CTOC에서 자바 바이트코드를 이용한 제어 흐름 분석에 관한 연구

A Study on Control Flow Analysis Using Java Bytecodes in CTOC

김기태, 유원희
인하대학교 컴퓨터 공학부

Ki-Tae Kim(kimkitae@inha.ac.kr), Weon-Hee Yoo(whyoo@inha.ac.kr)

요약

본 논문은 바이트코드(bytecode) 수준에서 프로그램 분석과 최적화를 위한 구조를 서술한다. 바이트코드 수준에서 분석을 수행하기 위해서는 우선 제어 흐름 그래프(CFG : Control Flow Graph)를 생성해야 한다. 바이트코드의 특성 때문에 기존의 제어 흐름 분석 기술을 바이트코드에 적합하게 확장해야 한다. CFG를 작성하기 위해 기본 블록을 생성하고 기본 블록간의 관계를 이용하여 최적화 과정에서 사용되는 각종 정보를 생성한다. 생성된 CFG는 자바 바이트코드의 이해와 유지보수를 위해 테스트되고, 데이터 흐름 분석과 의존성 분석과 같은 다른 분석을 위해서 사용된다.

본 논문에서는 바이트코드 수준의 제어 흐름 분석을 위해 CTOC(Classes To Optimized Classes)의 CTOC-BR(CTOC-Bytecode tRanslator)을 구현한다. CTOC는 자바 바이트코드의 최적화와 분석을 위해 현재 개발 중인 프레임 워크의 이름이고, CTOC에서 CTOC-BR은 스택 기반인 바이트코드의 최적화와 분석을 쉽게 하기 위해 트리 형태로 변환을 수행하는 도구이다.

■ 중심어 : | CTOC | 자바 바이트코드 | 제어 흐름 그래프(CFG) |

Abstract

This paper describes the data structure for program analysis and optimization of bytecode level. First we create an extended CFG(Control Flow Graph). Because of the special properties of bytecode, we must adaptively extend the existing control flow analysis techniques. We build basic blocks to create the CFG and create various data that can be used for optimization. The created CFG can be tested for comprehension and maintenance of Java bytecode, and can also be used for other analyses such as data flow analysis.

This paper implements CTOC's CTOC-BR(CTOC-Bytecode tRanslator) for control flow analysis of bytecode level. CTOC(Classes To Optimized Classes) is a Java bytecode framework for optimization and analysis. This paper covers the first part of the CTOC framework. CTOC-BR is a tool that converts the bytecode into tree form for easy optimization and analysis of bytecode in CTOC.

■ Keyword : | CTOC | Java Bytecodes | CFG(Control Flow Graph) |

* 이 논문은 인하대학교의 지원에 의하여 연구되었습니다.

접수번호 : #051006-001
접수일자 : 2005년 10월 06일
심사완료일 : 2005년 12월 27일
교신저자 : 김기태, e-mail : kimkitae@inha.ac.kr

I. 서 론

바이트코드는 자바 가상 기계(JVM)라고 불리는 인터프리터에 동적으로 적재되고 수행된다[1]. 바이트코드는 유용한 특징을 많이 갖지만 스택 기반 코드이기 때문에 수행 속도가 느리고, 프로그램 분석이나 최적화에 적절한 표현은 아니다[2][3]. 따라서 코드의 분석과 최적화를 위해서는 바이트코드를 적절한 형태로 변경한 후, 변경된 코드에 대해 제어 흐름 분석을 수행해야 한다.

기존의 자바 또는 자바 바이트코드의 제어 흐름 분석과 최적화에 관련된 연구로는 Purdue 대학의 Bloat 프로젝트나, Middle Tennessee 대학의 JAristotle 프로젝트, 그리고 McGill 대학의 SOOT 프로젝트 등을 들 수 있다[4-6]. Bloat와 JAristotle 프로젝트에서는 로딩 동작을 효율적으로 할 수 있는 방법을 제시하고 있으며 바이트코드 내에서 사용되는 메소드의 분석을 용이하게 하는 방법도 제시되어 있다. 하지만 이 프레임워크들은 바이트코드 형태에서 조작을 수행하는 특징을 가진다. SOOT 프로젝트는 3-주소 중간 표현을 이용해서 중요한 최적화를 실행한다. 하지만 바이트 코드를 네이티브 코드로 생성하기 때문에 해당 기계에 의존적인 최적화된 코드가 생성되는 특징을 가진다.

