DOI: 10.3745/KIPSTD.2008.15-D.2.197

CTOC에서 루프 트리 구성하기

김 기 태[†] · 김 제 민[††] · 유 원 희[†††]

요 약

최근 많이 사용되고 있는 자바 바이트코드의 분석과 최적화를 효율적으로 수행하기 위해 CTOC 프레임워크가 구현되었다. CTOC에서는 바이트코드에 대해 분석과 최적화를 수행하기 위해 가장 먼저 eCFG를 생성하였다. 분석하기 어렵다는 바이트코드의 특성 때문에 기존의 바이트코드를 제어 흐름 분석에 적합하게 확장하여 제어 흐름 그래프를 작성하였다. 이를 확장된 제어 흐름 그래프인 eCFG라 부른다. 또한 정적으로 분석하기 위해 eCFG를 SSA Form으로 변환 하였다. 변환 시 많은 프로그램에서 루프가 발견 되었다. 이전 CTOC에서는 루프에 대한 처리를 수행하지 않은 상태에서 직접 SSA Form으로 변환을 수행하였다. 하지만 SSA Form으로 변환하기 이전에 루프에 대한 부분을 처리하면 더욱 효율적인 SSA Form을 생성할 수 있게 된다.

본 논문은 루프에 대한 처리를 효율적으로 하기 위해 eCFG를 SSA Form으로 변환하는 과정 이전에 루프를 발견하고 이와 관련된 루프 트리를 생성하는 과정을 보인다.

키워드 : 자바 바이트코드, 제어 흐름 그래프, 정적 단일 배정 형태, 루프. 루프 트리

Constructing A Loop Tree in CTOC

Kim Ki Tae[†] · Kim Je Min[††] · Yoo Weon Hee[†††]

Abstract

The CTOC framework was implemented to efficiently perform analysis and optimization of the Java bytecode that is often being used lately. In order to analyze and optimize the bytecode from the CTOC, the eCFG was first generated. Due to the bytecode characteristics of difficult analysis, the existing bytecode was expanded to be suitable for control flow analysis, and the control flow graph was drawn. We called eCFG(extended Control Flow Graph). Furthermore, the eCFG was converted into the SSA Form for a static analysis. Many loops were found in the conversion program. The previous CTOC performed conversion directly into the SSA Form without processing the loops. However, processing the loops prior to the SSA Form conversion allows more efficient generation of the SSA Form.

This paper examines the process of finding the loops prior to converting the eCFG into the SSA Form in order to efficiently process the loops, and exhibits the procedures for generating the loop tree.

Key Wards : Java Bytecodes, Control Flow Graph, Static Single Assignment Form, Loops, Loop Tree

1. 서 론

바이트코드는 스택 기반 코드이다. 따라서 수행 속도가 느리고 프로그램 분석이나 최적화에 적절한 표현은 아니다[1, 2]. 바이트코드의 분석과 최적화를 위해서 *CTOC*가 개발되었다[3, 4, 5, 6]. *CTOC*는 기존의 바이트코드에 정보를 추가하여 분석을 용이하게 하고, 트리 형태의 중간 표현을 사용하여 최적화를 수행하는 프레임워크이다.

*CTOC*는 바이트코드의 제어 흐름 분석을 수행하기 위해 기존의 제어 흐름 분석 기술을 자바 바이트코드에 적합하게 확장하였다. 바이트코드 수준의 분석을 위해 우선 *eCFG*(확장된 제어 흐름 그래프)를 생성하였다[3]. *eCFG*는 유향 그래프(*directed graph*)이며 기본 블록 집합에 대한 제어 흐름 정보를 표현한다. *CTOC*의 *eCFG*는 일반적인 *CFG*에서 추가되는 시작 블록, 종료 블록뿐만 아니라 초기화 정보를 유지하기 위한 초기화 블록을 가진 확장된 *CFG*를 의미한다. *eCFG*를 생성한 후 데이터 흐름 분석과 최적화를 위해 변수가 어디서 정의되고 어디서 사용되는지에 대한 정보가 요구되었다. 이들 정보를 정의(*def*)와 사용(*use*)이라고 하는데 전통적인 컴파일러에서는 정의-사용 체인(*du_chain*)으로 정의와 사용에 대한 정보를 유지하였다[7, 8, 9]. 하지만 변수를 정적으로 다루기 위해서는 변수들을 정의와 사용에 따라 분리해야 한다. 왜냐하면 동일한 변수라도 정의와 사용에

† 정 회 원 : 인하대학교 컴퓨터공학부 강의전임강사
†† 준 회 원 : 인하대학교 정보공학과 박사과정
††† 종신회원 : 인하대학교 컴퓨터공학부 교수
논문접수 : 2007년 5월 8일, 심사완료 : 2007년 10월 13일

따라 다른 위치에서 다른 값과 다른 타입을 가질 수 있기 때문이다. 따라서 *CTOC*는 정적으로 값과 타입을 결정하기 위해 변수를 배정되는 것에 따라 분리하는 과정을 수행한다. 이를 위해 *CTOC*는 기존의 정의-사용 체인 대신 *SSA Form*(*Static Single Assignment Form*)을 사용하였다.

