[표 3-3] loading 동작

$W : LoadedClass \times Class \rightharpoonup LoadedClass$
$W(cls\_ld,\ cls) = c \Rightarrow cls\_nm(c,\ cls\_f) = cls$

$$\frac{W(cc.cls\_ld,\ java.lang.Object) = cls\_nm}{S \rightleftharpoons load(cc,\ java.lang.Object) \Rightarrow S \cup cls\_nm} \quad (4)$$

$$\frac{W(cc.cls\_ld,\ cls) = cls\_nm \qquad sup\_nm(cls\_f(cls\_nm)) = sup\_cls\_nm \qquad S \rightleftharpoons load(cc,\ sup\_cls\_nm) \Rightarrow S'}{S \rightleftharpoons load(cc,\ cls) \triangleright S' \cup cls\_nm} \quad (5)$$

W는 클래스 로더의 동작을 나타내는 함수이다. 이 함수에 의해 주어진 클래스 로더와 클래스 이름은 로드된 클래스를 반환한다. *cls*는 클래스를 *cc*는 현재 클래스를 나타내고 *sup_cls_nm*은 수퍼클래스의 이름을 나타낸다. $S \rightleftharpoons op(cc,\ cls) \Rightarrow S'$ 는 연산자 *op*가 수행되면 상태 S에서 상태 S'로 변화되는 것을 나타낸다.

다음으로 링킹 동작을 표현하면 다음과 같다.

[표 3-4] linking 동작

$$\frac{S \rightleftharpoons load(cc,\ java.lang.Object) \Rightarrow S' \qquad W(cc.cls\_ld,\ java.lang.Object) = cls\_nm \qquad S' \rightleftharpoons verify(cc,\ cls\_nm) \Rightarrow S''}{S \rightleftharpoons link(cc,\ java.lang.Object) \Rightarrow S''} \quad (6)$$

$$\frac{S \rightleftharpoons load(cc,\ cls) \Rightarrow S' \qquad c_n = W(cc.cls\_ld,\ cls) \qquad super(cls\_f(cls\_nm)) = sup\_cls\_nm \qquad S' \rightleftharpoons link(cc,\ sup\_cls\_nm) \Rightarrow S'' \qquad S'' \rightleftharpoons verify(cc,\ cls\_nm) \Rightarrow S'''}{S \rightleftharpoons link(cc,\ cls) \Rightarrow S'''} \quad (7)$$

링크 동작에서는 클래스를 *load(cc, cls_nm)*에 의해 로딩하고, *verify (cc, cls_nm)*에 의해 클래스가 기본적인 형태를 제대로 갖추고 있는지, 가상 머신의 보안과 관련된 제약 조건을 거스르지 않음을 보장할 수 있도록 클래스를 검증하는 단계를 수행한다.

지금까지 정의한 내용들을 이용하여 [그림 3-2]를 분석해보면 다음과 같다. $loader_1$에 의해 로드된 클래스들은 *Test, CallCheck, FirstCheck, Check*$_1$ 클래스이고 $loader_2$에 의해서 로드된 클래스는 *OtherCheck*와 *Check*$_2$ 클래스이다. 오류가 발생한 *OtherCheck*클래스를 중심으로 살펴보면 *OtherCheck* 클래스는 $loader_2$에 의해 로드되었고, *OtherCheck*의 현재 로더 역시 $loader_2$이다. 현재 로더를 *cc.loader* = $loader_2$로 나타낼 수 있다. *OtherCheck*에서의 동작을 보면 다음과 같이 나타낼 수 있다.

$$\frac{S \rightleftharpoons load(cc, OtherCheck) \Rightarrow S' \quad W(cc.cls\_Id, OtherCheck) = (cls\_f(OtherCheck), loader_2) \quad sup\_nm(cls\_f(OtherCheck)) = java.lang.Object \quad S' \rightleftharpoons link(cc, java.lang.Object) \Rightarrow S' \quad S' \rightleftharpoons verify(cc, (cls\_f(OtherCheck), loader_2) \Rightarrow S''}{S \rightleftharpoons link(cc, OtherCheck) \Rightarrow S''}$$

여기서 S'와 S''은

$$S' = S \cup \{(cls\_f(OtherCheck), loader_2)\}$$
$$S'' = S' \cup \{(cls\_f(FirstCheck), loader_1), (cls\_f(Check), loader_2)\}$$

로 표현할 수 있다. 그리고 현재 클래스인 *OtherCheck* 클래스의 참조를 보면 다음과 같다.

$$refs(cls\_f(OtherCheck)) = \{FirstCheck, Check\}$$

OtherCheck 클래스는 *FirstCheck*와 *Check* 클래스를 참조한다. 이 클래스들의 로딩은 다음과 같이 표현된다.

$$S' \rightleftharpoons load^*(cc, \{FirstCheck, Check\}) \Rightarrow S' \cup \{(cls\_f(FirstCheck), loader_1), (cls\_f(Check), loader_2)\}$$

그리고 *OtherCheck* 클래스에서 오류가 발생한 필드 i에 대한 접근은 *getfield* 명령어를 통해서 이루어진다. *getfield(cc, (FirstCheck, i))*은 다음과 같이 표현되어진다.

