C2JNI : 내장 C 언어에서 JNI 코드를 생성하는 변환기

(C2JNI : An Embedded C to JNI Translator)

유 재 우 † 최 종 명 †† 김 영 철 †††
(Chae-Woo Yoo) (Jong-Myung Choi) (Young-Chul Kim)

요 약 자바는 플랫폼에 독립적인 객체지향 프로그래밍 언어로서 널리 사용되고 있지만, 플랫폼에 의존적인 기능을 사용해야 하거나 혹은 유산 시스템(legacy system)을 재사용하기 위해서는 JNI를 이용해야 한다. JNI는 자바가 C/C++ 언어와 결합하기 위한 표준화된 규칙과 API를 제공하지만, 개발자가 JNI를 이용해서 프로그램을 개발하는 것은 매우 복잡하고, 번거롭다는 단점이 있다. 이러한 문제를 해결하기 위해서 본 논문에서는 자바 프로그램에 내장된 C 프로그램 코드를 자동적으로 JNI 명세에 맞는 C 언어로 변환할 수 있는 C2JNI라는 변환기를 소개한다. C2JNI를 사용하는 경우에 내장된 C 언어 프로그램은 JNI API를 사용하지 않고서도 C 언어와 자바 프로그램을 결합할 수 있기 때문에 개발자는 자바와 C 언어에 대한 지식만 있으면, 자바와 C 언어를 결합해서 사용할 수 있다.

키워드 : JNI, 내장된 C 언어, 유산 시스템

Abstract Java, a platform independent object-oriented programming language, is widely used, however it should be integrated with JNI to use system services or to reuse legacy systems. Though JNI provides the standard APIs which allow Java to be combined with C/C++, it is very hard and cumbersome for developers to use JNI APIs. In order to address this problem, we introduce a translator named C2JNI, which converts the embedded C program into a JNI compatible C program. With C2JNI, developers can integrate Java and C programs without JNI APIs, and it will reduce the complexity caused by JNI APIs.

Key words : JNI, embedded C, legacy system

1. 서 론

자바 언어는 플랫폼에 독립적인 객체지향 언어로서 내장형 시스템에서부터 엔터프라이즈 컴퓨팅에 이르기까지 많은 분야에서 널리 사용되고 있다. 자바는 또한 JNI(Java Native Interface)[1]를 통해서 C 혹은 다른 언어와 결합되어서 사용될 수 있는 기능을 제공한다. JNI 명세는 자바 언어와 C 언어를 결합하기 위한 API와 데이타 타입들을 매핑시키기 위한 규칙들로 정의되어 있다. 따라서 자바와 C 언어를 결합하기 위해서는 JNI 명세에 맞는 C 언어 프로그램을 작성해야 한다. 그러나 개발자가 JNI 명세에 맞게 C 프로그램을 작성하는 것은 상당히 복잡하고, 번거로우며, 에러가 발생하기 쉽다는 문제점을 가지고 있다[2]. 따라서 보다 간단하면서도 쉽게 자바와 C 언어를 결합할 수 있는 방법이 필요하다.

자바와 C 언어를 결합하기 위해서 JNI를 사용하는 대신에 C 라이브러리를 위한 자바 래퍼 클래스(wrapper class)를 자동 혹은 반자동적으로 생성하는 방법들이 연구되었다[3-6]. 그러나 이 방법들은 몇 가지 문제점들을 가지고 있다. 첫 번째 문제점은 자바에서 C 라이브러리로 접근할 수 있는 방법은 제공하지만, C에서 자바의 기능을 접근할 수 있는 방법은 제공하지 못한다. 따라서 자바와 C 언어를 밀접하게 결합해서 작성해야 되는 응용프로그램 개발에는 적용하기 어렵다. 두 번째 문제점은 래핑 API들이 표준화되어 있지 않기 때문에 새로운 래핑 도구를 사용하는 경우에 응용프로그램들을 변경해야 하는 문제점이 있다.

· 본 연구는 숭실대학교 교내 연구비 지원으로 이루어졌음
† 종신회원 : 숭실대학교 컴퓨터학부 교수
cwyoo@ssu.ac.kr
†† 정 회 원 : 목포대학교 정보공학부 교수
jmchoi@mokpo.ac.kr
††† 비 회 원 : 명지전문대학 교수
yckim@ss.ssu.ac.kr
논문접수 : 2003년 9월 26일
심사완료 : 2004년 9월 3일

본 논문에서는 사용자가 자바와 C를 쉽게 결합해서 사용할 수 있도록 자바 프로그램에 C 언어 코드를 내장시키는 방법을 소개한다. 내장된 C 언어는 EmC라고 불리며, C의 문법[7]에 $ 혹은 $$ 접두어를 갖는 식별자를 추가한 것이다. EmC는 자바 클래스의 메소드를 구현하기 위해 사용되며, EmC로 작성된 메소드에서는 자바 클래스의 멤버 필드와 메소드를 자유롭게 접근해서 사용할 수 있다. 따라서 EmC를 사용하는 경우에 자바와 C 언어는 양방향으로 접근이 가능하며, 문법 체크와 의미 체크가 수행될 수 있다는 장점을 가지고 있다. 또한 이 언어를 사용하는 경우에 사용자는 JNI API 혹은 래핑 API에 대해 고려할 필요가 없기 때문에 쉽게 배우고, 사용할 수 있다.