이전 연구에서는 루프에 대한 식별 없이 단순히 *eCFG*를 생성한 후 *SSA Form*로 변환하였다[4]. 하지만 프로그램에서 수행 중에 많은 시간이 루프를 수행하는데 소비된다. 따라서 루프는 최적화를 구현하는데 아주 중요한 부분 중에 하나이다. 비록 루프의 외부 코드가 증가된다 할지라도 내부 루프의 명령어를 줄일 수 있다면 프로그램의 수행시간을 크게 줄일 수 있게 된다[7, 9]. 따라서 프로그램 내에서 루프를 발견하는 것은 중요한 작업이다. 또한 *SSA Form*으로 변환한 후에 루프에 관한 처리를 수행하는 것보다 *SSA Form* 변환 전에 루프에 관한 처리가 수행되는 것이 더욱 효율적이다. 왜냐하면 루프를 처리하는 과정에서 코드가 옮겨지거나 새로운 변수를 생성하는 등의 동작에 의해 기존의 *eCFG*에 변화가 발생할 수 있기 때문이다. 또한 변경된 *eCFG*를 *SSA Form*으로 변환하는 것이 *SSA Form* 변환 후 생성된 중간 코드를 이용해서 루프 처리를 수행하는 것보다 쉽기 때문이다.

루프 또는 루프 트리를 발견하기 위한 대표적인 관련 연구로는 *Tarjan*의 인터벌 찾기 알고리즘이 있다[10]. 이것은 줄일 수 있는(*reducible*) 그래프로 제한되는 전통적인 루프 찾기 방법이었다. 반면 *Havlak*은 *Tarjan*의 알고리즘을 확장하여 줄일 수 있는 그래프뿐만 아니라 줄일 수 없는(*irreducible*) 그래프까지 처리하는 방법을 제안 하였다[11]. *Sreedhar-Gao-Lee*는 루프 중첩 숲(*loop-nesting forest*)을 생성하기 위해 *Havlak*과는 다른 방법을 제안 하였다[12]. 그들은 *DJ* 그래프라 불리는 새로운 그래프를 사용하였다. *DJ* 그래프는 *CFG*와 지배자 트리를 하나의 구조로 다루는 그래프이다. 또 다른 연구로 *Steensgaard*는 그래프에서 루프 식별을 위해 하향식 방식을 사용하고, 외부 루프를 제일 먼저 찾는 방식을 사용하였다[13].

*CTOC*에서 루프의 식별과 루프 트리의 생성은 *Havlak*의 방법과 *Steensgaard*의 방법을 기반으로 작성한다. 또한 생성된 루프와 루프 트리를 확인하기 위해 *eCFG*와 루프 트리를 그려 *CTOC*의 처리 결과를 확인한다. 본 논문은 루프 최적화보다는 루프의 발견과 루프 트리 생성과정에 중점을 둔다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구로 루프의 정의와 줄일 수 있는 루프와 줄일 수 없는 루프에 대해서 설명한다. 3장은 루프 식별을 위해 사용할 간단한 예제와 생성된 제어 흐름 그래프 정보 그리고 생성된 *eCFG*를 보인다. 4장에서는 루프를 식별하는 과정과 루프 트리를 생성하는 과정을 기술한다. 5장에서는 결론과 향후 계획을 제시한다. 마지막에 부록을 첨가하여 생성된 다른 루프 트리의 예를 보인다.

2. 관련연구

2.1 루프

프로그램의 실행 속도를 높이기 위해서는 다양한 제어 흐름 그래프에서 루프에 대한 식별이 요구된다. 흐름 그래프에서 루프라면 일반적으로 다음과 같은 노드들의 집합으로 정의된다. 첫째, 루프 안의 모든 노드는 강하게 연결(*strongly connected*)되어 있다. 즉, 루프의 어느 한 노드에서 다른 어떤 노드로든 갈 수 있는 경로가 존재한다. 둘째, 루프는 오직 하나의 입구(*entry*)가 존재한다. 루프 밖에서 루프 안으로 도달하기 위해서 항상 먼저 입구를 통과해야 하고 이러한 입구는 오직 하나이다.

