바이트 코드 검증을 위한 스택리스 중간표현 설계

김선태*, 김제민*, 박준석*, 유원희*
*인하대학교 컴퓨터정보공학과
e-mail: kst_1205@nate.com

BIRS ; ByteCode Intermediate Representation With Specification

SeonTae Kim*, JeMin Kim*, JoonSeok Park*, WeonHee Yoo*
*Dept of Computer and Information Engineering, Inha University

요 약

자바는 개발환경의 편리성과 재사용성, 이식성으로 다양한 시스템 환경에서 사용한다. 그러므로 자바는 오류 없이 안전하게 실행하는 것이 중요하다. 하지만 자바 바이트 코드를 통한 자바의 안전한 실행에 대한 검증은 스택코드, 코드의 정보부족 등의 이유로 검증을 어렵게 한다. 본 논문에서는 자바 바이트코드의 문제점을 해결하여 검증을 수행하는데 적합한 중간표현 언어를 소개한다. 중간표현 언어는 스택리스코드로 구현되며, 모든 명령어의 정보를 담고 있다. 이를 통해, 자바 바이트코드를 통한 검증을 수행할 것이다.

1. 서론

현재 자바는 객체지향 언어로서 개발환경의 편리성, 재사용성, 안전한 실행, 이식성 등의 많은 장점이 있다. 이러한 장점으로 자바는 응용 프로그램 뿐만 아니라 웹 환경과 임베디드 시스템 환경에서 널리 사용한다. 그러므로 자바 프로그램은 오류 없이 안전하게 실행하는 것이 보장되어야 한다. 하지만 자바의 안전한 실행을 보장하는 데에 있어 몇 가지 문제점이 존재한다. 첫째, 자바 코드는 프로그래머가 공개하지 않은 한 접근하여 분석할 수 없다는 문제가 존재한다. 둘째, 자바의 검증에는 자바 바이트를 이용하여 검증을 수행한다. 하지만 자바 바이트 코드를 역컴파일 했을 때 기존의 코드를 100% 복구할 수 없어 검증이 어렵다. 또한 자바는 스택기반 언어이기 때문에 아래와 같은 문제점으로 인해 검증을 어렵게 한다.

- 표현이 명확하지 않고, 복잡한 연산의 경우 여러 개의 바이트 코드 명령어를 이용하여 사용한다.
- 오퍼랜드 스택의 특성으로 인해 주어진 동작 명령어와 연관된 표현 명령어가 연속해서 나타나지 않을 수 있다.
- 바이트코드는 단편적인 구조를 가지고 있기 때문에 다양한 정보 및 동적 정보교환이 이루어지지 않을 수 있다.
- 스택은 값을 재사용하기 어려운 구조이기 때문에 불필요한 적재와 저장 명령어가 많다.

이를 해결하고자 본 논문에서는 자바 코드 검증에 필요한 중간 언어를 제안한다. BIRS(ByteCode Intermediate Representation With Specification)라고 명명한 이 언어는 소스가 컴파일 할 때 잃어버리는 정보를 복구하기 위해, 객체지향언어로서 가져야 하는 메소드, 클래스 구조 등의 구조물과 모듈정보를 포함하고 있으며, 명령어 표현은 스택리스 코드로 표현한다. BIRS는 추후 명세 언어 정의와 검증에 대한 연구를 할 때 사용될 것이다. 본 논문의 구성은 2장에서 관련연구와 3장에서 중간코드 설계 4장에서는 구현 5장에서는 중간 코드의 변경 예시와 평가를 다룬다. 마지막 6장에서는 결론과 향후 연구 과제에 관해 논의 할 것이다.

2. 관련연구

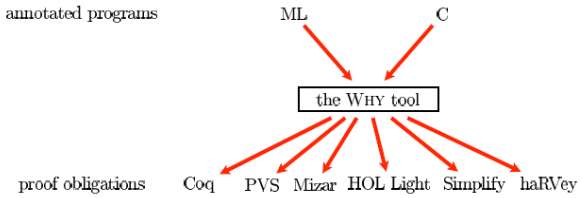
2.1 Sawja[1]

Sawja는 프로그램을 정적 분석하기 위한 오토매틱 검증의 테크닉 중 하나이다. OCAML[2] 모듈에서 자바의 .class 파일의 파싱을 통해 명령어를 스택리스 코드로 바꿔주는 역할을 한다.