EmC로 작성된 메소드는 Argos 시스템[8]에 포함된 C2JNI라는 컴파일러를 통해서 JNI 명세에 맞는 C 언어 소스로 변환된다. C2JNI는 EmC로 작성된 프로그램을 어휘 분석과 구문 분석을 통해 문법적인 오류를 체크한다. 또한 C2JNI는 Argos 파서로부터 전달받은 클래스의 멤버 필드와 메소드에 관련된 정보가 저장된 심볼 테이블을 이용해서 의미 분석을 수행하고, AST(Abstract Syntax Tree)를 순회하면서 구문 지향 변환 방법[9]을 통해서 타겟 코드를 생성한다.

EmC와 C2JNI를 사용하는 경우에 사용자는 C로 구현된 유산 시스템을 재활용함으로써 비교적 쉽게 객체 지향 시스템으로 변환할 수 있으며, 플랫폼에서 제공하는 기능을 사용하는 시스템을 상대적으로 쉽게 구현할 수 있다. 따라서 EmC와 C2JNI를 사용하는 경우에 사용자는 에러를 줄일 수 있고, 읽기 쉬운 코드를 생성할 수 있으며, 생산성을 높일 수 있다는 장점을 가지고 있다.

본 논문은 2장에서 관련 연구들을 소개하고, 3장에서 내장 언어인 EmC에 대해서 설명한다. 4장에서는 C2JNI 시스템의 구조와 작동 방법에 대해서 기술하고, 5장에서는 EmC 언어의 사용 예와 C2JNI에 의해 생성된 C 언어 코드를 소개한다. 6장에서는 결론 및 향후 연구를 밝힌다.

2. 관련 연구

자바와 C 언어의 결합은 C 언어가 사용되는 목적에 따라 크게 두 가지 형태로 구분할 수 있다. 첫째는 C 언어로 구축된 유산 시스템(legacy system)을 재사용하기 위해서 자바와 C를 결합하는 것이고, 두 번째는 플랫폼에 의존적인 기능에 접근하기 위해서 C 언어와 결합하는 형태이다.

첫 번째 형태의 결합은 주로 C 라이브러리를 래핑을 통해서 자바 언어에서 접근하는 방법이다. 이러한 형태의 대표적인 시스템으로는 JCI(Java-to-C Interface Generator)[3,4], JACAW(Java-C Automatic Wrapper)[5], SWIG[6] 등이 있다. JCI는 기존 C 라이브러리를 자바 언어와 자동적으로 바인딩시켜주는 인터페이스 생성 도구로서 C 언어의 헤더 파일을 입력으로 받고, C 스텁 함수와 자바의 네이티브 메소드 선언을 자동적으로 생성한다. JACAW는 JCI와 유사하게 C 라이브러리를 자동적으로 자바 코드로 래핑하는 도구이다. JACAW는 개별적인 C 루틴에 래핑을 적용할 수도 있고, 라이브러리 전체에 적용할 수도 있다. SWIG은 기존 C/C++ 언어로 작성된 라이브러리를 자바, Tcl/tk, Python 등의 고차원 스크립트 언어에서 접근할 수 있도록 지원한다. SWIG은 여러 언어를 지원할 수 있다는 장점을 가지고 있는데 비해서 인터페이스 파일을 사용자가 작성해야 한다는 단점을 가지고 있다.

첫 번째 형태의 시스템들은 자바에서 유산 시스템을 재사용하기 위한 용도로 개발되었기 때문에 이들 시스템의 주된 관점은 기존 C 라이브러리를 최대한 변경하지 않고, 자바 언어에서 재사용하는 것이다. 이러한 시스템들은 몇 가지 단점들을 가지고 있다. 첫째로 이들 시스템은 자바에서 C 언어로의 접근은 가능하지만, C 언어에서 자바 언어로의 접근은 제공되지 않기 때문에 자바와 C를 결합해서 새로운 시스템을 개발하는 경우에는 부적합하다. 둘째로 이 시스템들은 각자 자신만의 고유한 래핑 API를 사용하기 때문에 호환성이 떨어지는 문제점이 있다. 이에 반해 EmC와 C2JNI를 사용하는 경우에는 C 언어와 자바의 양방향에서 접근이 가능하며, C 언어 문법을 사용하고 API를 사용하지 않기 때문에 호환성이 높다는 장점을 가지고 있다.

두 번째 형태는 첫 번째 형태와는 달리 자바와 C 언어가 서로 접근해서 사용할 수 있어야 한다. 이러한 기능을 제공할 수 있는 시스템으로는 CNI(*Compiled* Native Interface)[2]가 있다. CNI는 GCC[10] 기반의 GCJ[11]에서 자바와 C++ 클래스가 서로 호환성을 갖도록 하는 API이다. CNI를 사용하는 경우에 C++ 코드는 자바 객체를 직접 접근할 수 있기 때문에 자바와 C++를 쉽게 결합해서 사용할 수 있다는 장점을 가지고 있다. 그러나 CNI는 GNU의 GCJ를 사용해야만 하고, 이 때 자바 클래스의 바이너리 호환성이 떨어지는 문제가 발생할 수 있다. 즉, 부모 클래스에 새로운 멤버 필드를 추가하는 경우에 모든 자식 클래스들도 새로 컴파일해야 한다. 또한 CNI는 인터페이스 타입의 메소드는 호출할 수 없는 문제점을 가지고 있다. 유연한 시스템의 개발을 위해서 인터페이스의 사용이 증가하는 현 상황에서 인터페이스를 지원하지 못하는 것은 심각한 단점이 될 수 있다. 이에 비해 Argos와 C2JNI는 순수 자바로

작성되어 있기 때문에 자바가 실행되는 모든 플랫폼에서 사용될 수 있으며, C2JNI가 생성하는 C 언어 코드 역시 JNI와 ISO 표준 C 언어를 지원하는 C 컴파일러에서 실행될 수 있기 때문에 CNI에 비해서 호환성이 뛰어나다. 또한 C2JNI는 순수 자바와 JNI를 기술을 사용하기 때문에 인터페이스 타입도 지원한다.