2.2 줄일 수 있는 루프와 줄일 수 없는 루프

구조적이지 않은 프로그램에서 루프는 하나 이상의 입구를 가지는 경우가 존재할 수도 있다. 따라서 루프는 줄일 수 있는 루프와 줄일 수 없는 루프로 구분할 수 있게 된다.

줄일 수 있는 루프(*reducible loop*)의 정의는 다음과 같다. 줄일 수 있는 흐름 그래프란 루프 밖에서 중간으로 들어오는 점프가 없는 경우이고, 루프의 유일한 입구는 헤더뿐인 경우이다. 만약 흐름 그래프를 줄일 수 있다면 다음의 조건을 만족해야한다. 첫째, 줄일 수 있는 흐름 그래프의 연결선은 순차 간선(*forward edges*)과 역 간선(*back edges*)의 두 그룹으로 나누어지고, 두 그룹의 공통부분은 없다고 가정한다. 둘째, 순차 간선 $a \rightarrow b$는 비 순환적이다. 또한 그래프의 시작 노드로부터 모든 노드에 도달 가능해야한다. 셋째, 역 간선은 순차간선이 $a \rightarrow b$의 형태일 때 간선에서 b가 머리(*head*)이고 a가 꼬리(*tail*)가 되는 간선으로 구성되지는 경우를 의미한다.

흐름 그래프가 줄일 수 있다는 의미는 남아있는 간선이 모두 순차 간선이라는 의미이다. 줄일 수 있는 흐름 그래프에서 루프라고 간주되는 노드들의 집합은 역 간선을 가져야 한다. 따라서 역 간선의 자연 루프(*natural loop*) 검사가 요구된다[7]. 이것은 흐름 그래프가 줄일 수 있는 프로그램 내의 모든 루프를 찾기 위한 방법 중 하나이다.

일반적으로 흐름 그래프에서 깊이 우선 탐색 순서와 지배관계를 이해하면 역 간선을 쉽게 찾을 수 있고, 또한 이를 제거할 수도 있다. 만약 그 흐름 그래프가 줄일 수 있는 경우라면 남아있는 간선은 순차 간선이 된다. 따라서 남아있는 순차 간선이 비순환 그래프인가를 체크하면 그 흐름 그래프가 줄을 수 있는지 아닌지를 판단할 수 있게 된다. 대부분의 언어에서 *goto*를 사용하지 않는 한 줄일 수 있는 흐름 그래프를 형성할 수 있다.

반면 줄일 수 없는 그래프(*irreducible loop*)의 정의는 다음과 같다. 줄일 수 없는 흐름 그래프란 루프 밖에서 중간으로 들어오는 점프가 존재하는 경우이고, 루프의 입구가 여러 개 존재하는 경우이다. 실제 구조적인 프로그램에서는 거의 발생하지 않는다. 따라서 줄일 수 없는 루프 판별에 대한 부분은 본 논문에서는 고려하지 않는다.

```
public void loop() {
    int n = 5;
    int x = 0;
    for (int a=0; a<n; a++)
        for (int b=0; b<n; b++)
            x++;
}
```

```
public void loop();
  Code:
    0:    iconst_5
    1:    istore_1
    2:    iconst_0
    3:    istore_2
    4:    iconst_0
    5:    istore_3
    6:    iload_3
    7:    iload_1
    8:    if_icmpge    35
    11:   iconst_0
    12:   istore       4
    14:   iload        4
    16:   iload_1
    17:   if_icmpge    29
    20:   iinc         2, 1
    23:   iinc         4, 1
    26:   goto         14
    29:   iinc         3, 1
    32:   goto         6
    35:   return
```

(그림 1) (a) 중첩된 루프 예제와 (b) 역 어셈블 된 바이트코드

3. eCFG

*CTOC*에서 루프 식별과 루프 트리 생성을 위해 (그림 1)과 같은 중첩된 루프 예제 프로그램을 사용한다.

(그림 1)(a)는 중첩된 루프 예제이고 (그림 1)(b)는 그림 1(a)를 *javap -c* 옵션을 통해 생성한 역 어셈블된 바이트코드이다. *CTOC*에서 입력은 (a)의 소스 형태가 아니라 (b)의 바이트코드 형태로 받아들인다. 실제 *CTOC*에서는 역 어셈블된 형태가 아닌 바이너리 형태의 *.class* 파일을 입력으로 사용한다. 여기서는 이해를 돕기위해 역 어셈블된 바이트코드를 나타냈었다.