2.2 BoogiePL[3] & WHY[4]

BoogiePL과 WHY는 BIRS와 같이 프로그램 검증을 위해 생성된 중간언어다. BoogiePL은 프로그램 분석과 검증을 위한 중간언어로 명령형 언어를 표현한다. BoogiePL은 Spec#을 제공하여 프로그램의 정보를 알려준다. WHY는

Coq[5], **PVS**[6], **HOL Light**[7], **Mizar**[8]의 4개의 Proof Assistant와 Verification tool로 Simplify[9], **haRVey** decision procedure[10]로 구성되어 있다. WHY는 입력 언어에 한계가 없이 모든 언어를 표현한다.

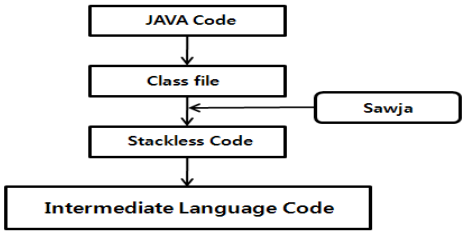


[그림 1] WHY Tool

3. 중간 코드 설계

3.1 중간코드 설계방법

BoogiePL과 WHY를 참조하여, 역컴파일 시 프로그램을 복구할 수 있을 정도로 많은 양의 정보를 저장하도록 중간언어를 설계한다. 기존의 중간표현 언어에서 표현한 Goto문을 없애고, Flow를 생성하여 제어흐름을 쉽고, 명확하게 판단할 수 있도록 설계하였다. 중간코드 설계에서 가장 중요한 부분인 Command 부분은 Sawja를 이용해 명령어를 스택리스 코드로 변경하고 이 코드를 OCAML을 이용하여 Basic Block을 구분하고 각 Block에 모든 정보를 저장할 수 있도록 설계하였다. [그림 2]는 중간언어 코드의 설계방식을 나타낸다.



[그림 2] 중간코드 설계

3.2 중간코드

중간코드는 아래와 같은 구조의 문법으로 기술한다.

3.2.1 Program and types

Program ::== *Variable* | *ComputationFunctionDec* | *PredicateDec* | *TypeDec* | *LogicalFunctionDec*

BIRS는 변수와 함수, 타입의 정보를 알려주며 시작한다.

Type ::== **bool** | **int** | **float** | **string** | **unit** | **x** | *Type* **ref** | *Type* **array**
TypeDec ::== **type** x
TypeList ::== *Type* ["," *TypeList*]

Type은 Basic Type과 User-defined Type 그리고 Array Type으로 구분한다. Unit의 의미는 C언어의 Void와 동일하다. Type ref는 Object, Pointer, address를 표현한다.

3.2.2 Logical Representation

LogicalFunctionDec ::== **logicalfunction** *Id* : (*TypeList*) → *Type*
LogicalExpr ::== **T** | **F** | *ArithExpr RelOp ArithExpr* | *ArithExpr BoolOp ArithExpr* | "¬" *LogicalExpr* | *LogicalExpr* "⇒" *LogicalExpr* | *LogicalExpr* "≡" *LogicalExpr* | "forall" *TypeIdList* "." *LogicalExp* | *PredicateId*(*ParamList*) | "exists" *TypeIdList* "." *LogicalExpr*| *PredicateDec*
PredicateDec ::== **predicate** *PredicateId* (*ParamList*) **:=** (*LogicalExpr*)
PredicateId ::== *Id*

LogicalFunction에서는 Function의 parameter가 무엇인지 return type이 무엇인지 알려주는 정보제공의 역할을 수행한다. LogicalExpr은 assert와 assume을 표현하기 위하여 설계하였다. PredicateDec는 복잡한 논리연산의 formula를 간단하게 표현할 수 있도록 하는 역할을 한다.

3.2.3 Function

ComputationFunctionDec ::==**computationfunction** *Id* : (*ParamList*) → *Type*{ [**require** *LogicalExpr*] [**read** (x,....,x)] [**write** (y,....,y)] *Body* [**ensure** *LogicalExpr*]}
ParamList ::== *Param* ["," *ParamList*]
Param ::== *Type VariableId*

ComputationFunction에서는 Function에서 수행하는 모든 정보를 가지고 있다. Function의 이름과 parameter와 return타입, Require와 ensure의 정보도 정의한다.

