정적 프로그램 분석 과정을 단순화 하기 위한 지원 툴 설계

윤준호† 이병정‡ 우치수†

†서울대학교 컴퓨터공학부
{junoyoon, wuchisu}@selab.snu.ac.kr

‡서울시립대학교 컴퓨터과학부
bjlee@venus.uos.ac.kr

Design of a Supporting Tool for Simplifying Static Program Analysis Process

Junho Yoon† Byungjeong Lee‡ Chisu Wu†
†School of Computer Science and Engineering, Seoul National University
‡School of Computer Science, University of Seoul

요 약

정적 프로그램 분석기는 프로그램의 결점을 찾아내는 초기 목적에서 진화하여, 역공학, 재공학, 메트릭 검증 등 도구의 Front-End 모듈로 많이 쓰이고 있다. 이에 대한 다양한 연구가 진행되고 있고, 또 많은 관련 도구들이 소개되었지만, 사용자가 생각을 직접 코드형태로 구현하고, 결과를 모델형태로 표현하여, 시험하고 검증할 수 있도록 도와주는 프레임웍 수준 도구들은 거의 나와있지 않다. 본 논문에서는 정적 프로그램 분석을 쉽게 할 수 있도록 지원하는 기반 도구의 설계를 제안한다. 본 연구에서는 언어의 파싱과 시각화 과정의 반복되는 작업들을 추상화하고, 분석 코드를 파싱 코드와 분리함으로써 사용자로 하여금 분석 과정에 집중할 수 있도록 도와준다.

1. 서론

정적 프로그램 분석기는 기존에 사용자가 원시코드를 보면서 결점을 발견하는 과정을 자동으로 처리해 주는 프로그램이다. 정적 프로그램 분석기는 프로그램을 실행하지 않고 프로그램 코드를 파싱하여 프로그램 문장들의 의미를 파악한다. 이 과정에서 문장이 적합한 구조로 잘 구성되었는지를 검사하고 프로그램 제어흐름에 대해 추론하거나 프로그램 데이터에 가능한 오류들을 검출한다. 즉 컴파일러가 기본적으로 제공하는 오류 탐지 기능을 보완한다고 할 수 있다.[1] 가장 널리 쓰이는 도구는 Lint로써 컴파일러가 오류로 인식하지 못하는 잠재적인 문제점들에 대해 경고를 제공함으로써 사용자가 문제점들을 수정하도록 유도한다.[2]

최근의 정적 프로그램 분석기는 위에서 기술한 초기 목적 이외에 여러 방향으로 진화되고 있는데, 대표적인 예가 역공학, 재공학, 메트릭, 포맷팅 도구들이다. 이러한 도구들은 대부분 대상 코드를 특정 형태로 변경하는 접근방식을 취하고 있으며, 이 과정 초기에 프로그램의 정적 의미 분석 과정을 거친다.

많은 연구와 도구들이 정적 프로그램 분석기를 사용하고, 이를 언급하고 있지만, 정적 프로그램 분석기를 제작하는 과정은 다루기 힘든 분야이다. 프로그램의 특정 부분에 대해 간단한 정적 분석을 하려고 할 때에도, 항상 렉서, 파서 레벨부터 처리하는 과정을 거쳐야 한다. 어휘 분석과 구문분석 프로그램을 편리하게 구현하기 위해, lex과 yacc이 각각 개발되어 사용되고 있지만, 이 자체도 BNF 정도의 지식이 아닌, 각 파서 생성기에 대한 지식을 많이 요구한다. 이러한 초기 접근의 어려움은 프로그램 이해(program understanding) 영역에서 실험 단계의 장애요소가 되고 있다.

기존 도구들은 도구 명세서에 정의되어 있는 영역에 대해서 처리하여 결과값을 제공하고 있지만, 사용자의 생각을 직접 코드형태로 구현하고, 결과를 모델형태로 표현하여, 시험하고 검증할 수 있도록 도와주는 프레임웍 수준 도구들은 거의 나와있지 않다. 또한 대부분의 도구들은 최근의 언어를 지원하지 않는다.

본 연구는 한국과학재단 목적기초연구(R01-1999-00238)지원으로 수행되었음.

본 논문에서는 정적 프로그램 분석을 쉽게 할 수 있도록 지원하는 기반도구를 제안한다. 이 도구는 대상언어에 대한 파싱 코드와 분석 코드를 분리하여, 사용자로 하여금 분석에 집중할 수 있도록 도와주고, 분석 결과를 시각화하기 위한 라이브러리를 제공한다.