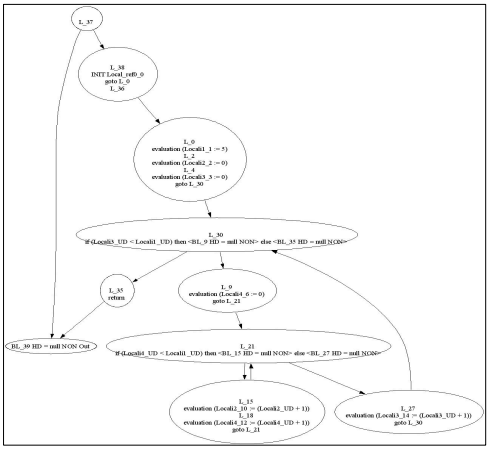
(그림 2)는 (그림 1)(b)의 바이트코드를 *CTOC*에서 분석하기 용이한 형태로 변환한 후 블록과 각 블록내의 내용을 나타낸 것이다.

(그림 2)에서 *BL*은 블록을 의미하고 *BL_n*에서 *n*에 해당하는 숫자는 블록의 번호를 의미한다. 이 블록의 숫자들은 라벨 정보를 이용해 생성된다. *HD*는 헤더 정보를 유지하기 위해 추가된 필드이다. *eCFG*를 생성하는 자세한 사항은 [3]을 참조하면 된다. 이 정보를 이용해 작성된 *eCFG*의 모습은 (그림 3)과 같다.

(그림 3)에 생성된 *eCFG*는 (그림 2)의 블록과 블록 내용에 관한 정보와 블록들 간의 관계를 이용하여 생성된 그래프이다. 각 블록의 이름은 첫 번째 나타나는 레이블의 이름과 동일하기 때문에 따로 표현하지는 않았다. <*BL_37*>, <*BL_38*> 그리고 <*BL_39*>는 각각 시작, 초기화 그리고 종료 블록을 나타내고 있다. 작성된 *eCFG*는 각 블록이 가지는 선행자(*predecessor*)와 후행자(*successor*)의 정보를 이용하여 생성한 것이다.

```
<BL_37 HD = null NON In>
    L_37

<BL_38 HD = null NON INIT>
    L_38
        INIT Local_ref0_0
        goto L_0

<BL_0 HD = null NON>
    L_0
        evaluation (Locali1_1 := 5)
    L_2
        evaluation (Locali2_2 := 0)
    L_4
        evaluation (Locali3_3 := 0)
        goto L_30

<BL_9 HD = null NON>
    L_9
        evaluation (Locali4_6 := 0)
        goto L_21

<BL_15 HD = null NON>
    L_15
        evaluation (Locali2_10 := (Locali2_UD + 1))
    L_18
        evaluation (Locali4_12 := (Locali4_UD + 1))
        goto L_21

<BL_21 HD = null NON>
    L_21
        if (Locali4_UD < Locali1_UD) then <BL_15 HD = null NON>
else <BL_27 HD = null NON>

<BL_27 HD = null NON>
    L_27
        evaluation (Locali3_14 := (Locali3_UD + 1))
        goto L_30

<BL_30 HD = null NON>
    L_30
        if (Locali3_UD < Locali1_UD) then <BL_9 HD = null NON> else
<BL_35 HD = null NON>

<BL_35 HD = null NON>
    L_35
        return

<BL_39 HD = null NON Out>
    L_39
```

(그림 2) 제어 흐름 그래프 정보



(그림 3) eCFG

4. 루프 트리 찾기

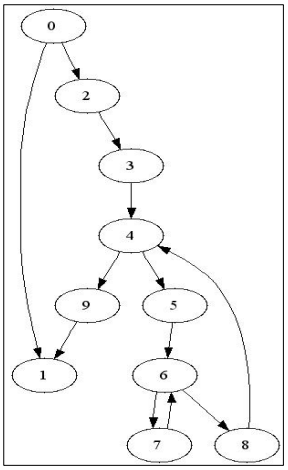
4.1 기본 정보 설정하기

루프 트리를 생성하기 위해서는 깊이 우선 탐색을 통해 방문한 노드에 대한 전위 순서와 후위 순서, 그리고 방문한 노드에 대한 정보들을 획득한다. (그림 4)는 *eCFG*에 대해 깊이 우선 탐색이 수행되는 경우 생성된 정보 중 전위순서(*preorder*)를 그래프로 표현한 것이다.