3.2.4 Body

Body ::== [*LocalVariable*] *Block*$^{+}$ *Flow*$^{+}$ *Return*$^{+}$
Block ::== **block** *Label* : *Command*$^{+}$
Flow ::== **from** *Label* **to** *Label* [**when** *BooleanExpression*]
Return ::== **returnblock** *Label*
Label ::== *Id*

Function의 주요 코드를 표현한다. Basic Block 단위로 각 명령어를 표현하며, Flow를 통해 프로그램의 흐름을 알려준다. Block은 프로그램 명령어 라인의 첫 번째 시작 라인으로 정의한다.

3.2.5 Command

```
Command ::== Index : Instr | assert LogicalExpr |
             assume LogicalExpr
Index ::== Id
Instr ::== assignVal VariableId := ArithExpr |
  referenceObj VariableID := RefExpr|
  modArray VariableId [ArithExpr] := ArithExpr|
  nop |
  affectField (ObjectId . VariableId :x.Type)
    := ArithExpr|
  affectStaticField(ObjectId . VariableId : x.Type)
    := ArithExpr|
  goto Index |
  ifd BooleanExpr Index |
  throw ExceptExpr |
  vreturn Expr |
  return |
  new VariableId := new ParamList|
  newArray VariableId
  := newarray Type [ArithExpr] |
  invokeStatic | invokeVirtual |
  monitorEnter  | monitorExit |
  mayInit | check
```

Command는 Index를 기준으로 표현한다. 위에서 정의한 PredicateDec를 이용하여 assert와 assume을 표현한다. 명령어는 Java에서 수행되는 모든 명령어를 인식할 수 있다.

3.2.6 Axioms

```
Fact ::== fact LogicalExpression
```

어떠한 경우에도 항상 사실인 값을 나타낸다.

3.2.7 Variable

```
Variable ::== variable TypeIdList
TypeIdList ::== ParamList
LocalVariable ::== Variable
VariableId ::== Id
ObjectId ::== Id
```

Variable은 위치에 따라 명칭을 다르게 하여 표현한다. ParamList와 TypeIdList는 형태는 같지만 위치가 다르기 때문에 구분되어 표현한다. LocalVariable과 Variable도 이와 같은 이유로 구분되어 표현한다.

3.2.8 Expression

```
Expr ::== ArithExpr | FloatExpr | CharExpr |
      ExceptExpr
ArithExpr ::== n | Id |
          ArithExpr ArithOp ArithExpr |
             "-" ArithExpr |
          VariableId [ ArithExpr ]
ArithOp ::== "+" | "-" | "*" | "/"
BooleanExpr ::== true | false |
         ArithExpr RelOp ArithExpr |
         BooleanExpr BoolOp BooleanExpr |
         "¬" BooleanExpr
RelOp ::== "=" | "≠" | "〈" | "≦" | "≧" | "〉"
BoolOp ::== "∧" | "∨"
RefExpr ::== null | RefId
RefId ::== Id
ExceptExpr ::==
FloatExpr ::==
CharExpr ::==
```

Expression은 산술연산식의 표현과 float, character형의 표현 그리고 Relation의 값들을 표현한다. 위 표현식에서 BooleanExpr를 명시한 이유는 LogicalExpr에서 사용되는 용도가 다르므로 구분하여 명시한다.

4. 구현

BIRS의 구현은 OCAML 모듈을 사용하여 구현한다. OCAML 모듈에서는 사용자가 원하는 타입을 정의할 수 있기 때문에 이를 이용하여 각 중간코드를 정의한다.

자바의 .class 파일은 Sawja를 통해 스택리스 코드로 바꾼 다음 분기와 Function Call의 명령어를 읽어 Basic Block으로 구분한다. 그 다음 각 리더를 읽어 Block의 이름을 할당한다. 그 후 Flow를 정의한다. 각 명령어의 정보는 라인별로 할당되어 표현한다.