C2JNI는 기본적으로 EmC를 입력으로 받고 JNI를 사용하는 C 코드를 생성하기 때문에 EmC는 JNI가 지원하는 기능 이외에 새로운 기능을 제공하지는 않는다. 그러나 EmC를 사용하는 경우에 JNI가 제공하는 200개가 넘는 API에 대해서 고려하지 않고서도 자바와 C를 결합할 수 있다는 장점을 가지고 있다. JNI를 사용하는 경우에는 복잡한 API 때문에 프로그램을 작성하기 어렵고, 에러가 발생하기 쉽기 때문에 생산성이 떨어진다. 또한 복잡한 API는 프로그램을 읽기 어렵게 만들기 때문에 결과적으로 소프트웨어 유지 보수도 어려워진다는 문제점을 가지고 있다. 이에 비해 EmC를 사용하는 경우에는 JNI의 API를 사용하지 않기 때문에 상대적으로 프로그램을 작성하기 쉽고, 에러 발생률이 낮으며, 프로그램의 가독성이 높아지는 장점을 가지고 있다.

3. 내장된 C 언어

3.1 내장된 C 언어에서 고려할 사항

내장된 언어(embedded language)는 호스트 언어(host language)에 포함되어서 사용되며, 호스트 언어와 밀접하게 연결되어서 문제를 해결한다. 내장된 언어는 API를 사용해서 것에 비해 이해하기 쉽고, 사용하기 쉽다. 또한 내장된 언어를 사용하는 경우에 개발자는 API와 데이타 변환에 관한 사항들을 고려하지 않아도 되기 때문에 복잡성이 줄고, 문제 자체에 집중할 수 있다. 내장된 언어는 응용프로그램 개발에서 특별한 목적을 위해서 주로 사용하며, 대표적인 것으로는 C 혹은 Java 언어에 내장되어서 사용되는 ESQL(Embedded SQL)과 웹 페이지에서 사용되는 자바 스크립트가 있다.

자바와 C를 결합하는 경우에 기존의 래핑 방법과 CNI의 문제점을 해결하기 위해서 본 논문에서는 자바 프로그램에 C 언어로 작성된 서브프로그램을 내장시키는 방법을 사용한다. 내장된 C 언어는 EmC라고 하며, 자바 클래스의 메소드를 정의하기 위해서 사용된다. EmC로 작성된 메소드는 자바 클래스에 소속되기 때문에 인지 과학적인 측면에서 프로그램을 작성하고, 이해하는데 좀더 효과적이다. 즉, EmC로 작성된 메소드에서 클래스의 메소드와 멤버 필드를 접근해야 할 때 개발자는 좀 더 쉽게 클래스의 메소드와 멤버 필드를 인지할 수 있게 된다. EmC는 ISO에서 정의한 표준 C 언어의 문법을 사용하면서, 자바 클래스의 멤버 필드와 메소드를 접근하기 위해 추가적인 문법 구조를 갖는다. 예를 들면, EmC로 작성한 메소드에서 자바 클래스의 멤버 필드와 메소드를 접근하기 위해서는 식별자 이름에 $ 접두어를 붙이고, 매개 변수를 접근하기 위해서는 $$ 접두어를 사용한다.

EmC를 이용해서 프로그램을 작성하는 경우에 유산 시스템의 라이브러리들 중에서 결합도가 높고, 연관성이 높은 것들을 모아서 새로운 형태의 클래스 혹은 컴포넌트로 만들어서 사용할 수 있다. 그림 1은 C 라이브러리 중에서 연관성이 많은 함수들을 자바 클래스의 메소드로 사용하는 것을 보여준다. 그림 1에서 큰 원은 자바 클래스이고, 원의 중간 부분은 메소드들을 의미하며, 내부 원은 자바 클래스의 멤버 필드들이다. 메소드들은 자바 언어 혹은 EmC 언어를 이용해서 작성할 수 있고, 클래스의 멤버 필드들을 자유롭게 접근할 수 있다.

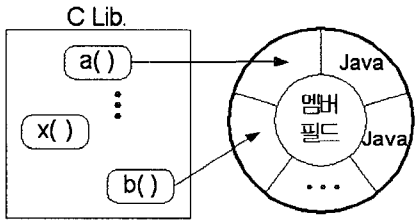

그림 1 C 라이브러리를 클래스로 재구성

EmC로 작성된 메소드를 JNI 명세에 맞는 C 언어 프로그램으로 변경하는 것은 얼핏 보기에는 $ 혹은 $$로 시작하는 식별자를 매크로를 이용해서 JNI 명세에 맞는 이름으로 변경하는 것으로 생각할 수 있다. 그러나 EmC로 작성된 프로그램을 JNI 명세에 맞는 C 언어 프로그램으로 변경하기 위해서는 몇 가지 어려움들이 존재한다.

