바이트코드의 효율적인 분석을 위한 중간코드의 설계

김경수° 김기태 조선문 유원희
인하대학교 컴퓨터 정보공학과
oe0916@naver.com

Design of Intermediated code for Efficient Analysis of Bytecode

Kyung Soo Kim° Ki Tea Kim Sun Moon Jo Weon-Hee Yoo
Dept of Computer Science and Engineering, In-Ha University

요 약

자바 언어는 객체 지향 언어이며 이식성에 좋은 언어로써 각광을 받고 있다. 하지만 자바 프로그램은 이식성은 좋지만 실행 시 인터프리터 방식으로 사용하기 때문에 실행속도가 느리다는 단점이 있다. 또한 바이트코드는 스택기반의 코드이기 때문에 코드 단편화 문제점과 스택 접근 연산들을 사용하여 프로그램 분석이 용이 하지 않고, 단순한 변환을 복잡하게 만들 수 있다는 단점이 있다. 따라서 바이트코드 자체로 분석과 최적화하기가 용이 하지 못하다. 본 논문에서는 바이트코드의 분석을 위한 트리구조 중간코드를 제안 한다. 트리구조 중간코드는 스택기반 코드의 문제점을 보완하고, 기존에 3-주소 형태의 최적화 기법들을 적용할 수 있다는 장점이 있다. 본 논문은 바이트코드와 각종 정보를 가지고 있는 클래스 파일을 입력받아 분석 후 기본블록을 나누고 BNF코드를 바탕으로 트리구조 중간코드를 생성하게 된다. 생성된 중간코드를 가지고 제어 흐름 그래프를 만들게 된다. 이러한 방식으로 트리구조 중간코드를 설계하게 된다.

1. 서 론

자바 언어는 객체 지향 언어이며 이식성에 좋은 언어로써 각광을 받고 있다[1]. 그 이유는 자바 프로그래밍 환경에서 이기종간의 실행환경에 적합하도록 자바 가상 기계(JVM: Java Virtual Machine) 코드인 바이트코드 때문이다[2, 3]. 하지만 자바 프로그램은 이식성은 좋지만 바이트코드를 실행 시 인터프리터 방식으로 사용하기 때문에 실행속도가 느리다는 단점이 있다. 또한 바이트코드는 스택기반의 코드이기 때문에 복잡한 수식의 경우 여러 조각으로 나누어서 코드상의 넓게 분포되어 나타나는 코드 단편화 문제점과 스택 접근 연산들을 사용하여 프로그램 분석이 용이 하지 않고, 단순한 변환을 복잡하게 만들 수 있다는 단점이 있다. 따라서 바이트코드 자체로 분석과 최적화하기가 용이 하지 못하다. 본 논문에서는 바이트코드의 분석을 위한 트리구조 중간코드를 제안 한다. 트리구조 중간코드는 스택기반 코드의 문제점을 보완하고, 기존에 3-주소 형태의 최적화 기법들을 적용할 수 있다는 장점이 있다. 본 논문의 구성은 2장에서는 기본블록과 제어흐름그래프에 대해서 설명하고 3장에서는 트리구조 중간코드 설계를 한다. 4장에서는 결론과 향후 연구 과제에 대하여 논의할 것이다.

이 논문은 정부(교육인적자원부)의 재원으로 한국학술진흥재단의 지원을 받아 수행된 연구임"(R05-2004-000-11694-0)

2. 관련 연구

2.1 기본블록

기본블록은 제어가 시작으로 들어가서 끝점으로 나올 때까지 정지나 분기의 가능성이 없는 연속적인 문장들의 집합을 말한다. 기본블록을 나누는 데는 리더를 찾아야 한다[4]. 그러나 바이트코드는 스택을 기반으로 하는 코드이기 때문에 리더는 3-주소 형태의 코드에서의 리더와는 다르게 된다. [표 1]은 바이트코드의 분기 명령어를 보여주고 있다.

[표 1] 바이트코드 분기 명령어

명령어 종류	바이트코드 명령어
조건 분기	ifeq, ifne, iflt, ifle, ifnull, ifnonnull, if_acmpeq, if_icmple, if_icmpge
비교	lcmp, fcmp, dcmpg, fcmpg
무조건분기	goto, goto_w
서브루틴	jsr, jsr_w, ret
예외처리	athrow
테이블 점프	tableswitch, lookupswitch
메소드	return, ireturn, lreturn, freturn, dreturn

2.2 제어 흐름 그래프

제어 흐름 그래프는 많은 유용한 분석과 최적화의 기초가 된다. 제어 흐름 그래프 내에 프로그램의 코드는 많은 기본 블록들로 나눠진다. 기본블록은 처음에 간선이

들어가게 되고, 끝에는 단지 나올 뿐이었다. 따라서 각 기본 블록은 들어가게 되는 간선과 나오게 되는 간선들의 정보를 알고 있어야 한다[5]. 제어 흐름 그래프를 사용하게 되면 우선 기본 블록을 간선으로 연결해 놓았기 때문에 선행자와 후행자를 찾기가 쉽고, 프로그램 전체 내용을 분석하기가 쉽다. 또한 제어 흐름 그래프는 일반적으로 컴파일러의 최적화뿐만 아니라 디버깅, 슬라이싱 등 여러 분야에 적용되고 있다.