또한 Verification Condition 생성에 필요한 정보를 표현하기 위해(assertion, assume) Two Pass로 구현한다.

```
type instr = AssignVal of JBir.var * JBir.expr |
ReferenceObj of JBir.var * JBir.expr |
ModArray of  JBir.expr * JBir.expr * JBir.expr |
Nop |
 Ifd of ([ `Eq | `Ge | `Gt | `Le | `Lt | `Ne ]
*JBir.expr *JBir.expr) * int |
Throw of JBir.expr |
Return of JBir.expr option
.....
```

[그림 3] 중간코드 구현(명령어)

5. 중간코드 변경 예시 및 평가

```
public class Example{
          int x;
          public void A(int y){
                  if(y>0)
```

```
                    x+=y;
            }
}
```

[그림 4] 예시 코드

```
variable int x
predicate P(int y) := (y>0)
logicalfunction A :( int ) → unit
computationfunction A :( int y) → unit{
read ( y ) write ( x )
block 0 : 0 : ifd  y>0  1
assert P(y)
block 1 : 1 : check
          2 : check
          3 : affectField this.x : Example.int :=
              y+x
block 4 : 4 : return
from 0 to 1 when y>0
from 0 to 4 when ¬ y>0
from 1 to 4
returnblock 4 }
```

[그림 5] BIRS 변경 코드

[그림 4]와 [그림 5]는 BIRS로 변경된 예제 코드다. [그림 5]에서 보는 것과 같이 VC 생성에 필요한 정보인 assert를 간단히 표현하기 위해 predicate P로 정의되어 표현한다. Block의 이름은 명령어 라인의 첫 번째 시작라인으로 정의하여 구분되며, Block 안에 있는 명령어는 라인으로 구분하여 모두 표현한다. 프로그램의 전체 흐름을 보여주는 Flow는 명령어를 모두 표현한 다음에 표현한다.

BIRS는 스택리스 표현으로 설계되었고, 기존의 중간표현 언어보다 많은 정보를 가지고 있기 때문에 검증에 유리하다. 또한 프로그래머가 판독하기 쉽도록 설계하였기 때문에 자바 바이트 코드를 분석하는데 많은 도움이 될 것이다.

6. 결론 및 향후과제

자바의 중간표현 언어는 다양하게 존재한다. 하지만 이와 같은 표현식은 자바언어 검증을 수행하는 데에는 적합하지 않다. 본 논문에서는 바이트코드를 스택리스 코드로 변경하고 많은 정보를 저장하여 VC생성에 유리하도록 중간표현 언어를 설계하였다.

향후 본 논문에 floating, Character, Exception Expression과 Instruction을 추가하여 BIRS를 완성한다. 완성된 BIRS의 내용을 바탕으로 객체지향 언어에 대한 검증을 위해 부작용, 상속 등과 같은 특징을 표현할 수 있는 명세언어를 정의한다. 생성된 명세언어는 SMT-sovler를 사용하여 검증을 수행할 예정이다.

논문 사사(acknowledgement)

이 논문은 2010년도 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임(No. 2010-0025660)

참고문헌

[1] Laurent Hubert, Nicolas Barré, Frédéric Besson, Delphine Demange, Thomas Jensen, Vincent Monfort, David Pichardie, and Tiphaine Turpin. Sawja: Static analysis workshop for java. Technical report, CNRS, INRIA, ENS Cachan, June 2010. Presented at the International Conference on Formal Verification of Object-Oriented Software (FoVeOOS) in June 2010.

[2] The Objective Caml language. http://caml.inria.fr/.

[3] Robert DeLine, K.Rustan M. Leino, BoogiePL : A Typed procedural language for checking object-oriented program, Technical Report MSR-TR-2005-70, 27 May 2005.

[4] Jean-Christophe Filliatre, Why : a multi-language multi-prover verification tool, LRI-CNRS UMR 8623, Université Paris Sud, March 2003.

[5] The Coq Proof Assistant. http://coq.inria.fr/.

[6] The PVS Specification and Verification System. http://pvs.csl.sri.com/.

[7] John Harrison. HOL Light. http://www.cl.cam.ac.uk/users/jrh/hol-light/.

[8] The Mizar project. http://mizar.uwb.edu.pl/.

[9] The Simplicify decision procedure(part of ESC/Java).http://research.compaq.com/SRC/esc/simplify/.

[10] Silvio Ranise and David Déharbe. The haRVey decision procedure.http://www.loria.fr/~ranise/haRVey/.