(그림 4)에서는 블록 이름 대신 간단히 방문되어지는 순서로 노드를 나타냈다. 해당 노드를 보면 방문되어지는 순서를 확인할 수 있다. 이때 하나의 노드는 *eCFG*에서 하나의 기본 블록을 의미하고 실제 구현에서는 생성되는 순서와 방문되는 순서에 따라 다른 형태의 *eCFG*와 다른 전위 방문 순서가 존재할 수 있지만, 루프를 발견하거나 지배관계를 고려할 때, 특별한 영향을 미치지는 않는다. 이렇게 생성된 정보를 이용하여 루프 트리를 생성하게 된다. 루프 트리를 생성하기 위해서는 모든 노드를 앞에서 생성한 전위 순서로 방문한다.

(그림 3)에서 보듯이 시작 블록 <*BL_37*>과 종료 블록 <*BL_39*>을 제외한 경우, 각각의 기본 블록에는 선행자와 후행자가 존재한다. 시작 블록은 선행자가 존재하지 않으며, 종료 블록은 후행자가 존재하지 않는다. *CTOC*에는 선행자 정보만을 이용하여 루프의 헤더를 결정한다. 선행자 정보는 깊이 우선 탐색 후 두 가지로 나눌 수 있는데, 하나는 선행자가 자신보다 작은 전위 순서를 가진 경우이고 또 다른 하나는 선행자가 자신보다 큰 전위 순서를 가진 경우이다. 루프를 식별하기 위해서는 역 간선을 알아야 하기 때문에 이러한 선행자의 정보는 아주 중요하다.

순 방향의 선행자인 *FP*(*Forward Predecessor*)와 역 방향의 선행자인 *BP*(*back Predecessor*)를 집합 배열로 설정한 후 현재 존재하는 블록의 크기만큼 생성한다. 이때, *FP*는 현재 노드가 선행자 노드보다 깊이 우선 탐색 순서가 작은 순 방향 선행자 집합을 나타낸다. 반면 *BP*는 현재 노드가 선행자 노드보다 깊이 우선 탐색 순서가 큰 역 방향 선행자 집합을 나타낸다.



(그림 4) DFS

블록을 하나씩 방문하면서 각 블록에 대한 초기화 동작을 수행한다. 우선 모든 블록에 대해서 각 블록이 가지는 *FP*와 *BP*를 *null*로 초기에 설정한다. 그리고 모든 블록의 헤더를 시작 블록으로 설정한다. 하지만 시작 블록은 선행자를 갖지 않기 때문에 헤더를 시작 블록의 헤더는 *null*로 설정한다.

각 블록에 대해 *FP*와 *BP*를 설정하는 과정으로는 몇 가지 경우가 존재한다. 예를 들어, 첫 번째 경우, (그림 3)에서 <*BL_37*>의 경우 시작 블록이기 때문에 이 블록의 선행자는 없다. *CTOC*는 선행자에 대해서만 고려하기 때문에 특별한 동작이 일어나지 않는다. 두 번째 경우, <*BL_39*>의 경우는 종료 블록이다. 이 블록의 선행자는 시작 블록인 <*BL_37*>과 <*BL_ 35*>가 해당한다. 선행자와 현재 블록의 관계를 확인하는 메소드인 ***isBackEdge(w, v)***를 통해 현재 블록 *w*의 선행자인 *v*가 조상인가를 확인하여, *v*가 조상인 경우는 자손에서 조상으로 역 간선이 존재하는 경우이기 때문에 *BP*에 해당 블록에 대한 정보를 저장한다. 역 간선이 아닌 경우라면 *FP*에 해당 선행자를 추가한다. 즉 각 블록별로 순차 간선과 역 간선의 정보를 수집하게 되는 것이다. 따라서 종료 블록인 <*BL_39*>의 경우에는 시작 블록 <*BL_37*>과 바로 앞의 <*BL_35*>이 종료 블록 <*BL_39*>의 *FP*에 저장된다. 반면 *BP*는 존재하지 않기 때문에 여전히 *null*이 설정된다. 세 번째의 경우는 두 가지 값이 다 설정되는 경우이다. <*BL_30*>의 경우 순차 간선도 존재하고 역 간선도 존재하는 경우이다. <*BL_30*>의 선행자는 <*BL_0*>과 <*BL_27*>이다. 이중에 <*BL_0*>는 <*BL_30*>의 조상으로 존재하는 경우이기 때문에 이전과 같이 *FP*에 추가되지만, <*BL_27*>의 경우는 <*BL_0*>보다 깊이 우선 탐색에서 늦게 나타나는 경우이기 때문에 역 선행자에 해당한다. 따라서 BP에 추가된다. 따라서 <*BL_30*>을 수행한 후 *FP*는 <*BL_0*>가 설정되고 *BP*에는 <*BL_27*>이 설정 된다. 위와 같은 과정을 모두 수행한 후 각 블록의 *FP*와 *BP*의 내용은 <표 1>과 같다.