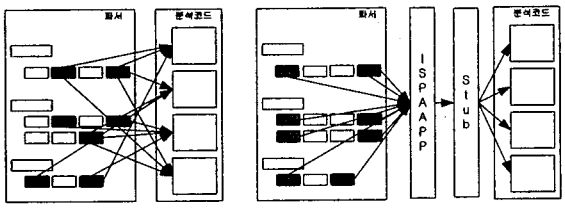
본 논문의 구성은 다음과 같다. 2장에서는 기존의 프로그램 정적 분석 도구에 대해서 기술하고, 3장에서는 본 도구의 구조를 설명한다. 4장에서는 본 도구의 장점에 대하여 논하고, 5장에서는 결론과 향후 연구를 기술한다.

2. 관련 연구

H. Lichter와 G.Riedinger는 [3]에서 자동화된 정적 프로그램 분석기가 가지는 기능들을 설명하고 있다. [3]은 정적 분석 도구를 분석기(Analyser), 결과(Result), 편집기(editor), 품질 모델(Quality Model)로 나누고 각각의 역할에 대해 설명한다. 정적 분석의 초기 단계에서 프로그래밍 언어에 의존적인 코드 검사기(programming language dependent code checker)에 의해 파싱하여 분석되고, 그 결과가 파일 또는 데이터베이스에 저장된다. 분석 결과는 편집기와 품질 모델에 의해 평가되는데, 편집기는 분석결과를 사용자가 이해하기 쉬운 형태로 표현하여 판단을 사용자에게 맡기는 반면, 품질 모델은 자동으로 분석결과를 감사하기 위하여 쓰인다. 본 논문에서 다루고 있는 형태와 가장 유사한 것은 JJTree[4]이다. JJTree는 자바용 파서 생성기인 JavaCC과 같이 제공되는 툴로써, JavaCC 코딩을 쉽게 할 수 있도록 도와주는 선처리기이다. JavaCC는 하향식의 LL(n) 파서이고, 사용자는 yacc과 비슷한 방식으로 문법을 기술한다. JavaCC에서는 각 문법이 파싱될 때 이루어져야 하는 액션들을 생성 규칙내에 삽입함으로써 파서 코딩을 하는데 비해 JJTree에서는 추상 의미 트리(Abstract Syntax Tree)를 구성하고, 각 노드에 특정 기능을 수행하는 Visitor[5]를 삽입하는 방식으로 이루어 진다. JJTree는 구문 분석과 각 해당 구문의 처리를 분리함으로써, 처리 코드를 별도의 모듈로 모을 수 있게 한다. 그러나 JJTree는 많이 사용되는 yacc 문법이 아닌 JavaCC 자체 문법을 기반으로 하고 있고, 구문 분석 처리과정만을 단순화하는 초점을 맞추고 있다는 한계점이 있다.

3. ISPA

본 논문에서는 정적 프로그램 분석을 위한 기반구조(Infrastructure for Static Program Analysis)를 제안한다. (이하 ISPA로 표기). ISPA 목적은 프로그램 분석기들을 작성할 때 분석 대상 언어에 대한 의존도를 줄이고, 프로그램 분석과정에 공통적으로 필요한 부분들을 라이브러리 형태로 제공하여 쉽게 시험할 수 있는 환경을 제공하는 것이다. 다양한 응용 프로그램으로 확장될 수 있도록 일반화된 구조를 가진다. 비슷한 형태를 가진 언어들의 코드들을 분석할 때, 각각을 별도로 프로그래밍하기보다는 한 언어를 대상으로 만들어진 하나의 분석 코드를 다른 언어에도 쉽게 적용하여 재사용 할 수 있도록 한다. [3]에서 제시한 정적 프로그램 분석기 구조 중 분석기와 편집기를 수정하여 본 ISPA를 제안한다. 결과 및 품질 모델은 별도의 모듈로 구성하지 않고 분석 코드에 사용자가 코드 형식으로 직접 구현하여 통합한다. ISPA의 분석기는 그림1의 (b) 접근방식을 사용한다.