반면 본 논문에서 제시하는 CTOC는 기존의 바이트코드에 정보를 추가하여 분석을 용이하게 하고 트리 형태의 3-주소 중간 표현을 이용한 최적화를 수행한다. 하지만 최적화된 형태를 다시 바이트코드로 만들기 때문에 플랫폼에 독립적인 특징을 가지게 된다.

바이트코드의 제어 흐름 분석을 수행하기 위해서는 기존의 제어 흐름 분석 기술을 자바 바이트코드에 적합하게 확장해야 한다. 본 논문에서는 바이트코드 수준의 분석을 위해 확장된 CFG(제어 흐름 그래프)를 생성한다. 일반적으로 CFG에 의해서 표현되는 컴파일러 최적화와 프로그램 분석 기술은 제어 흐름 정보에 기반한다[7][8]. CFG는 방향 그래프(directed graph)이며 기본 블록 집합에 대한 제어 흐름 정보를 표현하는 그래프이다.

본 논문에서는 바이트코드를 위해 제어 흐름을 분석하고 최적화를 위해 바이트코드를 트리 형태로 변환한다. 이 변환을 효과적이고 체계적으로 하기 위하여 BNF 코드를 정의하여 사용한다. 그리고 CFG로부터 분석과 최적화를 위한 여러 가지 정보를 획득한다. 또한 CFG 생성 후 도달 불가능한 블록은 그래프에서 제거된다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구로 현재 개발 중에 있는 CTOC와 기본 블록에 대해서 설명한다. 3장은 제어 흐름 그래프 생성과 논문에서 사용하는 BNF와 표현 트리 생성, 그리고 도달 불가능한 블록을 제거하는 방법에 대해서 기술한다. 4장에서는 예제 프로그램을 이용하여 CTOC에 의해 작성된 제어 흐름 그래프의 여러 정보를 살펴보고, SOOT, JAristotle과 같은 기존의 다른 개발 도구와 비교한다. 5장에서는 결론과 향후 계획을 제시한다.

II. 관련 연구

1. CTOC

CTOC는 현재 개발 중인 최적화 프레임워크의 이름이다[9][10]. CTOC는 McGill 대학의 SOOT 프로젝트와 Purdue 대학의 Bloat 프로젝트에 기반을 두고 있다[5][6]. CTOC의 전체적인 구성도는 [그림 1]과 같다.

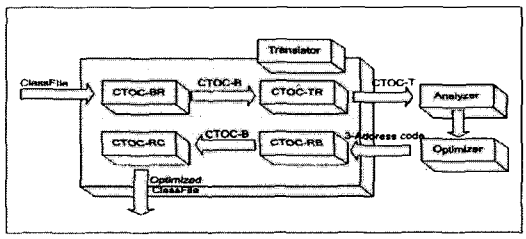


그림 1. CTOC 전체 구성도

2. 기본 블록

기본 블록은 내부에 제어의 이동이나 분기가 존재하지 않고 블록 내부로의 제어 이동도 존재하지 않는 연속적인 코드의 집합이다. 전통적인 코드에서는 기본 블록의 첫 번째 문장을 리더(leader)라고 하며, 이 리더들의 집합을 결정한다[7][8]. 각 리더마다 기본 블록은 다음 리더나 프로그램의 끝을 만나기 바로 전까지 모든 문장

으로 구성된다고 할 수 있다. 리더로 사용 가능한 명령어는[표 1]과 같다.

표 1. 리더로 사용가능한 명령어

명령어 종류	바이트코드 명령어
조건 분기 명령어	ifeq, ifne, iflt, ifge, ifle, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, ifnull, ifnonnull, if_acmpeq, if_acmpne
비교 명령어	lcmp, fcmpl, fcmpg, dcmpl, dcmpg
무조건 분기 명령어	goto, goto_w
서브루틴 명령어	jsr, jsr_w, ret
예외처리 명령어	athrow
테이블 점프 명령어	tableswitch, lookupswitch
메소드 반환 명령어	ireturn, lreturn, freturn, dreturn, areturn, return

본 논문에서는 위에서 서술한 리더 관련 명령어와 바이트코드의 코드 속성에 존재하는 라인 번호 테이블(LineNumberTable)을 이용하여 기본 블록을 생성한다.