<표 1>에서 **굵게** 표현된 부분을 살펴보면, 전위 순서 0인 <*BL_37*>의 경우 시작 노드이기 때문에 순차 선행자도

〈표 1〉 각 블록별 FP와 BP의 내용

preorder	Block	FP	BP	header
0	**BL_37**	**ø**	**ø**	**null**
1	**BL_39**	**{BL_35, BL_37}**	**ø**	**BL_37**
2	BL_38	{BL_37}	ø	BL_37
3	BL_0	{BL_38}	ø	BL_37
4	**BL_30**	**{BL_0}**	**{BL_27}**	**BL_37**
5	BL_9	{BL_30}	ø	BL_37
6	**BL_21**	**{BL_9}**	**{BL_15}**	**BL_37**
7	BL_15	{BL_21}	ø	BL_37
8	BL_27	{BL_21}	ø	BL_37
9	BL_35	{BL_30}	ø	BL_37

역 선행자도 존재하지 않는다. 또한 시작 되는 부분이기 때문에 헤더도 *null*로 설정된다. <*BL_39*>의 경우는 종료 블록이기 때문에 시작 블록으로부터 오는 간선과 모든 수행 후 종료 간선으로 오는 두 개의 선행자가 존재한다는 것을 확인 할 수 있다. 마지막으로 블록 <*BL_30*>과 <*BL_21*>은 순차 선행자와 역 선행자를 가진 경우를 보인다. 그리고 이 시점에서 시작 블록을 제외한 모든 블록은 초기화 동작에 의해 시작 블록 <*BL_37*>을 헤더로 갖게 된다.

4.2 루프 헤더 설정하기

각 블록별로 초기화 동작으로 선행자에 대한 정보를 획득한 후 제어 흐름 그래프에 루프가 존재하는지 확인하기 위해서는 전위 순서의 역으로 정보를 찾아야 한다. 이는 각 내부 루프의 헤더 정보를 외부 루프의 헤더 정보보다 먼저 찾아내기 위해서이다. <알고리즘 1>은 루프에서 헤더를 설정하는 알고리즘이다. 이 알고리즘은 *Havlak*과 *Steensgaard*의 알고리즘과 유사하다[11, 13].

만약 제어 흐름 그래프 내에 루프가 존재하는 경우라면 <알고리즘 1>을 사용하여 루프의 헤더를 설정 한다. 이를 위해 각 블록의 *FP* 정보와 *BP* 정보가 필요하다. 우선 블

〈알고리즘 1〉 루프 헤더 설정 알고리즘

```
Input : node ∈ Set of Node
Output : node ∈ Set of Node
Procedure setHeader()
begin
    for each node w ∈ reverse preorder
        body ← {}
        for each node v ∈BP[w]
            if (v not equal w)
                vn ← preOrderIndex(v)
                f ← findNode(vn)
                add f to body
            fi
        endfor
        if (body equal {})
            continue;
        fi
        worklist ← body
        while (worklist not equal {})
            select a node x ∈ worklist and delete from worklist
            for each node y ∈ FP[x]
                z ← findNode(y)
                if(z ∈ body) and (z not equal w)
                    add z to body and worklist
                fi
            endfor
        endwhile
    endfor
    for each node x ∈ body
        header[x] ← w
        union(x, w)
    endfor
end
```

〈알고리즘2〉 findNode 알고리즘

```
Input : num ∈ int
Output : node ∈ Node
Procedure findNode(int num)
begin
    node ← nodes.get(num);
    if(node equal null)
        root ← new Node(num);
        root.child ← new Node(num);
        root.child.parent ← root;
        nodes.set(num, root.child);
        return root;
    fi
    return findNode(node);
end

Input : node ∈ Node
Output: node ∈ Node
Procedure findNode(Node node)
begin
    create stack
    while(node.parent .child equal null)
        stack push node;
        node ← node.parent ;
    endwhile
    rootChild ← node;
    while(stack not empty)
        node ← stack pop
        node.parent ← rootChild;
    endwhile
    return rootChild.parent
end
```

