자바 바이트코드의 .NET MSIL 중간언어 번역기

정지훈*, 박진기*, 이양선*
*서경대학교 컴퓨터공학과
e-mail: {wpenguin, jkpark, yslee}@nit.skuniv.ac.kr

Java Bytecode-to-.NET MSIL IL Translator

Ji-Hoon Jung*, Jin-Ki Park*, Yang-Sun Lee*
*Dept. of Computer Engineering, Seokyeong University

요 약

자바는 썬 마이크로시스템즈사의 제임스 고슬링(James Gosling)에 의해 고안된 언어로 운영체제 및 하드웨어 플랫폼에 독립적인 차세대 언어로 최근에 가장 널리 사용하는 범용 프로그래밍 언어 중 하나이다. 자바 프로그램은 컴파일러에 의해 각 플랫폼에 독립적인 중간 코드 형태의 바이트코드로 변환된 클래스 파일로 생성되면 JVM(Java Virtual Machine)에 의해 실행된다. 마이크로소프트사의 .NET 플랫폼과 C# 언어는 프로그래머들의 요구를 충족시키고 썬사의 JVM 환경과 자바 언어에 대응하기 위해서 개발된 플랫폼과 언어이다. C#과 같은 .NET 언어는 컴파일러에 의해 MSIL(MicroSoft Intermediate Language) 코드로 번역되며 번역된 MSIL 코드는 .NET 플랫폼 환경에서 런타임 엔진인 CLR(Common Language Runtime)에 의해 실행이 된다.

자바로 작성된 프로그램은 JVM 플랫폼에서는 실행이 되지만 .NET 플랫폼에서 실행이 되지 않고, 반대로 C#과 같은 .NET 언어로 작성된 프로그램은 .NET 플랫폼에서는 실행이 되지만 JVM 플랫폼에서 실행이 되지 않는다. 이런 이유로 본 논문에서는 자바소스를 컴파일하여 생성된 클래스 파일에서 Oolong 코드를 생성하고 생성된 Oolong 코드를 .NET의 MSIL 코드로 변환하여 자바로 구현된 프로그램이 .NET 환경에서 실행되도록 하는 Bytecode-to-MSIL 번역기 시스템을 구현하였다. 따라서, 자바 프로그래머는 JVM이나 .NET 플랫폼 환경에 관계없이 프로그램을 작성하여 실행시킬 수 있다.

번역기 시스템의 구현을 정형화하기 위해 Oolong 코드의 명령어들을 문법으로 작성하였으며, PGS를 통해 생성된 어휘 정보를 가지고 스캐너를 구성하였으며, 파싱테이블을 가지고 파서를 설계하였다. 파서의 출력으로 AST가 생성되면 번역기는 AST를 탐색하면서 의미적으로 동등한 MSIL 코드를 생성하도록 시스템을 컴파일러 기법을 이용하여 모듈별로 구성하였다.

1. 서론

자바는 썬 마이크로시스템즈사의 제임스 고슬링(James Gosling)에 의해 고안된 언어로 운영체제 및 하드웨어 플랫폼에 독립적인 차세대 언어로 최근에 가장 널리 사용하는 범용 프로그래밍 언어 중 하나이다. 자바 프로그램은 컴파일러에 의해 각 플랫폼에 독립적인 중간 코드 형태의 바이트코드로 변환된 클래스 파일로 생성되면 JVM(Java Virtual Machine)에 의해 실행된다. 하지만 바이너리 코드로 되어있는 클래스 파일에서 바이트코드를 읽는다는 것은 상당히 어렵고 복잡한 일이다. 반면에 또 다른 형태의 자바 중간 언어인 Oolong 코드로 작성된 파일은 클래스 파일 내에 있는 바이트코드와 유사한 형태의 실제 프로그램 로직 부분을 텍스트 형식으로 저장하므로 프로그래머 입장에서 좀 더 쉽게 접근할 수 있기 때문에 코드의 이해와 프로그램의 작성 및 수정을 용이하게 한다[2,5,8,11,12].