III. 제어 흐름 그래프 구현

본 논문에서는 CTOC에서 CFG의 생성을 서술하기 위해 [그림 2](a) 예제 프로그램을 사용한다.

```
1:  public class Temp {
2:    int f(boolean b){
3:      int x;
4:        x = 1;
5:        if(b)
6:          x = 2;
7:        else
8:          x = 3;
9:        return x;
10:   }
11: }
```

```
public class Temp extends java.
lang.Object{
public Temp();
  Code:
    0:   aload_0
    1:   invokespecial  #9;
    4:   return

int f(boolean);
  Code:
    0:   iconst_1
    1:   istore_2
    2:   iload_1
    3:   ifeq          11
    6:   iconst_2
    7:   istore_2
    8:   goto 13
    11:  iconst_3
    12:  istore_2
    13:  iload_2
    14:  ireturn
}
```

그림 2. (a)예제 프로그램 (b)바이트코드

[그림 2](a)를 javap -c 옵션을 사용한 결과는 [그림 2](b)와 같다. [그림 2](b)에서 #9는 상수 풀의 인덱스를 의미하고, ifeq 11은 "만약 같은 경우"라면 11번 라벨이 있는 곳으로 분기하라는 바이트코드 명령어를 나타내는 것이다. 11:은 명령어의 오프셋(offset)을 알려주는 라벨이다.

1. 라인 정보 수집

CFG를 생성하기 위해서 가장 먼저 기본 블록을 생성한다. 기본 블록을 만들기 위해 소스 코드의 라인 번호와 코드 시작 위치(pc)에 대한 정보를 획득해야 한다. 이 두 가지 정보는 바이트코드가 생성될 때 메소드의 라인 번호 테이블 속성에 포함된 정보이다. [표 2]와 [표 3]은 자바 가상 기계의 라인 번호 테이블에 대한 형식과 라인 번호 정보 테이블의 형식을 나타낸다[1].

표 2. 라인 번호 테이블

타입	이름	사용 바이트 수
u2	attribute_name_index	1
u4	attribute_length	1
u2	line_number_table_length	1
line_number_info	line_number_table	line_table_length

표 3. 라인 번호 정보 테이블

타입	이름	사용 바이트 수
u2	start_pc	1
u2	line_number	1

라인 정보 테이블을 통해 소스 코드 상의 라인 번호(line_number)와 실제 시작 위치(start_pc)에 대한 정보를 얻을 수 있다. 시작 위치에 대한 정보를 이용해 기존 코드에 라벨을 추가하고 기본 블록을 생성할 수 있는 기반을 만든다.

[그림 2](a)의 예제 프로그램에 대한 라인 번호 정보의 내용을 eclipse의 변수(variables) 속성 창을 통해 출력한 내용은[그림 3]과 같다.

```
(lines   (line #4 pc=0)
         (line #5 pc=2)
         (line #6 pc=6)
         (line #8 pc=11)
         (line #9 pc=13) )
```

그림 3. 그림 2(a)의 라인 번호 정보 내용

[그림 3]의 라인 번호 정보에서 #4는 소스 코드에서의 라인 번호를 의미하고 pc=0은 바이트코드에서 해당 코드의 시작 위치를 의미한다. 따라서 각 pc는 프로그램 내에서 라벨이 추가될 수 있는 위치를 알려주게 된다. pc 정보를 이용하여 코드에 라벨에 대한 정보를 추가한다. 추가된 라벨은 기본 블록을 생성하는데 중요한 정보가 된다. [그림 4]는 [그림 2](b)의 바이트코드에 라벨 정보가 추가된 후의 코드 형태를 보여준다.

[그림 4]의 코드 중 f.(Z)I에서 f는 메소드 이름, (Z)는 블리언 타입의 매개변수가 1개라는 것을 나타내고, 뒤에 나오는 I는 정수형 반환 타입을 의미한다. [그림 2](b)의 int f(boolean b);와 비교하면 의미를 쉽게 파악할 수 있다. Z나 I는 블리언과 정수 타입에 대해 자바 가상 기계에서 사용되는 표기 방법을 이용한 것이다[1]. label_0과

```
f.(Z)I:
    label_0
                ldc 1
                istore Local#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
```

그림 4. 라벨이 추가된 후 int f(boolean b)

같은 라벨 정보는 기존의 코드에 새롭게 추가된 것이다. iload b#1은 [그림 2](b)에서 단순히 iload_1로 지역 변수 배열에 첫 번째 위치한 정수형 값을 스택에 적재하는 경우인데, b#1은 해당 명령이 트리 형태로 변경된 CFG를 작성하기 위해 클래스 파일에 존재하는 지역 변수 테이블(LocalVariableTable)과 상수 풀 정보로부터 지역 변수 배열의 첫 번째에 해당하는 변수 이름인 b를 직접 가져와서 표현한 것이다. 정보를 직접 표현한 이유는 필요한 정보를 코드에 직접 나타내어 더 이상 지역 변수 테이블이나 상수 풀에 대해 참조하지 않아도 필요한 정보를 획득할 수 있도록 하기 위해서이다.