3. 중간코드의 설계

3.1 라인 정보

기본블록을 만들기 위해서 바이트코드의 라인 정보와 실제 자바 소스의 명령어 라인 정보를 가지고 있는 라인 번호 테이블을 이용해야 한다.

```
public class Temp {
  int f(boolean b){
        int x;
        x = 1;
        if(b)
          x = 2;
        else
          x = 3;
        return x;
       }
     }
```

```
label_0
        ldc 1
        istore x$2
label_2
        iload b$1
        ifeq label_11
label_6
        ldc 2
        istore x$2
        goto label_13
label_11
        ldc 3
        istore x$2
label_13
        iload x$2
        ireturn
```

자바소스	label을 추가한 바이트코드

[그림 1] label을 추가한 모습

라인 번호 테이블은 클래스 파일 내부에 있는 정보로써 실제 자바소스와 바이트코드의 라인정보에 대해서 가지고 있다. 라인 번호 테이블에서 실제 바이트코드의 시작 위치에 대한 정보를 얻어 그 위치에 레이블을 추가 하여 기본 블록을 생성하게 된다. [그림 1]은 레이블을 추가한 모습을 보여주고 있다. 레이블의 번호는 라인 번호 테이블에 있는 바이트코드 시작위치(start_pc)이다.

3.2 중간코드

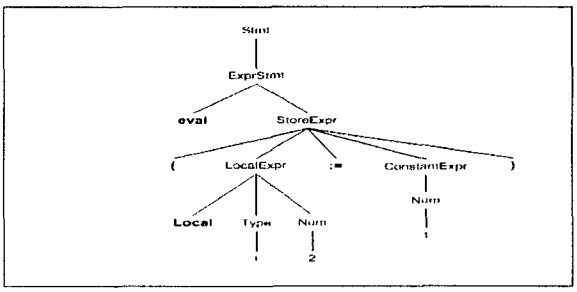
중간코드는 제어 흐름 그래프를 만들기 전에 수행된다. 트리구조 중간코드는 바이트코드의 특성을 3-주소 형태의 언어에 맞게 만드는 것으로 우선 트리 형태로 만들기 위한 BNF가 있어야 한다. [그림 2]는 BNF 코드의 일부를 보여주고 있다. BNF 코드는 두 가지로 분류 할 수 있다. 하나는 추상클래스인 Expr 클래스로부터 파생되는 표현식이고 다른 하나는 추상클래스인 Stmt 클래스로부터 파생되는 문장이다. 이 두개의 차이점은 Expr은 파생 클래스들이 값과 타입을 가지는 반면에 Stmt는 3-주소 형태의 문장을 표현한다. 이때 스택의 내용을 참고하여 스택의 위치와 해당 위치에 들어있는 값을 읽어 들여 처리하게 된다.

```
Expr → ConstantExpr | DefExpr | StoreExpr
ConstantExpr → ' Id ' | Num + F | Num + L | Num
StoreExpr → ( MemExpr := Expr )
DefExpr → MemExpr
MemExpr → MemRefExpr | VarExpr
VarExpr → LocalExpr | StackExpr
LocalExpr → Stack | Local Type Num
Stmt → ExprStmt | InitStmt | JumpStmt | LabelStmt
LabelStmt → Label
ExprStmt → eval Expr
InitStmt → INIT LocalExpr[ ]
JumpStmt → GotoStmt | IfStmt | ReturnExprStmt
IfStmt → IfZeroStmt
GotoStmt → goto Block
IfZeroStmt → if0( Expr ==|!=|>|>=|<|<=|null|0)
          then Block else Block
ReturnExprStmt → return Expr
Block → <block Label >
Label → label_ Num
```

[그림 2] BNF 코드 중 일부

앞에서 구한 레이블을 추가한 바이트코드 형태를 바탕으로 레이블별로 문장을 읽어 들인 후 각각의 명령어 별로 처리한다. 읽어 들인 명령이 레이블인지 명령어인지를 확인한 후 명령어인 경우 명령어가 리더인지를 다시 한번 확인하게 된다. 명령어는 다음 레이블이 나올 때까지 계속 읽어 들인 후 저장하게 된다. 만약 다음 레이블이 나올 경우 현재 스택의 높이를 계산하고 마지막에 나온 명령어의 형태 BNF를 이용하여 트리를 구성하게 된다. [그림 3]는 [그림 2]의 BNF를 이용해서 트리구조 중간 코드를 구성한 모습이다.