첫 번째 어려움은 EmC에서 멤버 필드를 접근할 때 동일한 이름이라도 사용되는 위치에 따라 다른 형태의 코드로 변환해야 한다는 것이다. 예를 들어, EmC에서는 멤버 필드의 값을 읽을 때와 값을 할당할 때 동일한 방법으로 이름을 이용해서 기술한다. 즉, 정수형의 count라는 멤버 필드의 값을 하나 증가시키기 위해서는 $count = $count + 1 이라는 문장을 사용할 수 있다. 그러나 JNI 명세를 따르는 경우에 멤버 필드의 값을 읽기 위한 함수와 값을 할당하기 위한 함수가 다르기 때문에 동일한 $count라도 위치에 따라서 변경되는 코드가 달라진다. count 멤버 필드의 값을 읽기 위해서는 JNIEnv 레코드의 GetIntField() 함수를 이용해야 하고, count의 값을 할당하기 위해서는 SetIntField() 함수를 이용해야 한다. 설상가상으로 C 언어에서 변수의 값을

변경하는 할당 연산은 식(expression)으로 정의되어 있기 때문에, 거의 모든 C 문장에서 변수 값을 가져오거나 변경할 수 있다는 점이다. 예를 들어, 다음과 같은 문장이 존재하는 경우에 멤버 필드와 매개 변수에 해당되는 내용들을 모두 JNI API에 따라 변환해야 한다.

for(i = 0; i < $len; $len = $len + $$diff, i++) { .. }

이러한 문제 때문에 EmC 언어의 내용을 처리하기 위해서는 C 언어의 문법을 파싱하고, 의미에 맞게 코드를 생성할 수 있는 컴파일러가 필요하다.

두 번째는 JNI를 사용할 때 클래스의 멤버 필드와 메소드 이름들이 변경된다는 점이다. 특히 메소드의 경우에 자바는 메소드 오버로딩을 지원하는데 비해서 C 언어는 메소드 오버로딩을 지원하지 않기 때문에 자바의 메소드 이름을 JNI 규칙에 맞게 변경해야 한다. 또 다른 문제점은 자바에서는 멤버 필드와 메소드의 이름으로 유니코드를 사용할 수 있는데 비해, C 언어에서는 유니코드를 사용할 수 없다는 점이다. 따라서 오버로딩과 유니코드가 사용되는 경우에 JNI 규칙에 맞게 메소드 이름을 변경해주어야 한다.

세 번째 어려움은 데이타 타입의 불일치이다. 자바와 C 언어는 데이타 타입이 다르기 때문에 JNI에서는 C 언어와 자바가 결합해서 사용될 수 있는 새로운 형태의 데이타 타입들을 정의하고 있다. 따라서 내장된 C 언어 코드를 JNI 명세에 맞는 C 프로그램으로 변경하기 위해서는 데이타 타입들을 변경해야 한다.

3.2 EmC 언어

EmC는 자바 클래스의 메소드를 구현하기 위해서 사용되며, EmC를 지원하는 자바 프로그램은 표 1과 같은 형태의 메소드를 지원한다. 표 1에서 []로 묶인 부분은 생략할 수 있다. <!ns="C">는 현재 정의하는 메소드가 EmC로 구현되어 있다는 것을 명시하기 위해서 사용된다.

표 1 메소드 선언 형태

[*method-modifier*] **<!ns="C">** *returnType methodName* (*arglist*) { *body* }

EmC와 자바 언어가 서로 협력하기 위해서는 공통적인 데이타 타입을 사용하여야 한다. JNI 명세는 자바와 C 언어를 결합하는 경우에 필요한 데이타 타입 매핑에 대해서 정의하고 있다. 다음 표 2는 자바의 네이티브 메소드 선언과 구현된 C 언어 함수 헤더 사이의 매핑을 보여준다. 자바에서 getLine() 메소드의 매개 변수와 리턴 타입은 String 타입이고, C 언어에서는 String에 매핑되는 jstring 타입을 사용한다.

EmC 언어는 C 언어의 문법을 그대로 사용하지만 특성상 약간의 추가적인 구조들을 갖는다. 표 3은 내장된 C 언어에서 추가된 문법 구조들을 보여준다. 표 3에서 밑줄 친 부분들이 새로 추가된 문법 구조이다.

표 3 EmC 언어에서 변경된 문법

<block> ::= <pragma_list>? <declaration_list>? <statement_list>? <pragma_list> ::= ('@' 'pragma' <pragma_code>)+ <pragma_code> ::= (*any character except newline*)+ <expression> ::= <lvalue> | <assignment> | ... <assignment> ::= <lvalue> <assignment_op> <expression> <lvalue> ::= | '$$' <identifier> /* 매개 변수 */ | '$' <identifier> /* 클래스 멤버 필드와 메소드 */ ...

첫째로 pragma는 EmC 언어에서 사용되는 헤더 파일 인클루드와 같이 C 프로그램에서 기본적으로 사용되는 코드들을 기술하기 위해서 사용된다. pragma에 기술된 내용들은 C2JNI에 의해 @pragma 키워드는 삭제되고, 내용만 JNI C 프로그램의 처음에 복제된다. 표 4는 @pragma의 사용법을 보여준다.

표 4 pragma 사용

EmC 언어 코드	JNI 명세에 맞는 C 언어 코드
@pragma #include <stdio.h>	#include <stdio.h>

두 번째로 변경된 사항은 lvalue에서 $ 문자로 시작되는 멤버 필드와 매개 변수를 위한 식별자가 추가된 것이다. lvalue에서 멤버 필드와 매개 변수는 할당 연산자에서 사용되는 위치에 따라 다른 형태의 코드를 생성하게 된다. 즉, 자바의 멤버 필드가 L-value로 사용될 때와 R-value로 사용될 때 JNI의 함수들이 달라진다. 예를 들어, EmC에서 자바 클래스의 boolean 멤버 필드의 내용을 읽기위해서는 GetBooleanField() 함수를 사용하여야 하고, boolean 멤버 필드의 내용을 변경하기

표 2 자바와 C 데이타 타입

자바 메소드 선언	JNI 명세에 맞는 C 언어 코드
native String getLine(String prompt)	... jstring ...getLine(..., jstring prompt)

표 5 멤버 필드 접근

EmC 언어 코드	JNI 명세에 맞는 C 언어 코드

printf("%d₩n", $value);	jclass cls = (*env)->GetObjectClass(env, obj);
	jfieldID fid = (*env)->GetFieldID(env, cls, "value", "I");
	printf("%d₩n", (*env)->GetIntField(env, obj, fid));

위해서는 SetBooleanField() 함수를 사용하여야 한다. JNI 명세는 멤버 필드와 매개 변수를 접근하기 위한 메소드들을 정의하고 있다.