(a). Ad-Hoc 접근방식 (b). CSAI 접근방식

그림 1

본 논문의 접근방식에서는 파서와 분석 코드를 분리하고, 그 사이에 ISPAApp(ISPA Application)와 stub를 둠으로써 계층화 아키텍쳐의 형태를 갖는다. ISPAApp는 일정한 인터페이스를 가지고 있고, 파서내에서 이 인터페이스를 호출한다. ISPA는 이미 공개되어 있는 yacc 코드에서 ISPAApp 계층의 인터페이스를 호출하도록 yacc 코드를 변경하는 도구를 포함한다. 이와 같은 접근방식은 파서 계층을 쉽게 다른 것으로 변경할 수 있게 한다. ISPAApp는 분석코드 계층으로 단순 문자열을 전달하고 stub는 사용자가 작성한 분석 코드와 생성 규칙(production rule)의 대응 표(mapping table)를 참조하여 문자열을 적절한 분석 코드 호출로 변환한다. 이 시스템은 그림 2 의 구조를 갖는다.

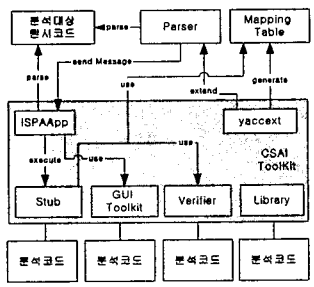


그림 2

ISPA를 사용하기 위한 사용자 시나리오는 다음과 같다.

A. 패키지에 같이 제공하고 있는 언어 파서를 사용하거나, ISPA와 함께 제공되는 응용인 yaccext를 이용하여 기존의 yacc 코드를 확장하고 컴파일한다.
B. 분석 코드를 갖는 클래스를 작성한다. 이 클래스는 ISPA에서 제공하는 클래스를 상속하여 구현하여야 한다.
C. B에서 구현된 클래스를 컴파일하고 ISPA에 삽입한다.
D. 생성규칙-분석코드 대응 표를 작성한다.
E. 대응 표를 변경하면서 적절한 생성 규칙을 선택한다.

그림 3은 일반적인 분석 과정에서 클래스들 사이의 상호작용을 보여준다. 위의 사용자 시나리오를 수행하여 모든 사전 조건이 만족되면 다음과 같은 순서로 분석과정이 진행된다.

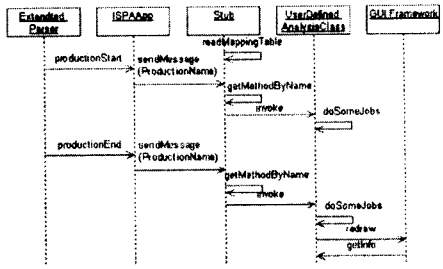


그림 3

3.1 Yacc 코드의 확장

ISPA에서 사용하는 파서는 문법의 모든 생성 규칙이 확장할 때와 감소(reduce)할 때 ISPAApp에 메시지를 보낸다. 전달되는 메시지는 어떤 생성 규칙이 사용되고 있는지를 ISPAApp가 알 수 있도록 yacc 코드내에서 유일한 값으로 지정되어야 한다. 본 연구에서는 생성 규칙의 이름을 기반으로 식별자를 생성한다. 다음은 분석 대상언어가 C 언어이고, yacc에서 C 파서의 함수 정의 부분이 파싱되고 있을 때 ISPAApp로 메시지를 전달하도록 작성한 예이다.

```
function_definition:
        { ISPAApp.productionStart (" funtion_definition" ); }
          identifier_declarator
          compound_statement
        { ISPAApp.productionEnd(" funtion_definition" ); }
    |
        { ISPAApp.productionStart (" funtion_definition" ); }
          declaration_specifier
          identifier_declarator compound_statement
        { ISPAApp.productionEnd(" funtion_definition" ); }
        .....
;
```

음영으로 돌어간 부분이 ISPAApp에 메시지를 보내도록 추가된 코드이다. 규칙 우측 부분이 여러 개의 요소로 구성되어 있는데, 중간부분에 종단 기호(terminal symbol)가 있고 이 종단 기호를 처리 중일 때는 현재 진행 상황에 대해 ISPAApp에 메시지를 전달하지 않는다. 여기서는 단지 비종단 기호를 처리할때만 메시지 전달을 수행한다.

일반적으로 대상 언어의 문법이 복잡하거나, 사용자가 직접 작성하지 않은 yacc 코드를 사용할 경우, 현재 처리중인 토큰이 어떤 생성 규칙에 의해 파싱되는지 알아내는데 어려움을 겪는다. 이 때 많이 쓰는 방법은 파싱에 사용되는 규칙들을 모두 순차적으로 기록하는 것이다. 분석기 개발자는 기록된 정보를 바탕으로, 적절한 곳에 분석 코드를 집어 넣는다.