2. 기본 블록 생성

기본 블록은 CFG에서 제어 흐름에 영향을 주지 않는 명령어들로 구성된다[7][8]. 프로그램의 제어 흐름에 영향을 주는 리더를 찾아 기본 블록을 작성한다.

본 논문에서 사용되는 기본 블록의 첫 문장은 새롭게 추가된 라벨(label)이 위치한다. 또한 각 블록의 마지막은 다른 블록으로 제어를 전달하는 분기문이 위치하게 된다. 이런 정보가 추가되는 이유는 CFG 작성 시 선행자(predecessor)와 후행자(successor)에 대한 정보를 쉽게 획득할 수 있고, 간선의 추가를 쉽게 하기 위해서이다. 선행자란 현재 블록을 기준으로 현재 블록을 호출한 블록을 의미하고 후행자란 현재 블록이 호출하는 블록을 의미한다.

생성될 CFG는 유향 그래프로 작성되며, 그래프의 각 노드는 기본 블록으로 이루어진다. CFG를 생성하기 위해서는 시작 노드(entry node), 종료 노드(exit node), 그리고 초기화 노드(initial node)로서 세 가지 디폴트 노드를 생성한다. 시작 노드는 그래프의 진입점을 알리기 위해 사용되고, 종료 노드는 그래프의 진출점을 알리기 위해 사용된다. 초기화 노드는 그래프에서 사용되는 매개변수와 같은 정보들의 초기화를 위해 사용된다. 이 세 가지 노드는 실제 흐름 제어에는 관련 없지만 CFG를 처리하기 위한 정보를 포함한다.

라벨 정보를 가진 블록을 생성하기 위해, 라벨 정보를 이용하여 새로운 블록을 생성한다. 시작 블록을 설정할 수 있는 블리언 필드를 추가한 후 setStartBlock(true); 메소드를 수행하여 시작 블록을 설정한다. 새로운 블록을 생성하기 위해서는 new Block(int labelNum) 형태의 생성자를 호출한다. 이 과정을 통해서 프로그램에서 사용할 모든 기본 블록이 생성된다. [그림 5]는 생성자

를 통해 생성된 기본 블록 나타낸다.

```
// entryBlock / initBlock / exitBlock 생성
   new block <label_16>      // entryBlock
   new block <label_17>      // initBlock
   new block <label_18>      // exitBlock

// 라벨위치에 따라 생성된 기본 블록
   new block <label_0>
   new block <label_6>
   new block <label_11>
   new block <label_13>
   new block <label_15>
```

그림 5. 기본 블록 생성

[그림 5]의 new block<label_16>에서 new block은 새로운 블록을 생성했다는 것을 나타내고, <label_*num*>은 해당 라벨 정보를 표현하는 것이다. 따라서 라벨 정보 label_16을 통해서 생성된 기본 블록이란 의미를 가진다.

앞에서 설명한 세 개의 디폴트 블록을 우선 생성한 후 코드내의 시작 위치 정보를 이용하여 나머지 새로운 블록들을 생성한다. 이 과정에서는 단순히 사용될 기본 블록만을 생성한다. [그림 6]은 가장 기본적인 CFG가 생성된 모습이다.

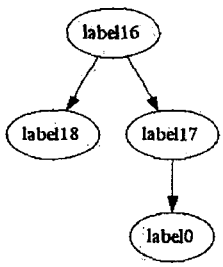


그림 6. 기본적인 그래프

[그림 6]에서 label_16은 시작 블록, label_17은 초기화 블록, label_18은 종료 블록, label_0은 프로그램의 시작 블록을 나타낸다. 시작 블록과 초기화 블록, 시작 블록과 종료 블록, 그리고 초기화 블록과 프로그램 시작 블록을 간선으로 연결하여 가장 기본적인 CFG를 나타낸다. 간선들의 연결을 위해서 addEdge(entryBlock, initBlock) 메소드를 사용한다. 이 메소드는 시작 블록과 초기화 블록 사이에 간선을 연결하는 역할을 한다. 나머지 기본 블록들도 동일한 방법으로 간선이 연결된다. 또한 그래프 관계를 유지하기 위해서 블록 간에 선행과 후행 관계를 유지하는 후행자 집합과 선행자 집합을 간선을 따라 설정하게 된다. 이 두 집합은 선행 순서 리스트와 후행 순서 리스트 작성에 사용된다.