표 5는 내장된 C 언어에서 자바 클래스의 멤버 필드를 접근하는 방법을 보여준다. 내장된 C 언어에서는 $ 문자와 멤버 필드 이름을 이용해서 접근할 수 있지만, JNI 명세를 따르는 경우에는 JNIEnv의 포인터 타입을 이용해야 한다.

4. C2JNI 시스템

4.1 C2JNI 시스템 구성

EmC 언어는 Argos 컴파일러와 C2JNI에 의해서 그림 2와 같은 처리 단계를 거치게 된다. 그림에서 회색으로 채워진 사각형은 기존에 존재하는 도구를 의미한다. EmC 언어 코드를 포함한 응용프로그램은 Argos 파서에 의해서 분리되어서 C2JNI에 전달된다. Argos 파서는 응용프로그램을 컴파일하면서, 프로그램의 클래스에서 정의된 멤버 필드에 대한 정보들을 추출해서 심볼 테이블에 저장한다. C2JNI는 전달받은 C 언어 코드를 파싱해서 JNI 코드를 생성한다. C 언어를 파싱하기 위해서 C2JNI는 LL(k) 파싱 컴파일러 생성기인 JavaCC [12]를 이용해서 구현되었다.

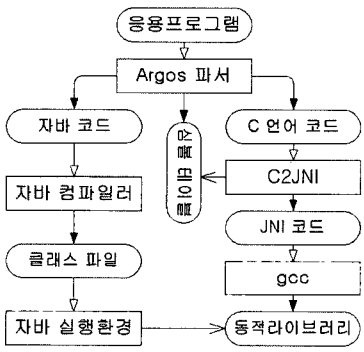

그림 2 내장된 C 언어가 처리되는 과정

C2JNI는 기본적으로 EmC 언어의 문법을 검사하고, 의미에 따라 EmC 언어 코드를 JNI 명세를 따르는 코드로 변환하게 된다. 그림 3은 C2JNI가 EmC 언어 코드를 JNI 코드로 변환하는 과정을 보여준다.

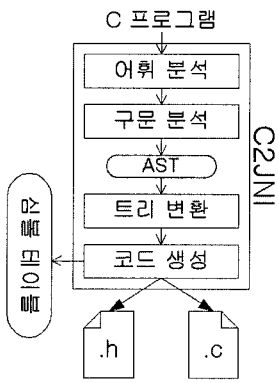

그림 3 C2JNI의 처리 단계

C2JNI는 입력으로 전달받은 프로그램을 어휘 분석과 구문 분석을 통해서 AST로 표현한다. 이때 EmC 언어로 작성된 프로그램에 문법적인 에러가 있는 경우에는 구문 분석을 수행하는 과정에서 에러를 발견할 수 있다. 프로그램에 문법적인 에러가 없는 경우에 구문 분석 단계에서 AST를 생성한다. AST의 노드들은 INode 인터페이스를 구현한 클래스의 인스턴스들이다. AST로 표현된 프로그램은 트리 변환 단계에서 의미 분석을 수행한다. 즉, EmC 프로그램에서 접근하는 Argos에서 작성한 심볼 테이블을 이용해서 EmC 프로그램에서 사용하는 자바 클래스의 멤버 필드와 메소드에 대한 타입 체킹을 수행한다. 의미 분석이 올바르게 수행된 경우에 AST를 타겟 코드를 생성하기 쉬운 형태로 변형된다. 최종적으로 코드 생성 단계에서는 변형된 AST를 순회하면서 타겟 코드를 생성한다.

EmC로 작성된 프로그램에 에러가 있는 경우에 C2JNI는 2가지 방법으로 처리합니다. 첫 번째 EmC 언어의 문법적인 C2JNI의 파서 부분에서 처리합니다. C2JNI에 포함된 파서는 EmC 언어로 작성된 프로그램을 문법적으로 에러가 있는지 여부를 파악합니다. 둘째로 EmC에서 사용되는 자바 클래스의 식별자들은 Argos 컴파일러에 의해 심볼 테이블에 관리되기 때문에 의미 에러는 의미 분석 과정에서 파악합니다.

4.2 타겟 코드 생성

EmC 언어 프로그램은 문장 단위로 구성되어 있고, 각 문장은 2가지 형태로 분류될 수 있다. 하나는 순수 C 언어로 구성된 문장이고, 다른 하나는 EmC 언어에서

자바 클래스, 메소드, 혹은 매개 변수의 내용에 접근하거나 내용을 변경하는 문장이다. 첫 번째 형태의 문장을 클린(clean) 문장이라고 하고, 두 번째 형태를 더티(dirty) 문장이라고 부른다. 클린 문장은 순수 C 언어로 구성된 문장이기 때문에 내용이 변경되지 않고 그대로 타겟 코드에 복사된다. 반면에 더티 문장은 JNI 명세에 따르는 C 언어 코드로 적절히 변경되어야 한다. C2JNI는 구문 분석 단계를 거치면서 각 문장 내에 $ 문자가 존재하는지 여부에 따라, 클린 문장과 더티 문장을 구분하여 관리한다. 구문 분석 단계를 거치면 원본 C 언어 프로그램이 추상 구문 트리로 표현된다. 클린 문장은 문장 단위의 소스 코드 내용을 그대로 포함하는 CLEAN_NODE라는 하나의 노드로 표현되지만, 더티 문장은 변경되어야 하기 때문에 문법에 따라 추상 구문 트리 내에 여러 노드들을 형성하게 된다.