$$S'' \rightleftharpoons getfield(cc, (FirstCheck, i)) \Rightarrow S'' \cup \{(cls\_f(Check), loader_1)\}$$

그리고

$$\frac{S'' \rightleftharpoons load(cc, OtherCheck) \Rightarrow S'' \quad S'' \rightleftharpoons link(cc, OtherCheck) \triangleright S'' \cup \{(cls\_f(Check), loader_1)\} \quad \{Check\} = refs(cls\_f(FirstCheck)) \quad S'' \rightleftharpoons load((cls\_f(FirstCheck, loader_1), Check) \Rightarrow S'' \cup \{(cls\_f(Check), loader_1)\}}{S'' \rightleftharpoons verify((cls\_f(FirstCheck, loader_1), Check) \Rightarrow S'' \cup \{(cls\_f(Check), loader_1)\}}$$

이 때 로드된 클래스들의 상태는

$$S \cup \{(cls\_f(OtherCheck), loader_2), (cls\_f(FirstCheck), loader_1), (cls\_f(Check), loader_2), (cls\_f(Check), loader_1)\}$$

이고

$$cls\_nm((cls\_f(Check), loader_2)) \neq cls\_nm((cls\_f(Check), loader_1))$$

인 관계가 성립하기 때문에 이때 찾고자하는 i에 해당하는 필드를 찾지 못하게 된다. 초기 버전에서는 이 부분에서 로더에 대해서는 고려하지 않고 단지 이름이 동일한가만 확인하였기 때문에 타입에 관련된 문제가 발생하였던 것이다. 이러한 문제는 클래스 이름을 표기하면서 함께 로더를 표시하여 주기 때문에 이름만 확인하여 클래스를 로딩하던 문제를 해결할 수 있고 정확한 클래스 로딩을 수행할 수 있다.

4. 결론

본 논문은 자바 가상 머신의 클래스 로더에 대한 동작을 분석하고 연산적 의미론으로 추상화하여 이전에 제시되었던 타입 안전에 대해 문제를 분석하였다. 그리고 자바 가상 머신에서 클래스가 로드된 상태를 정의하고, 로드와 링킹 동작을 정의하였다. 상태의 전위는 조건이 만족되어 질 때 이루어지고, 이러한 조건은 실제로 실행되는 자바 가상 머신에 의해 실행시간 체크에 상응하게 된다. 실패하게 되면 자바 가상 머신에서 예외가 발생하게 된다.

본 논문에서는 간단한 예제 코드를 이용하여 타입 안전에 대한 문제를 다양한 방법으로 분석하였다. 그리고 다중 클래스 로딩을 사용하고 지연 로딩 접근 방식으로 클래스 로딩을 접근하였다. 또한 클래스 로딩 동작을 다루기 위해 몇 가지 함수와 명령도 정의하였다. 하지만 동시성과 예외에 대해서는 고려하지 않아 향후과제에서는 이 문제에 대해 고려할 것이다.

참고문헌

[1] Bracha, *"A Critique of Security and Dynamic Loading in Java : Foramlization."*, Sun Java SoftWare, 1999

[2] Drossopoulou, S. *"Towards an abstract model of Java dynamic linking and verification."*, In ACM SIGPLAN Workshop on Type in Compilation(TIC)(Montreal, Canada, Sept.), pp. Paper 19. Computer Science Department, Carnegie Mellon University.

[3] Drossopoulou, S., Wragg, D. and Eisenbach, S., *"Is the Java type system sound?"* Theory and Practice of Object Systems 5, 1, 3-24., 1999

[4] Drossopoulou, S. and Eisenbach, S., *"Manifestations of Java Dynamic Linking"*

[5] Fritzinger,j. S. and Mueller, M., *"Java Security"*, Sun Microsystems Inc, Mountain View, Calif. 1996.

[6] Gosling, J. Joy, B. Steele, G. and Bracha, G. "The Java Language Specification (second ed.)." Addision Wesley, Reading, Mass. 2000.

[7] Jensen, T., Metayer, D. L., and Thorn, T. *"Security and dynamic class loading in java: A formalization."*, In Computer Language, pp. 4-15. IEEE Comput. Soc. Press, Los Alamitos, Calif., 1998

[8] Liang, S. and Bracha, G., *"Dynamic class loading in the Java virtual machine."*, In Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) (Vancouver, Canada, Oct.), pp. 36-44. SIGPLAN Notices 33, 10. 1998.

[9] Qian, Z., Goldberg, A., and Coglio, A., 2000. *"A formal specification of Java™ class loading."*, In Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)(Minneapolis, Minnesota, Oct), pp. 325-336. ACM, New York. 2000.

[10] Saraswat, V., *"Java is not type-safe."*, Tech. Rep. (Aug.), AT&T Research, Florham Park, New Jersey. 1997.

[11] Tozawa, A, and Hagiya, M. *"Careful analysis of type spoofing."*, In C. H. Cap, Ed., JIT'99 Java Informations-Tage, pp. 290-296. Informatik aktuell, Springer-Verlag. 1999.