CFG를 생성하기 위해서는 이미 생성한 기본 블록에 각 코드에 해당하는 문장을 트리 형태로 만들어 추가하고 기본 블록을 간선으로 연결하여 CFG를 생성한다. 트리를 생성하기 위해서는 makeTrees(firstBlock, labelPosition) 메소드를 사용한다. 이 메소드에서는 블록에 대한 정보인 시작 블록과 라벨에 대한 위치를 가진 해쉬 맵이 사용된다.

블록에 새로운 트리를 생성하고 명령어를 트리 형태로 추가한다. 이러한 트리를 표현 트리(expression tree)라 하는데 기본 블록 내부에서 각 명령어를 3-주소 형태의 문장으로 표현한다. 표현 트리는 커다랗게 두 가지 종류로 분리된다. 하나는 표현식이고 다른 하나는 문장이다. 표현식은 추상 클래스인 Expr 클래스로부터 파생되고, 문장 역시 추상 클래스인 Stmt 클래스로부터 파생된다. [그림 7]은 기존의 바이트코드를 트리 형태로 표현하기 위한 BNF 형태를 표현한다.

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

그림 7. BNF 코드 중 일부

Expr과 Stmt의 파생 클래스들의 차이는 Expr로부터 파생된 클래스들은 값을 유지할 수 있는 필드를 가진다는 것이다. 또한 각 Expr은 타입을 표현하기 위해 타입 필드도 가진다. Stmt는 단순히 3-주소 형태의 문장을 표현한다.

각 블록 별로 트리를 추가하기 위해서는 우선 라벨이 추가된 형태의 바이트코드를 읽어드린다. 읽어드린 내용이 라벨인지 명령어인지를 확인한 후 명령어인 경우 현재 명령어가 리더인가를 확인한 후 명령어 별로 트리를 생성한다. 이를 위해서는 각 문장에 맞게 트리가 구성되어 있어야한다. [그림 8]은 [그림 7]의 BNF를 이용하여 ExprStmt인 eval(locali2_2 := 1) 문장을 트리로 구성한 모습이다.

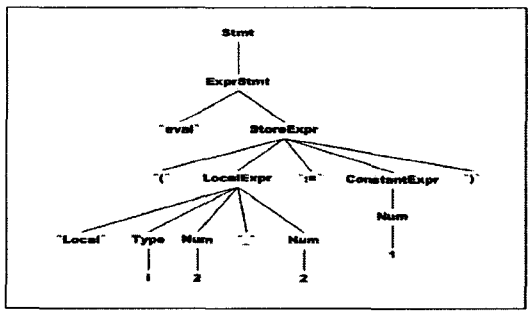


그림 8. eval(locali2_2 := 1)의 트리 구성

새롭게 트리가 추가된 각 블록에 제어 흐름을 이용하여 기본 블록 사이에 간선을 추가한다. 간선을 추가하기 위해 addEdge(block, next) 메소드를 사용한다. block은 현재 블록을 의미하고 next는 제어 흐름이 진행할 다음 블록을 의미한다. 다음 블록에 관한 정보들은 특정 명령어 다음에 나타나는 라벨 정보를 통해 구할 수 있다. 제어 흐름 정보에 의해 생성된 CFG는 [그림 9]와 같다.

3. 리스트 생성

CFG에서 그래프를 운행하면서 도달 불가능한 블록의 제거, 정적 단일 배정 형태로 변환, 그리고 다양한 최적화를 수행하기 위해서는 방문한 노드를 집합으로 유지하는 방문 집합(visited)과 전위 운행 순서를 리스트로 유지하는 전위 리스트(preorder), 후위 운행 순서를 리