그림 4는 AST 트리의 노드를 표현하기 위해 사용되는 클래스들의 관계를 보여준다. AST의 노드는 INode라는 추상 클래스로부터 상속받는 클래스의 인스턴스이다. INode는 노드 타입을 기술하기 위한 type 멤버 필드와 노드가 자바 클래스의 멤버 필드를 접근하는지 여부를 알아보는 isMemberField()라는 메소드를 갖는다. 또한 unparse()는 AST 노드를 프로그램 코드로 변환하는 역할을 수행하는 메소드이다.

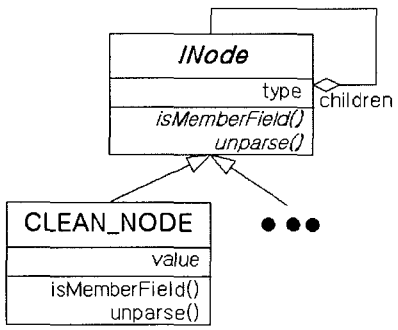

그림 4 추상 구문 트리 노드의 클래스도

EmC 언어를 JNI 명세에 맞는 C 언어로 변환하기 위해서 C2JNI는 C 언어를 파싱하고, AST를 관리한다. 멤버 필드가 RHS에 사용되는 경우에 AST는 그림 5와 같은 형태로 변경된다. 즉, AST에서 멤버 필드에 해당되는 노드가 나타나는 경우에 Get<type>Field() 함수를 호출하는 노드로 교체된다.

멤버 필드의 값을 변경해야 하는 할당이 존재하는 경우에 AST는 그림 6과 같은 형태로 변경된다. 즉, 할당에 해당되는 AST 노드는 멤버 필드의 값을 변경하는 Set<type>Field() 함수를 호출하는 노드로 변경된다.

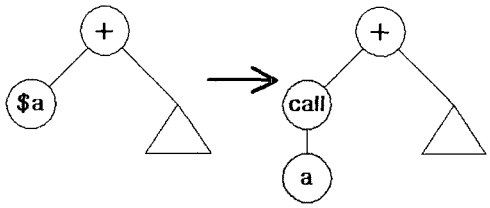

그림 5 멤버 필드가 RHS에 사용된 경우

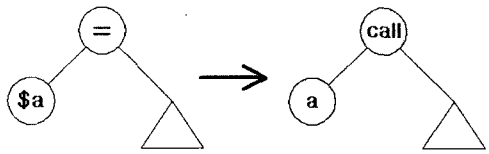

그림 6 멤버 필드가 LHS에 사용된 경우

멤버 필드가 배열인 경우에는 좀더 복잡하다. 배열 타입인 멤버 필드는 EmC로 작성된 메소드에서는 배열 내용을 복사해서 얻어온다. 따라서 자바 런타임에서 관리되는 배열 멤버 필드와 JNI에 존재하는 배열과 내용이 일치하기 위해서는 EmC 메소드에서 배열 내용을 변경한 다음에는 배열 내용을 다시 자바 런타임에 전달해야 한다. 배열 내용이 변경될 수 있는 것은 할당문뿐만 아니라 다른 C 언어의 매개 변수로 전달되는 경우에도 발생할 수 있다. 따라서 내장된 C 언어에서 매개 변수로 배열 타입의 멤버 필드가 전달되는 경우에 함수 호출이 끝난 다음에는 배열 내용을 일치시키기 위한 함수를 호출해야 한다. 그림 7은 함수의 매개 변수로 배열 타입의 멤버 필드 a가 전달되는 경우에 AST가 변경되는 것을 보여준다. 함수가 호출된 다음에 배열 값을 일치시키기 위한 함수를 호출하는 노드가 추가된다.

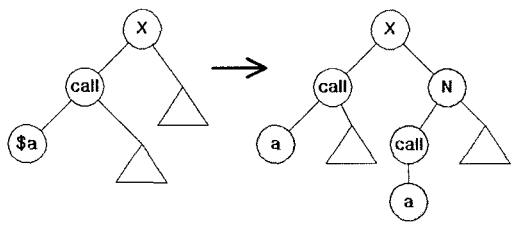

그림 7 배열 멤버 필드가 함수의 매개 변수로 사용된 경우

트리 변환 단계는 추상 구문 트리를 입력으로 받아서 더티 문장들의 내용들을 JNI 명세에 맞게 추상 구문 트리를 변경한다. 추상 구문 트리의 변경은 다음의 트리_변환 알고리즘에 따라 이루어진다.