ISPA에서도 마찬가지로 ISPAApp는 파서에서 전달된 메시지를 모두 기록한다. 그러므로 파싱 과정을 시각화 하기 위해서 사용자는 모든 생성 규칙마다 ISPAApp로 메시지를 보내는 코드를 삽입해야 한다. 이러한 과정이 쉽게 이루어지도록 ISPA 도구집합은 자동으로 yacc 코드를 파싱하여 ISPA와 연동될 수 있도록 확장해주는 응용 프로그램인 yaccext을 제공한다. yaccext는 yacc 코드에서 생성 루틴을 만날 때 마다, 생성 규칙의 시작 부분과 끝나는 부분에 생성 규칙 식별자를 전달하는 코드를 자동으로 삽입한다. 이외에도 응용 프로그램으로 확장할 때 쓰이지 않는 메시지 전달 코드를 삭제하는 기능도 수행한다. 또 yaccext는 대응 자료를 위해 템플릿을 생성한다. 다음은 위에서 예제의 함수 정의 생성 규칙 부분의 템플릿이다.

```
...
<function_definition_start>
    <!--Join Method here-->
</function_definition_start>
<function_definition_end>
    <!--Join Method here-->
</function_definition_end>
...
```

3.2 ISPAApp

ISPAApp는 ISPA의 중심에 해당하는 모듈이다. ISPAApp는 시스템을 실행하는 main 함수를 가지고 있고 분석과정을 초기화하며, 파서가 보낸 메시지를 stub으로 전달하는 역할을 맡는다. 이 외에도 파서에서 처리하지 못한 대상 코드에 대한 정보를 보완하는 역할도 맡는다. 정적 프로그래밍 분석의 응용분야, 예를 들면 코딩 표준 등의 분야에서는 기존의 어휘 분석기에서 토큰 구분자(delimiter)로만 사용하고 처리를 하지 않는 구분자도 토큰으로써 인식하여야 한다. 이를 어휘 분석기에서 토큰으로 인식하도록 변경하기 위해서는 yacc 코드의 문법도 변경되어야 한다. 그러나 이런 접근방식은 yacc에서 처리할 수 있는 문법을 넘어서게 된다 [6]. 그래서 여기서는 이 정보를 yacc이 아닌 ISPAApp에서 처리한다. ISPAApp는 파서로부터 토큰을 전달 받고 토큰의 위치를 대상코드에서 알아낸 다음 이전 토큰과의 위치 계산을 통하여 구분자 토큰을 별도로 저장한다. ISPAApp는 전달된 모든 토큰과 어휘 분석기에서 얻어낸 정보를 이용하여 다음을 유지한다.

- 코드상의 현재 토큰의 줄번호, 열번호, 토큰번호
- 토큰 문자열, 이전 토큰 문자열
- 파싱된 토큰과 구분자 목록
- 파일의 이름, 생성 날자, 크기
- 읽혀진 토큰까지의 바이트 수

또한 ISPAApp는 분석 코드를 작성하는 사용자가 분석 프로그램의 흐름을 파악하기 위하여 파서에서 전달된 메시지를 기록한다.

3.3 stub

stub는 ISPAApp로부터 전달된 식별자에 연결된 루틴을 이용하여 적절한 정적 프로그램 분석을 수행한다. 이를 위해서 별도로 대응표를 갖고 있는데, 사용자는 이 대응 표를 작성할 책임을 가진다. 사용자는 yaccext에 의해 만들어진 대응 표 템플릿 중에서 정적 분석 루틴의 메소드를 처리할 식별자에 기록함으로써 대응 표를 작성한다. 파서 표에는 한 식별자에 대해 다수의 대응이 존재할 수 있다. 다음은 대응 표 예로서 처리가 쉽도록 XML 문서로 구현한다.

```
.....
<function_definition_start>
    startFunctionLineCount
</function_definition_start>
<function_definition_end>
    endFunctionLineCountEnd
</function_definition_end>
.....
```