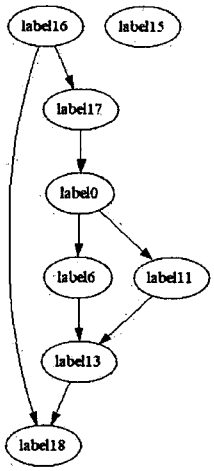


그림 9. 생성된 제어흐름 그래프

스트로 유지하는 후위 리스트(postorder)를 작성해야 한다. 이들 집합과 리스트를 생성하기 위해서는 시작 블록을 루트로 설정한 후 깊이 우선 탐색 방법으로 방문 집합과 전위, 후위 리스트를 생성한다. allList() 메소드를 이용해서 전위 리스트, 후위 리스트, 그리고 방문 집합을 생성한다. [그림 9]의 경우 우선 시작 블록에 해당하는 label_16을 루트로 설정한다. 이를 위해 number (root, visited)메소드를 호출한다. 이 메소드는 방문 집합에 루트를 추가하는 동작을 수행한다. 또한 이 메소드는 재귀적으로 호출되면서 현재 방문한 노드가 기존의 방문 집합에 있는가를 확인한 후 존재하지 않는다면 추가하는 동작을 수행하여 현재 방문한 노드가 이전에 방문한 적인 있는가에 대한 정보를 나타낸다. 각 리스트가 만들어지는 동작은 다음과 같다. 우선 현재 루트에 해당하는 label_16을 방문 집합에 추가하고, 전위를 설정하기 위해 해당 인덱스를 0으로 초기화 한다. 전위 리스트에 현재 노드를 추가한다. 그 후 succ(node) 메소드에 의해 현재 노드의 후행자들의 정보를 가져온다. [그림 9]에서 보면 현재 노드인 label_16의 후행자는 label_17과 label_18이다. 이들 중에 하나의 노드인 label_18을 선택한 후 이 노드가 방문 집합에 존재하는지를 확인한 후 만약 존재하지 않는다면 number(succ, visited) 메소드를 호출하여 후행자에 해당하는 노드를 방문하여 방문한 노드를 방문 집합에 추가하고 방문한 노드에 대해 인

덱스를 1만큼 추가시킨 후 전위 리스트에 추가한다. 여기서 인덱스는 전위로 추가되는 순서를 나타낸다. 현재 방문된 노드는 label_18이 된다. 이 노드에 대한 후행자는 존재하지 않기 때문에 더 이상 깊이 우선 탐색을 수행할 수 없다. 이 경우 현재 노드를 후위 리스트에 추가한다. 여기까지 수행된 후 방문 집합에는 {label_16, label_18}, 전위 리스트에는 [label_16, label_18], 후위 리스트에는 [label_18]이 추가된 상태가 된다. 계속해서 모든 노드에 대한 방문이 끝난 후 방문 집합은 {label_16, label_18, label_17, label_0, label_6, label_13, label_11}, 전위 리스트는 [label_16, label_18, label_17, label_0, label_6, label_13, label_11], 후위 리스트는 [label_18, label_13, label_6, label_11, label_0, label_17, label_16]이 된다.

4. 도달 불가능 블록 제거

기본 블록을 통한 CFG를 살펴보면 도달 불가능한 블록이 존재하는 경우가 있다. [그림 9]의 경우 label_15가 해당한다. label_15는 기본 블록을 생성하기 위해 추가된 내용이기 때문에 실제 코드와는 아무런 관계가 없다. 이런 경우 해당 블록을 제거해야 한다. 도달 불가능한 블록을 판단할 수 있는 방법이 존재해야 하는데 이를 위해 앞에서 리스트를 생성할 때 각 블록에 인덱스를 값을 설정하였다. 처음에 노드를 생성할 때 모든 블록은 인덱스 값을 -1값을 갖도록 하였다. 따라서 도달 불가능한 블록을 판단하기 위해서는 블록의 인덱스 값이 음수 값을 갖는가를 확인하는 것이다. 만약 블록의 인덱스 값이 음수를 갖는다면 이 블록은 방문 집합에 존재하지 않을 뿐더러 전위 리스트에도 존재하지 않는다는 의미이다. 즉, 현재 CFG를 통해서는 도달 불가능한 블록이라는 의미이다. 이러한 도달 불가능한 블록은 쓰레기이기 때문에 제거해 주어야 한다. [그림 10]은 도달 불가능한 블록을 제거한 모습이다.

[그림 10]과 [그림 9]를 비교해 보면 [그림 9]에서 아무런 간선에 대한 연결이 존재하지 않았던 label_15의 기본 블록이 [그림 10]에서는 제거되었다. 도달 불가능

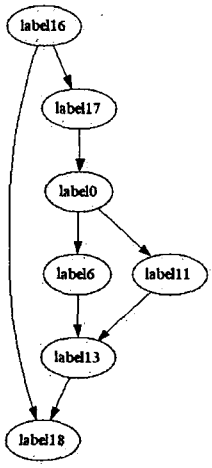


그림 10. 도달 불가능한 블록을 제거한 모습

한 블록을 제거한 후 최종적으로 작성된 CFG는 [그림 11]과 같다.

```
⟨block label_16⟩
    label_16

⟨block label_17⟩
    label_17
        INIT Local_ref0_0 Locali1_1
        goto label_0

⟨block label_0⟩
    label_0
        eval (Locali2_2 := 1)
    label_2
        if0 (Locali1_undef == 0) then ⟨block label_11⟩ else
            ⟨block label_6⟩

⟨block label_6⟩
    label_6
        eval (Locali2_6 := 2)
        goto label_13

⟨block label_11⟩
    label_11
        eval (Locali2_4 := 3)
        goto label_13

⟨block label_13⟩
    label_13
        return Locali2_undef

⟨block label_18⟩
    label_18
```

그림 11. 생성된 제어 흐름 그래프 내용

[그림 11]은 최종적으로 생성된 CFG를 보여준다. <block label_16>은 기본 블록을 표현하고 기본 블록안의 코드는 트리 구성이 가능한 3-주소 형태로 변경된 문장으로 표현하였다.