<알고리즘. 트리_변환>

```
매개변수: INode n
BEGIN
    foreach c in n.children
        트리_변환(c)
    end-foreach
    if ( n.type == ASSIGN_NODE ) then
        if ( n.children[0].isMemberField() == TRUE ) then
            if( n.children[0].type != ARRAY_TYPE ) then
                심볼 테이블에서 n.name에 해당되는 멤버 필드의
                타입을 알아본다.
                할당문 노드를 Set<type>Field() 함수를 호출하는 노
                드로 변경한다.
            else
                n의 부모 노드에 Next라는 노드를 추가하고, Next의
                자식 노드에 배열 내용을 플러시하는 함수 호출 노
                드를 붙인다.
            end-if
        end-if
    else if ( n.type == RHS_ID_NODE ) then
        if ( n.isMemberField() == TRUE ) then
            심볼 테이블에서 n.name에 해당되는 멤버 필드의 타입
            을 알아본다.
            RHS_ID_NODE를 멤버 필드의 값을 얻어오는
            Get<type>Field() 함수를 호출하는 노드로 변경한다.
        end-if
    else if ( n.type == CALL_NODE ) then
        foreach c in n.parameter
            if ( c.isMemberField() == TRUE ) then
                if ( c.type != ARRAY_TYPE ) then
                    심볼 테이블에서 n.name에 해당되는 멤버 필드
                    의 타입을 알아본다.
                    RHS_ID_NODE를 멤버 필드의 값을 얻어오는
                    Get<type>Field() 함수를 호출하는 노드로 변경
                    한다.
                else
                    n의 부모 노드에 Next라는 노드를 추가하고,
                    Next의 자식 노드에 배열 내용을 플러시하는
                    함수 호출 노드를 붙인다.
                end-if
            end-if
        end-foreach
    else if ( n.type == DOUBLE_PLUS_NODE
            || n.type == DOUBLE_MINUS_NODE ) then
        if ( n.isMemberField() == TRUE ) then
            심볼 테이블에서 n.name에 해당되는 멤버 필드의 타입
            을 알아본다.
            n을 Set<type>Field() 함수를 호출하는 노드 m 으로 교체
            한다.
            m의 자식 노드에 PLUS_NODE 혹은 MINUS_NODE를
            추가한다.
            PLUS_NODE 혹은 MINUS_NODE의 자식 노드로 멤버
            필드의 값을 읽는 Get<type>Field() 함수를 호출하는 노
            드와 숫자 1을 표현하는 노드를 추가한다.
        end-if
    end-if
END
```

코드 생성 단계에서는 추상 구문 트리를 순회하면서 JNI 명세에 맞는 코드를 생성한다. 첫째로 노드가 CLEAN_NODE은 경우에는 노드의 value 값을 결과 코드로 복사한다. CLEAN_NODE가 아닌 경우에는 각 노드의 unparse() 메소드를 호출해서 추상 구문 트리의 노드를 소스 코드로 변환하도록 한다.

5. EmC 언어 사용 예

C 언어는 오랫동안 사용되어 왔고, 많은 분야에서 사용되었기 때문에 상당히 많은 소프트웨어 라이브러리들이 구축되어 있다. EmC 언어를 사용하는 경우에 기존의 C 언어로 작성된 라이브러리들은 비교적 간단하게 재사용할 수 있다. 예를 들어, 기존에 사용자가 quicksort라는 함수를 동적 라이브러리로 작성한 경우에 EmC 언어에서는 quicksort 함수를 일반 함수 호출과 동일한 방법으로 재사용할 수 있다. 표 6은 EmC 언어를 이용해서 매개 변수로 전달된 배열의 내용을 하나씩 증가시키고, quicksort 함수를 이용해서 정렬하는 것을 보여준다. quicksort 라이브러리를 사용하기 위해서 pragma를 이용해서 "quicksort.h" 헤더 파일을 인클루드한다. quicksort 함수를 호출하는 것은 일반 C 언어에서 사용하는 것과 동일한 방법으로 이루어진다.

표 6 Argos 언어에서 C 플러그인 언어

```
public class TestNative {
...
    public <ns="C"> void useSort(int a[]) {
        @pragma #include "quicksort.h"
        int i;
        for (i = 0; i < $$a.length; i++) {
            $$a[i] = $$a[i] + 1;
        }
        quicksort($$a, $$a.length);
    }
...
```

표 6의 EmC 언어를 처리하는 경우에 C2JNI는 C 언어 부분을 JNI의 명세에 맞는 별도의 C 언어 프로그램을 생성한다. C2JNI는 Argos 프로그램에서 C 언어에 대한 내용은 삭제하고, 네이티브 메소드 선언에 관한 코드를 기술한다. 표 7은 Argos 프로그램에 나타나는 네이티브 메소드 선언 코드이다.

표 7 C2JNI에 의해 생성된 코드 I

```
public class TestNative {
    public native void useSort(int a[]);
...
```

C2JNI는 표 6의 코드에서 C 언어 부분을 처리해서 JNI 명세에 맞는 표 8과 같은 코드를 생성한다.

표 8 C2JNI에 의해 생성된 코드 II

```
JJNIEXPORT void JNICALL Java_Test_useSort(JNIEnv
*ARGOS_env,
                jobject ARGOS_obj, jintArray data) {
...
   jint *ARGOS_elements_data;
   int i;
   for (i = 0; i < (*ARGOS_env)->GetArrayLength
   (ARGOS_env, data); i++) {
      ARGOS_SetIntArrayElement(ARGOS_env, data,
            I, ARGOS_GetIntArrayElement(ARGOS_env, data,
            i) + 1);
...
      quicksort(ARGOS_elements_data,
            (*ARGOS_env)->GetArrayLength(ARGOS_env,
            data));
...
```

C2JNI를 통해서 생성된 코드는 개발자가 직접 작성한 JNI 프로그램과 거의 유사한 성능을 갖는다. 표 9는 EmC를 이용한 표 6의 코드와 개발자가 직접 작성한 JNI 프로그램의 성능을 비교한 것을 보여준다. 표에서 JNI는 개발자가 직접 작성한 JNI 프로그램을 의미하고, C2JNI는 EmC로 작성되고 C2JNI를 통해서 생성된 프로그램을 의미한다. 성능 비교를 위해서는 퀵 소트의 데이타 개수를 변경하면서 수행 시간(단위: 1/1000 초)을 측정하였다. 테스트는 800MHz의 펜티엄 3 CPU, 256MB의 메모리를 가진 컴퓨터에서 GCC 3.2와 J2SDK 1.4.1에서 수행하였다.