한편, 마이크로소프트사의 .NET 플랫폼은 프로그래머들의 요구를 충족시키고 썬사의 JVM 환경과 자바 언어에 대응하기 위해 C# 언어를 새로이 개발하였다. C# 프로그래밍에서는 C/C++에서와 같이 직접적으로 포인터를 조작할 필요가 없다. C# 프로그래밍에서는 자바의 가비지 콜렉션과 같이 메모리 관리가 자동으로 되며, 비주얼 베이직에서처럼 클래스 프로퍼티라는 개념이 있고, C++처럼 클래스에서 연산자를 오버로드할 수도 있다. C#은 자바처럼 문법

본 연구는 한국과학재단 목적기초연구(R01-2002-000-00041-0) 지원으로 수행되었음.

적으로 깨끗하고, 비주얼 베이직처럼 쉽고, C++처럼 유연한 장점들을 가진 언어이다. 또한 C#은 다른 .NET 언어인 비주얼 베이직 닷넷이나 비주얼 C++ 닷넷 등의 언어와 공동으로 하나의 응용 프로그램을 개발할 수 있다. 이는 C#으로 짜여진 프로그램 소스는 컴파일 과정을 통해서 MSIL 코드를 만들어 내는데 .NET 플랫폼 언어는 어떤 프로그래밍 언어이든지 C#과 마찬가지로 MSIL 코드를 내기 때문에 가능하다. 이렇게 해서 생성된 MSIL 코드는 하드웨어에 독립적으로 특정 하드웨어 내에서 그 하드웨어에 맞는 .NET 플랫폼 환경에서의 런타임 엔진에 의해 실행이 된다[1,3,4,7,13].

자바로 작성된 프로그램은 JVM 플랫폼에서는 실행이 되지만 .NET 플랫폼에서 실행이 되지 않고, 반대로 C#과 같은 .NET 언어로 작성된 프로그램은 .NET 플랫폼에서는 실행이 되지만 JVM 플랫폼에서 실행이 되지 않는다. 이런 이유로 본 논문에서는 자바소스를 컴파일하여 생성된 클래스 파일에서 Oolong 코드를 추출하고 추출된 Oolong 코드를 .NET의 MSIL 코드로 변환하여 자바로 구현된 프로그램이 .NET 환경에서도 실행되도록 하는 Bytecode-to-MSIL 번역기 시스템을 구현하였다.

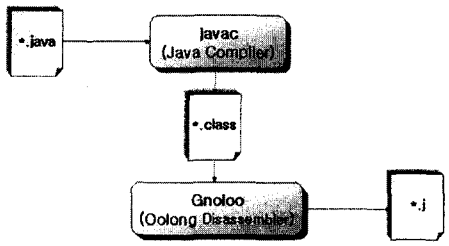
2. 바이트코드와 MSIL 코드

2.1 바이트코드

바이트코드[2,5,11,12]는 JVM의 기계언어로 간주할 수 있고, JVM 이 클래스 파일을 로드할 때, 클래스 내에서 각각의 메소드에 대하여 스트림 형태로 얻어지며, 이 때의 스트림은 8-bit의 바이트로 구성된 이진스트림 형태이다. 또한, 바이트코드는 기본적으로 스택 지향 구조를 가지고 있고, 처음부터 인터프리터를 목적으로 설계되었다.

바이트코드를 바이너리 형식으로 가지고 있는 클래스 파일은 분석하거나 수정하기가 매우 어렵다. 이에 비해 또 다른 형태의 자바 중간언어인 Oolong 코드[2,5,11]는 읽고 쓰기가 클래스 파일 형식에 비해서 훨씬 쉽다. 이 Oolong 코드는 존 메이어(John Meyer)의 Jasmin 언어를 기반으로 만들어 졌으며 프로그래머가 바이트코드 수준에서 프로그램을 작성할 수 있도록 설계되어 있다.

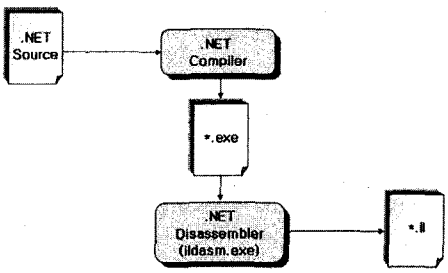
본 논문에서 개발한 번역기 시스템에 입력으로 들어갈 Oolong 코드는 자바 클래스 파일로부터 Oolong 코드를 추출하는 과정이 필요하다. [그림1]은 클래스 파일로부터 Oolong 코드를 추출하는 과정이다.