stub는 위의 대응 자료를 읽어 ISPAApp에서 전달된 메시지를 분석 코드 호출로 변경한다. ISPA는 자바로 구현되었고, 실행 시간에 어떤 메소드를 호출해야 하는지 결정하기 위해 자바 리플렉션(reflection)[7]을 사용한다. 다음은 ISPA에 입력되는 분석 코드의 예이다.

```
import kr.ac.snu.selab.*;
public class FunctionLineCalc extend AnalysisBaseObject {
        int rLineCount[100];  // 각 함수의 길이를 저장하는 배열
        int cFuncCount = 0;   // 처리한 함수의 개수
        int tLineCount = 0;   // 함수의 시작 줄번호 기록

        public String getAnalysisName() {
            return " Function Line Count" ;
        }
        public void startFunctionLineCount() {
            tLineCount = getISPAApp().getCurrentLine();

        }
        public void endFunctionLineCount() {
            rlineCount[cFuncCount] = tLineCount -
                        getISPAApp().getCurrentLine();
            redrawAll();
            cFuncCount++;
        }
        public analysisModel getModel() {
          .....
        }
}
```

앞의 분석 코드는 함수 길이를 알아내는 간단한 분석 클래스이다. 모든 분석 클래스는 AnalysisBaseObject를 상속한다. 이 부모 클래스에는 GUI 도구 집합과 ISAPApp에 접근하는 메소드들이 구현되어 있으므로 분석 코드가 ISPAApp으로부터 파싱 정보를 얻거나 GUI의 화면 갱신을 할 수 있도록 도와준다.

3.4 GUI 도구 집합

GUI 도구 집합은 정적 프로그램 분석기의 편집기에 해당하는 부분으로써 분석결과를 시각화 하는 것을 도와주는 라이브러리이다. 본 연구에서는 MVC 모델[5]을 이용하여 작성한다. 지원하는 GUI 표현형태는 그래프, 트리, 표로써 각 형태마다 자료 모델을 기반으로 하여 화면에 출력한다. 분석 코드는 이 모델을 반환하는 메소드를 포함하고 있어야 하며, GUI 도구 집합은 분석 코드의 화면 갱신 요구시 분석 코드로부터 자료 모델을 가져온다.

4 기대효과

정적 프로그램 분석기를 ISPA를 기반으로 작성하면 다음과 같은 장점을 얻는다.

1. 코드 정적 분석 프로그래밍 과정을 단순화한다.
   - 사용자는 분석 코드에 초점을 맞출 수 있고 공개되어 있는 yacc 코드를 자동으로 확장하여 연동하므로 파싱 처리에 드는 비용을 줄일 수 있다.
   - 분석 코드의 디버깅이 용이하다.
2. 한 분석 코드를 다양한 언어에 쉽게 적용할 수 있다.
   - C++과 Java 같이 언어의 형태가 유사한 경우 대응 표와 파서만을 대치함으로써 분석 코드를 옮길 수 있다. 대응 표와 파서를 변경하는 과정도 자동화한다.

5 결론

본 논문에서는 정적 프로그래밍 분석이 더 용이하고 간단하게 구현될 수 있도록 도와주는 하부구조를 설계하였다.

본 도구는 기존의 파서를 이용하여 확장하였고, 사용자는 대응 자료와 분석 코드를 작성하여 도구에 입력함으로써 정적 분석 과정에 필요한 노력을 최소화 한다. 본 도구를 사용하면 오류를 줄일 수 있고 분석 과정에만 초점을 맞춤으로써 신뢰성을 더욱 높일 수 있다.

그러나 아직 심볼 테이블 등의 파서 구축에 필요한 요소들에 대해서는 정의하고 있지 않으므로 사용자가 직접 작성해야 한다.

현재 프로토타입을 구현하고 있으며 앞으로 언어에 독립적인 심볼 테이블 구축에 대한 연구와 정적 프로그램 분석 과정의 패턴을 조사하여 이것을 일반화시키는 연구를 수행할 것이다.

6 참고문헌

[1] I. Sommerville, *Software Engineering*, 6th Edition, Addison-Wesley, 2001.

[2] S. C. Johnson, " Lint, a C program checker," *Unix Programmer' s Manual*, Seventh Edition Volume 2A, January 1979.

[3] H. Lichter and G. Riedinger, " Improving Software Quality by Static Programme Analysis," *Proc. Of Software Process Improvement*, 1997.

[4] Webgain Corporation. JJTree. http://www.webgain.com/products/java_cc/

[5] E. Gamma, et al., *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[6] J. Levine, T. Mason and D. Brown, *lex & yacc, 2nd Edition*, O'Reilly, 2002.

[7] K. Arnold and J. Gosling, *The Java Programming Language*, Second Edition, Addison-Wesley, 1998.