[그림 3]은 [그림 1]에서 label_0에 있는 바이트코드 ldc 1, istore x$2를 트리구조 코드로 표현한 것이다. ldc 1은 스택에 1을 넣게 되고, istore x$2는 스택에 있는 1을 지역 변수 배열 2번째에 넣게 된다. 결국 eval (Locali2 := 1) 과 같은 하나의 문장으로 표현을 할 수 있다.



[그림 3] eval (Locali2 := 1)트리구조로 표현 한 모습

분기가 있는 명령어인 경우 기본블록과 제어 흐름 그래프를 만들기 위해서 필요하다. 분기가 있는 명령어일 경우 분기의 목표가 되는 레이블을 저장한다. 그 다음 레이블 간에 간선으로 연결하게 된다. 분기가 일어나는

레이블은 레이블 블록으로 만들어 기본블록을 이루게 된다.

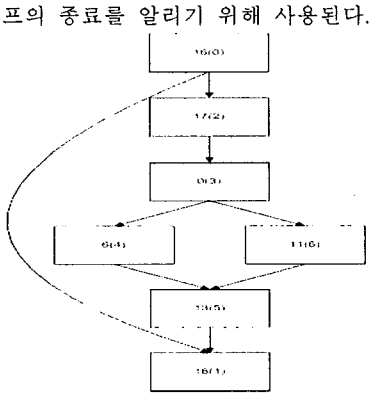
[그림 4]는 레이블을 붙인 바이트코드에서 트리구조 중간코드를 표현한 상태의 모습을 나타내고 있다.

label_0 ldc 1 istore x$2 label_2 iload b$1 ifeq label_11 label_6 ldc 2 istore x$2 goto label_13 label_11 ldc 3 istore x$2 label_13 iload x$2 ireturn label_15	<block 0> label_0 eval (Locali2 := 1) label_2 if0 (Locali1 == 0) then <block 11> else <block 6> <block 6> label_6 eval (Locali2 := 2) goto label_13 <block 11> label_11 eval (Locali2 := 3) goto label_13 <block 13> label_13 return Locali2
label을 추가한 모습	트리구조 중간코드

[그림 4] 트리구조 중간코드

레이블 간에 간선들은 연결한 후에 블록 간에 간선으로 연결하게 되면 제어 흐름 그래프가 만들어 지게 된다. 앞서 말한 것 같이 제어 흐름 그래프는 선행자와 후행자를 간선으로 가지고 있어야 한다. [그림 5]는 [그림 4]의 기본블록을 제어 흐름 그래프로 나타낸 것이다.

[그림 5]에서 3개의 노드를 추가 되어 있다. 이것은 시작노드(entry node), 종료노드(exit node) 초기화 노드(init node) 이렇게 세 가지의 노드가 추가가 된 것이다. 시작노드는 그래프의 시작을 알리기 위해서 사용되고 종료노드는 그래프의 종료를 알리기 위해 사용된다. 초기



[그림 5] 제어 흐름 그래프

화 노드는 그래프에서 사용되는 정보들을 초기화 시키는데 사용된다. 이 세 가지 노드들은 실제 제어의 흐름과는 상관없이 정보만을 가지고 있게 된다. 이 노드들은 다음 작업으로 지배자와 지배자 경계를 구할 경우 필요하게 된다. 제어 흐름 그래프는 지배자 트리와 지배자 경계를 구하기 위해 필요하다.

4. 결론 및 향후 연구

자바의 실행 속도 개선을 위한 많은 방법들이 있다. 자바 바이트코드를 이용해서 최적화를 하기 위해서는 바이트코드의 분석이 필요하다. 하지만 바이트코드는 스택기반이기에 분석과 최적화가 용이 하지 못하다. 본 논문에서는 바이트코드에서 트리구조 중간코드를 설계하였다. 트리구조 중간코드는 바이트코드의 단점을 보완하고, 효율적으로 바이트코드를 분석할 수 있게 만든 중간 표현이다.

향후 연구로 본 논문을 바탕으로 효율적 바이트코드 분석과 최적화를 위해서 지배자트리와 지배자 경계를 구해서 정적 단일 배정문을 설계할 예정이다.

참고문헌

[1] Ken Arnold and James Gosling,"*The Java Programming Language*", Sun Microsystem, 1996.

[2] Tim Lindholm and Frank Tellin."*The Java Virtual Machine Specification.*" Morgan Kaufmann,1997.

[3] John Meyer, Troy Downing,"*Java Virtual Machine*",O'RELLAY,1997.

[4] A.V.Aho, R.Sethi, and J.D.Ullman. "*Compilers: Principles, Techniques, and Tools.*" ,Addison -Wesley, 1986.

[5] Stenve S. Muchinck, *"Advanced Compiler Design and Implementation"*, Mogan Kaufmann, 1997.