Ⅳ. 실 험

실험은 펜티엄 4 2.4GHz, 메모리 512MB를 가진 PC에서 수행하였으며, 사용한 소프트웨어는 CTOC-BR 작성과 테스트를 위해 자바 IDE인 eclipse 3.0을 사용하였고, 바이트코드 출력을 위해 editplus 2.11 버전을 사용하였다. 자바 컴파일러는 j2sdk1.4.2_03을 사용하였다.

예제 프로그램은 실험 결과의 비교를 위해 제어 흐름을 살펴볼 수 있는 6가지 경우를 분석하였다. 이 데이터들은 실험 결과의 비교를 위해 Don Lance의 논문에서 사용한 것을 이용하였다[4]. [표 4]는 실험에 사용될 프로그램에 대한 간단한 설명이다.

표 4. 사용 예제와 간단한 설명

프로그램	설명
SquareRoot	숫자의 제곱근 찾기
SumOfSquareRoots	주어진 숫자 n에 대해 1부터 n까지 제곱근의 합 구하기
Fibonacci	주어진 숫자 n에 대해 피보나치 숫자인 Fn 찾기
BubbleSort	버블 정렬을 이용하여 정수 배열 정렬하기
LabelExample	라벨화된 break와 continue 프로그램
Exceptional	try-catch-finally 예외처리

[표 4]의 예제를 이용해서 실험한 항목은 각 프로그램의 원시 소스의 라인 수, 바이트코드의 라인 수, 코드 변경 후 라인 수, 기본 블록 수, 간선의 수, 전체 노드의 수 등이다. [표 5]는 실험 결과이다.

[표 5]에서 소스(no.)는 소스 코드의 라인 수를 의미하고, 바이트코드(no.)는 javap -c를 이용하여 생성된 바이트코드의 라인 수를 의미한다. 변경 후(no.)는 CTOC를 통해 기본 블록을 생성하기 위해 기존의 코드를 변경하는 과정에서 추가되거나 삭제된 후의 코드 라인 수를 의미한다. 기본 블록(ea)은 변경된 코드와 기본 블록을 위한 리더를 통해 생성된 기본 블록의 수를 의미한다. 간선(ea)은 기본 블록과 다른 기본 블록 사이의 관계를 표현하기 위해 사용된 간선의 수를 의미한다. 노드(ea)는 기본 블록 내에 명령어와 문장을 인식하기 위해 사용된 노드의 개수를 의미한다.

표 5. 실험 결과

	소스(no.)	바이트코드(no.)	변경 후(no.)
SquareRoot	37	94	71
SumOfSquareRoot	38	103	77
Fibonacci	42	76	54
BubbleSort	30	79	68
LableExample	28	51	38
Exceptional	41	99	76
	기본 블록(ea)	간선(ea)	노드(ea)
SquareRoot	15	18	117
SumOfSquareRoot	18	19	129
Fibonacci	18	22	108
BubbleSort	16	21	117
LableExample	13	16	70
Exceptional	26	29	197

[표 6]에서는 기존에 작성된 CFG를 작성하는 도구인 SOOT, JAristotle와 CTOC를 비교하였다. 비교는 기본 블록의 수와 간선의 수에 대해서 수행하였다.

표 6. CTOC와 다른 도구들의 비교

프로그램	도구	기본 블록(ea)	간선(ea)
SquareRoot	CTOC	15	18
	Soot	8	10
	JAristotle	17	19
SumofSquareRoot	CTOC	18	19
	Soot	8	10
	JAristotle	17	19
Fibonacci	CTOC	18	22
	Soot	9	12
	JAristotle	19	22
BubbleSort	CTOC	16	21
	Soot	11	14
	JAristotle	15	18
LabelExample	CTOC	13	16
	Soot	11	14
	JAristotle	9	12
Exceptional	CTOC	26	29
	Soot	23	28
	JAristotle	24	27

[표 6]에서 CTOC가 다른 도구들 보다 많은 노드와 간선을 갖는다는 것의 의미는 다른 도구들 보다 좀더 미세 단위의 처리가 가능하다는 의미이다. 따라서 기존의 노드에 좀 더 세밀한 정보의 추가에 의해 더 나은 분석과 최적화를 적용할 수 있다는 것을 의미한다.