록을 전위 순서의 역 순으로 불러온다. 고려되는 부분은 해당 블록에 *BP*의 정보가 존재하는 경우이면서 현재 블록과 현재 블록의 역 선행자가 같지 않은 경우라면, 메소드 ***findNode()***를 수행하고 이 때 찾아낸 블록을 루프 몸체에 해당하는 *body*에 추가한다. 예를 들어 <표 1>을 살펴보면 블록 내에 *BP*가 존재하는 블록은 <*BL_30*>과 <*BL_21*>의 경우이다. 전위 순서의 역순으로 정보를 찾기 때문에 <*BL_21*>를 먼저 처리하게 된다. <*BL_21*>의 경우를 보면, *FP*로 <*BL_9*>가 존재하고 *BP*로 <*BL_15*>가 존재한다. 이 때, 관심의 대상은 단지 역 선행자를 의미하는 *BP*이다. 지금의 예제에서는 *BP*가 하나 존재하지만 경우에 따라서는 여러 개가 존재할 수 있다. 현재 블록 <*BL_21*>의 *BP*가 <*BL_15*>일 때 현재 블록과 역 선행자가 같은가를 확인한다. 만약 같은 경우라면 자신으로의 루프에 해당한다. 하지만 지금의 경우엔 서로 다른 경우이기 때문에 역 선행자 블록의 전위 순서 인덱스에 의해 *findNode* 알고리즘을 적용한다. <알고리즘 2>는 *findNode* 알고리즘을 나타낸다.

알고리즘 *findNode*는 해당 블록이 속한 블록을 찾는다. <알고리즘 2>는 *findNode*()메소드에 노드의 전위 순서가 들어오는 경우와 직접 노드가 들어오는 두 가지 경우에 대

〈알고리즘3〉 UNION 알고리즘

```
Input : a, b ∈ int
Output: union, node ∈ Node
Procedure union(int a, int b)
begin
    nodeA ← findNode(a);
    nodeB ← findNode(b);
    if(nodeA equal nodeB){
        return;
    fi
    if(nodeA.height > nodeB.height ){
        nodeB.child .parent ← nodeA.child
        nodeA.value ← b
    else
        nodeA.child .parent ← nodeB.child
        nodeB.value ← b
        if(nodeA.height equal nodeB.height ){
            nodeB.height ++;
        fi
    fi
end
```

한 알고리즘이다. 다시 예제로 돌아가서 해당 블록이 속한 블록을 찾기 위해 ***Block f = findNode (vn)***을 수행한다. 이를 좀 더 자세히 살펴보면, 만약 역 선행자 노드인 <*BL_15*>의 전위 순서 번호가 7이라면 *findNode*(7)을 수행하고, 수행 후에 반환되는 블록은 <*BL_15*>이 된다. 이 블록을 루프 몸체에 해당하는 *body*에 추가한다. 이때, *body*의 정보는 [<*BL_15*>]가 된다. 작업리스트인 *worklist*는 연결리스트로 생성하는데 앞에서 생성한 *body* 정보를 이용하게 된다. 따라서 이 동작까지 *worklist* = [<*BL_15*>]이 된다.

*worklist*에 원소가 존재하면 *worklist*의 원소 중 하나를 가져온다. *worklist*에는 현재 <BL_15> 밖에 없기 때문에 *worklist*에서 해당 블록을 제거한 후 그 블록에 대한 전위 순서 인덱스를 가져오고, 다시 해당 인덱스에 해당하는 *FP*를 찾는다. <*BL_15*>의 경우 전위 순서가 *7*에 해당하고 이것의 *FP*는 현재 블록인 <*BL_21*>이 된다. 이후 해당 블록이 속한 곳을 다시 *findNode*를 통해 찾는다. 이렇게 해서 찾아진 블록은 <*BL_21*>이 된다.