[그림1] Oolong 코드 추출과정

2.2 MSIL 코드

MSIL(MicroSoft Intermediate Language)[1,3,6,13]은 C#을 포함한 .NET 언어들의 중간언어로 .NET 언어로 작성된 소스코드가 컴파일되면 MSIL로 작성된 코드가 생성된다. MSIL은 오퍼런드 스택을 이용하는 스택 기반의 명령어 집합으로 언어 상호 운용성(Language Interoperability)과 플랫폼 독립성이라는 두 가지 큰 특징을 가진다. MSIL의 명령어 종류에는 산술/논리 연산, 제어 흐름, DMA, 예외처리, 메소드 호출 등이 있고, 객체지향 프로그래밍 구조에 영향을 주는 가상 메소드 호출, 필드 접근, 배열 접근, 객체 할당과 초기화 등도 MSIL에서 직접 지원하는 명령어 종류이다. 또한, 처음부터 JIT를 목적으로 설계되었으며, 자바언어와는 다르게 처음부터 언어독립적으로 설계되어 포괄적인 프로그래밍(Generic Programming)을 목적으로 하기 때문에 프로그램의 기능 및 구조의 변화에 잘 적응하는 언어이다. [그림2]는 MSIL 코드의 추출과정을 나타낸 것이다.



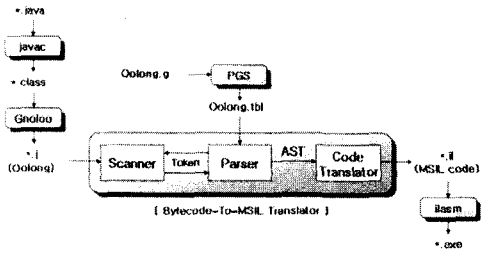
[그림2] MSIL 코드 추출과정

3. Bytecode-to-MSIL 번역기 시스템

3.1 시스템 구성

자바 바이트코드의 .NET MSIL 중간언어 번역기 시스템은 Oolong 코드 추출과정에 의해서 생성된 *.j 파일을 입력으로 받아서 .NET 플랫폼에서 실행

될 수 있도록 MSIL 코드 즉 *.il를 생성한다. [그림 3]은 Oolong 파일이 번역기 시스템에 입력으로 들어가 출력으로 MSIL 코드가 나오면 그것을 다시 MSIL IL 어셈블러를 이용해 실행이 되는 과정을 나타낸 것이다.

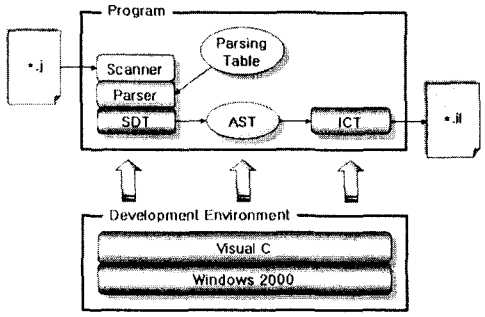


[그림3] Bytecode-to-MSIL 번역기 시스템

3.2 Bytecode-to-MSIL 번역기

본 번역기 시스템은 윈도우즈 2000 환경 하에서 Visual C를 이용하여 구현하였으며, 스캐너, 파서, SDT(Syntax Directed Translation), AST (Abstract Syntax Tree) 그리고 ICT(Intermediate Code Translator)로 구성된다.

[그림4]는 시스템의 구조를 나타낸 것이다.



[그림4] 번역기 시스템 구성도

첫 번째 단계로 번역기를 정형화한 형태로 구성하기 위해 Oolong 코드의 명령어 집합을 context -free 문법(Oolong.g)을 이용하여 고안하였다. 두 번째 단계로 고안된 문법(Oolong.g)을 가지고 파서 생성기(PGS)를 이용하여 문법에 대한 어휘 정보와 파싱 테이블을 얻는다. 여기서 얻어진 어휘 정보를 이용하여 스캐너를 작성하고 파싱 테이블을 이용하여 파서를 구성하였다. 세 번째 단계로 입력 문법의 구조에 따라 그 생성 규칙에 대한 의미 수행 코드를 작성하여 AST를 생성하고 이 AST를 탐색하면서 입력 코드인 Oolong 코드와 의미적으로 동등한 MSIL 코드를 생성하는 ICT를 작성하였다.