표 9 수행 시간 성능 비교 (단위: 1/1000 초)

방법 \ 개수	1000	10000	20000	50000	100000
JNI	1	6	12	35	75
C2JNI	1	6	12	35	75

개발자가 작성한 JNI 프로그램과 C2JNI를 통해서 생성된 코드의 성능이 거의 동일한 것에 비해 개발자의 노력은 상대적으로 많은 차이가 발생한다. 표 10은 EmC로 작성한 코드와 JNI를 이용한 코드 사이의 복잡도와 프로그래밍 효율성을 여러 가지 측면에서 비교한 것이다. 표에서 Volume, Difficulty, Effort는 Halstead [13] 방법을 이용해서 측정한 것이다.

표 10에서 볼 수 있듯이 EmC를 이용하는 대신에 JNI를 사용하는 경우에 프로그램의 길이는 3배 정도 늘어나고, 프로그래밍을 위한 노력은 3.7배 정도 더 소모된다. 따라서 EmC를 사용하는 경우에 JNI를 사용하는 것에 비해 상대적으로 쉽고, 효과적으로 소프트웨어를 작성할 수 있는 장점을 가지고 있다.

표 10 EmC와 JNI 코드의 복잡도 비교

요소 \ 방법	EmC	JNI	비고
LOC	6	11	1.8배
Volume	130	394	3배
Difficulty	18	22	1.2배
Effort	2331	8697	3.7배

6. 결 론

자바 언어는 플랫폼 독립성이라는 장점을 바탕으로 많은 영역에서 사용되고 있다. 자바에서 플랫폼에 종속적인 기능을 사용하기 위해서는 JNI 명세에 따라 C 혹은 C++ 언어를 이용해서 프로그램을 작성해야 한다. 그러나 JNI 명세에 따르는 프로그램을 작성하는 것은 상당히 복잡하고, 번거롭다는 문제점이 있다.

JNI의 API를 사용할 때 발생하는 문제점을 해결하기 위해서 본 논문에서는 자바 프로그램에 C 언어 코드를 내장시키는 방법과 내장된 C 언어 코드를 JNI 명세에 맞는 코드로 변환할 수 있는 C2JNI라는 변환기를 소개하였다. C2JNI는 내장된 C 언어 프로그램을 파싱한다. 이때 내장된 C 언어에서 자바 클래스의 멤버 필드, 매개 변수, 메소드에 접근하는 코드는 문법에 맞게 AST를 구성하고, 그렇지 않은 경우에는 문장을 CLEAN_NODE라는 AST의 노드로 표현한다. 생성된 AST는 트리 변환 단계에서 자바 클래스의 멤버를 접근하는 노드를 JNI 명세에 맞는 함수 호출 노드로 변환한다. 변환된 AST는 코드 생성 단계를 거치면서 JNI 명세에 맞는 C 언어 코드를 생성한다.

C2JNI를 사용하는 경우에 JNI를 사용하는 것에 비해 보다 쉽고, 간단하게 자바와 C 언어를 결합할 수 있는 장점을 가지고 있다. 또한 개발자가 JNI 명세를 모르면서도 자바와 C 언어를 결합할 수 있으며, 에러를 줄일 수 있다는 장점을 가지고 있다. 따라서 내장된 C 언어를 사용하는 경우에 프로그래머는 생산성을 높일 수 있다.

참 고 문 헌

[1] Sheng Liang, *Java Native Interface: Programmer's Guide and Specification*, Addison-Wesley, 1999.

[2] Per Bothner and Tom Tromey, "Java/C++ integration: Writing native Java methods in natural C++", available at http://www.bothner.com/~per/papers/UsenixJVM01/CNI01.pdf.

[3] Sava Mintchev and Vladimir Getov, "Automatic

Binding of Native Scientific Libraries to Java," In *Scientific Computing in Object-Oriented Parallel Environments*, LNCS 1343, Springer-Verlag, pp. 129-136, 1997.
[4] Vlasimir Getov, Paul Gray, Sava Mintchev, and Vaidy Sunderam, "Multi-Language Programming Environments for High Performance Java Computing," In *Scientific Programming*, Vol. 7, No. 2, pp. 139-146, 1999.
[5] Yan Huang, Ian Taylor, David W. Walker, and Robert Davies, "Wrapping Legacy Codes for Grid-Based Applications," In *Proc. of IPDPS*, pp. 139-145, 2003.
[6] David M. Beazley, "SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++," In *Proc. of USENIX Tcl/Tk Workshop*, 1996, available at http://www.swig.org/.
[7] JavaCC Grammar Repository, available at http://www.cobase.cs.ucla.edu/pub/javacc/.
[8] 최종명, *Argos: 플러그인 지향 프로그래밍 언어*, 박사 학위 논문, 숭실대학교 컴퓨터학과, 2003.
[9] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, 1986.
[10] GNU Compiler Collection, available at http://gcc.gnu.org/.
[11] The GNU Compiler for the Java Programming Language, available at http://gcc.gnu.org/java/.
[12] JavaCC-The Java Parser Generator, available at https://javacc.dev.java.net/.
[13] Halstead and Maurice H., *Elements of Software Science*, Elsevier North-Holland, New York, 1977.

유 재 우
정보과학회논문지 : 소프트웨어 및 응용
제 31 권 제 1 호 참조

최 종 명
정보과학회논문지 : 소프트웨어 및 응용
제 31 권 제 2 호 참조

김 영 철
정보과학회논문지 : 소프트웨어 및 응용
제 31 권 제 1 호 참조