V. 결 론

CTOC는 자바 바이트코드 수준의 최적화와 분석을 위해 현재 개발 중인 프레임 워크 이름이다. 본 논문에서 다루는 CTOC의 CTOC-BR은 스택 기반인 바이트코드의 최적화와 분석을 쉽게 하기 위해 트리 형태로 변환을 수행하였다.

바이트코드 수준에서 분석을 수행하기 위해서는 우선 CFG를 생성해야 한다. 따라서 본 논문에서는 자바 바이트코드의 특성을 고려하여 기존의 제어 흐름 분석 기술을 자바 바이트코드에 적합하게 확장하였다. 그리고 분석을 위해, 자바 바이트코드의 정보를 이용하여 CFG를 생성하였다. 또한 기본 블록을 생성하고 기본 블록간의 관계를 이용하여 최적화 과정에서 사용되는 각종 리스트를 생성하였다. 생성된 그래프는 자바 바이트코드의 이해와 유지보수를 위해 테스팅과 복잡성 측정, 그리고 데이터 흐름 분석과 의존성 분석과 같은 다른 분석을 위해서 사용될 수 있는 자료가 되었다.

본 논문에서는 자바 바이트코드를 위해 제어 흐름을 분석하고 최적화를 위한 준비로 탐색을 위한 리스트와 도달 불가능한 기본 블록에 대한 제거를 수행하였다. 마지막으로 CTOC와 기존의 도구들 사이에 비교를 수행하였다. CTOC에서 좀더 많은 기본 블록과 노드가 발생하는 이유는 CTOC에서는 미세 단위의 분석을 수행할 수 있도록 구성되었기 때문이다.

향후 계획으로는 현재 개발된 CFG를 최적화와 분석을 위해 정적 단일 배정 형태로 변환하고, 타입에 대한 추론을 수행하여 좀 더 많은 정보를 이용하여 프로그램의 분석과 최적화를 수행하는 것이다.

참 고 문 헌

[1] T. Linholm and F. Yellin, *The Java Virtual Machine Specification*, The Java Series, Addison Wesley, Reading, MA, USA, Jan., 1997.

[2] J. Gosling, B. Joy, and G. Steel, *The Java Language Specification*, The Java Series, Addison Wesely, 1997.

[3] T. Shpeismans and M. Tikir, "Generating Efficient Stack Code for Java," Technical report, University of Maryland, 1999.

[4] http://www.mtsu.edu/~java

[5] http://www.cs.purdue.edu/s3/projects/bloat

[6] http://www.sable.mcgill.ca/soot

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

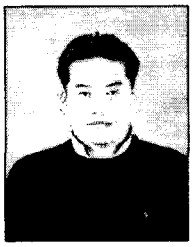
[8] A. W. Appel, *Modern Compiler Implementation in Java.* CAMBRIDGE UNIVERSITY PRESS, pp.437-477, 1998.

[9] 김기태, 유원희, "CTOC에서 3-주소 코드를 위한 정적 타입 추론", 제 22회 한국정보처리학회 추계 발표대회 논문집, 제11권, 제2호, pp.437-440, 2004(11).

[10] 김기태, 이갑래, 유원희, "CTOC에서 정적 단일 배정문 형태를 이용한 지역 변수 분리", 한국콘텐츠학회 논문지, 제5권, 제3호, pp.73-81, 2005(6).

저 자 소 개

김 기 태(Ki-Tae Kim) 정회원



- 1999년 2월 : 상지대학교 전산학과(이학사)
- 2001년 2월 : 인하대학교 전자계산공학과(공학석사)
- 2001년 3월~현재 : 인하대학교 전자계산공학과(박사수료)
- 2004년 3월~현재 : 인하대학교 컴퓨터 공학부 강의전임 강사

<관심분야> : 컴파일러, 프로그래밍 언어, 정보보안

유 원 희(Weon-Hee Yoo) 정회원

- 1975년 2월 : 서울대학교 응용수학과(이학사)
- 1978년 2월 : 서울대학교 대학원 계산학(이학석사)
- 1985년 2월 : 서울대학교 대학원 계산학(이학박사)
- 1979년~현재 : 인하대학교 컴퓨터 공학부 교수

<관심분야> : 컴파일러, 프로그래밍 언어, 병렬시스템