결과적으로 자바 언어를 사용하여 작성된 프로그램이 .NET 플랫폼에서 실행 가능하게 된다.

다음 [표1]과 [표2]는 Oolong 코드와 MSIL 코드를 비교 할 수 있도록 매칭시킨 테이블이다. ICT에서는 이것을 바탕으로 코드 변환을 한다.

Oolong	MSIL
String	string
...	...
byte	int8
short	int16
int	int32
long	int64
...	...
float	float32
double	float64

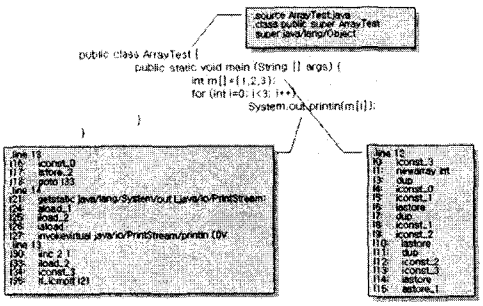
[표1] 데이터 타입 매핑표

Oolong	MSIL
nop	nop
...	...
iload_num	ldloc.num
istore_num	stloc.num
iadd, isub	add, sub
imul, idiv	mul, div
...	...
return	ret
...	...

[표2] 명령어 집합 매핑표

3.3 실험 결과 및 분석

다음은 자바로 작성된 배열 프로그램으로부터 Oolong 코드를 추출한 후에 MSIL로 변환하여 .NET 환경에서 실행하는 예제이다.



[예제1] 자바프로그램 및 추출된 Oolong 코드

```
Nonterminal: PROGRAM
    Nonterminal: HEAD_PART
        Nonterminal: PSEUDO_SOURCE
            Nonterminal: FILE_NAME
                Terminal: ArrayTest
                ...
    Nonterminal: METHOD_PART
        Nonterminal: METHOD_START
            ...
                    Nonterminal: PSEUDO_LINE
                        Terminal: 16
                    Nonterminal: SOURCE
                        Nonterminal: LABEL
                            Nonterminal: LABEL_NAME
                                Terminal: I38
                        Nonterminal: RETURN
                Nonterminal: METHOD_END
```

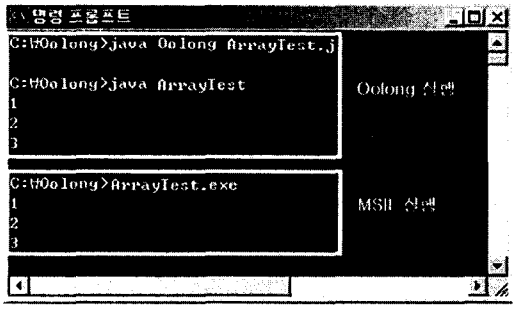
[예제2] Oolong 코드의 AST

```
.assembly extern mscorlib {
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89)
  .ver 1:0:3300:0
}
.assembly ArrayTest {
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.class public auto ansi beforefieldinit ArrayTest
                extends [mscorlib]System.Object {
  .method public hidebysig specialname rtspecialname
          instance void .ctor() cil managed {
    ....
  }
  .method public hidebysig static void Main() cil managed {
    .entrypoint
    .maxstack 4
    .locals init (int32 V_0, int32[] V_1, int32 V_2)
    I0:  ldc.i4.3
    I1:  newarr      [mscorlib]System.Int32
    I3:  stloc.1
```

```
        ldloc.1
l4:     ldc.i4.0
l5:     ldc.i4.1
l6:     stelem.i4
l7:     ldloc.1
l8:     ldc.i4.1
l9:     ldc.i4.2
l10:    stelem.i4
l11:    ldloc.1
l12:    ldc.i4.2
l13:    ldc.i4.3
l14:    stelem.i4
l15:    nop
l16:    ldc.i4.0
l17:    stloc.2
l18:    br          l33
l21:    nop
l24:    ldloc.1
l25:    ldloc.2
l26:    ldelem.i4
l27:    call        void [mscorlib]System.Console::WriteLine(int32)
l30:    ldloc.2
        ldc.i4.1
        add
        stloc.2
l33:    ldloc.2
l34:    ldc.i4.3
l35:    blt         l21
l38:    ret
    }
}
```

[예제3] 생성된 MSIL 코드

[그림5]는 번역기를 통해 생성된 MSIL 코드를 실행파일로 변환해서 실행한 결과를 자바 프로그램의 출력결과와 비교한 화면이다.



[그림5] 실행 화면 비교

4. 결론

자바로 작성된 프로그램은 JVM 플랫폼에서는 실행이 되지만 .NET 플랫폼에서 실행이 되지 않고, 반대로 C#과 같은 .NET 언어로 작성된 프로그램은 .NET 플랫폼에서는 실행이 되지만 JVM 플랫폼에서 실행이 되지 않는다. 이런 이유로 본 논문에서는 자바소스를 컴파일하여 생성된 클래스 파일에서 Oolong 코드를 생성하고 생성된 Oolong 코드를 .NET의 MSIL 코드로 변환하여 자바로 구현된 프로그램이 .NET 환경에서 실행되도록 하는 Bytecode -to-MSIL 번역기 시스템을 구현하였다. 따라서, 자바 프로그래머는 JVM이나 .NET 플랫폼 환경에 관계없이 프로그램을 작성하여 실행시킬 수 있다.

번역기 시스템의 구현을 정형화하기 위해 Oolong 코드의 명령어들을 문법으로 작성하였으며, PGS를 통해 생성된 어휘 정보를 가지고 스캐너를 구성하였고, 파싱테이블을 가지고 파서를 설계하였다. 파서의 출력으로 AST가 생성되면 번역기는 AST를 탐색하면서 의미적으로 동등한 MSIL 코드를 생성하도록 시스템을 컴파일러 기법을 이용하여 모듈별로 구성하였다.

앞으로 자바 플랫폼 프로그래밍 언어에서 지원하는 기능 및 모듈들을 .NET 플랫폼 환경에서도 동일하게 사용할 수 있도록 번역기를 확장하고, Oolong 코드와 MSIL 코드 자체를 분석하여 실험을 통해 보다 나은 코드를 낼 수 있는 코드 최적화를 위한 연구를 수행할 예정이다.

참고문헌

[1] Andrew Troelsen, C# and the .NET Platform, 2nd ed, APRESS, 2001.

[2] Bill Venners, Inside the JAVA Virtual Machine, 2nd ed., McGraw-Hill, 2000.

[3] Don Box & Chris Sells, Essential .NET Volume 1 The Common Language Runtime, Addison Wesley, 2002.

[4] Arthur Gittleman, Computing With C# and the .Net Framework, Jones & Bartlett Publishers, 2003.

[5] Hoshua Engel, Programming for the Java Virtual Machine, Addison Wesley, 1999.

[6] Jeff Prosise, Programming Microsoft .NET, Microsoft Press, 2002.

[7] John Gough, Compiling for the .NET Common Language Runtime(CLR), Prentice Hall, 2002.

[8] Ken Arnold, James Gosling & David Holmes, The Javatm Programming Language, 3rd ed., Addison Wesley, 2000.

[9] Microsoft Corporation, Common Language Infrastructure(CLI), Dec. 2001.

[10] Serge Lindin, Inside Microsoft .NET IL Assembler, Microsoft Press, 2002.

[11] Troy Downing & Jon Meyer, Java Virtual Machine, O'REILLY, 1997.

[12] Tim Lindholm & Frank Yellin, The Java™ Virtual Machine Specification, 2nd ed, Addison Wesley, 1999.

[13] Tom Archer, INSIDE C#, 2nd ed, 정보문화사, 2002.

[14] 오세만, 컴파일러 입문, 정익사, 2000.

[15] 오세만 & 이양선 & 김상훈 & 고광만, 자바 입문, 생능출판사, 1998.