〈표 2〉 각 블록별 변경된 FP, BP와 Header 의 내용

preorder	Block	FP	BP	Header
0	BL_37	∅	∅	null
1	BL_39	{BL_35, BL_37}	∅	BL_37
2	BL_38	{BL_37}	∅	BL_37
3	BL_0	{BL_38}	∅	BL_37
4	BL_30	{BL_0}	{BL_27}	BL_37
5	**BL_9**	**{BL_30}**	**∅**	**BL_30**
6	**BL_21**	**{BL_9}**	**{BL_15}**	**BL_30**
7	**BL_15**	**{BL_21}**	**∅**	**BL_21**
8	**BL_27**	**{BL_21}**	**∅**	**BL_30**
9	BL_35	{BL_30}	∅	BL_37

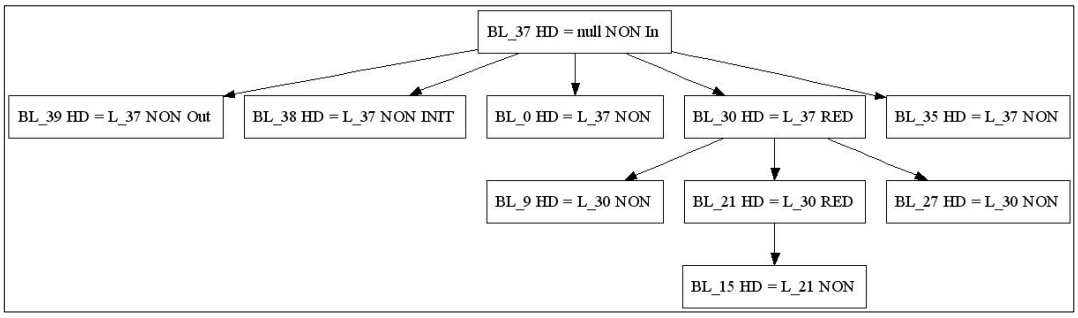
만약 루프의 몸체가 해당 블록을 포함하지 않고 *findNode* 통해 찾은 블록과 현재 블록이 같지 않다면 찾아낸 블록을 다시 루프 몸체에 추가하고 역시 *worklist*에도 찾아낸 블록을 추가한다.

이후에 하나의 루프로 정보를 유지하기 위해서는 해당 루프에 해당하는 것끼리 병합해야 한다. 이를 위해서는 <알고리즘 3>의 *UNION* 알고리즘을 사용하였다.

<알고리즘 3>은 *UNION* 알고리즘을 나타낸다. 이 알고리즘은 하나의 루프에 속하는 노드들에 대해 헤더에 해당하는 노드를 기준으로 트리를 생성한다. 따라서 각 노드의 정보 중 *height* 정보를 이용하여 루프 트리 생성에 이용한다. 이때, *height* 정보는 루프의 중첩 정도를 나타낸다. 다시 예제로 돌아가서 <알고리즘 1>에 대해 (그림 3)의 예제는 루프 몸체에 남아있는 블록 *x*를 <*BL_15*>라고 하고 현재 블록 *w*를 <*BL_21*>라고 했을 때, *x*에 대해 *w*를 헤더로 설정하게 된다. 즉 해당 블록의 진입점이 *w*가 된다는 의미이다. 따라서 이 동작을 수행하고 나면 <*BL_15*>의 헤더는 <*BL_21*>이된다. <표 1>에 있는 전위 순서 인덱스 *6*과 *7*은 <알고리즘 3>에 의해 병합되어진다. 즉 하나의 루프로 인식되어진다.

다음 *BP*가 존재하는 경우는 <*BL_30*>인데 위와 마찬가지 동작을 수행한다. 동작이 수행된 후 *FP*, *BP*, 그리고 헤더의 정보는 <표 2>와 같이 변경된다.

<표 2>의 내용을 바탕으로 루프 트리를 그리면 (그림 5)와 같다.



(그림 5) 루프 트리

(그림 5)의 루프 트리를 보면 아래쪽에서부터 살펴보면 <*BL_15*>의 경우 헤더로 <*BL_21*>을 가진다. 이것의 의미는 <*BL_15*>와 <*BL_21*>이 하나의 내부 루프를 이룬다는 의미이다. 하지만 <BL_21>의 경우엔 헤더로 <*BL_30*>을 가진다. 또한 이것의 의미는 <*BL_30*>, <*BL_9*>, <*BL_21*>, <*BL_27*>이 또 다른 루프를 구성한다는 의미이다. 현재 루프 트리를 보면 깊이가 3인 것을 확인할 수 있다. 즉, 2개의 중첩된 루프를 가지며 특히 트리의 레벨은 내부 루프와 외부루프를 나타내고 이 정보를 확인하면 루프의 개수와 중첩된 정보를 확인할 수 있게 되어 추후 수행할 루프 최적화와 분석에 중요한 정보로 사용될 수 있게 된다.

5. 결론 및 향후 계획

*CTOC*에서는 바이트코드를 적절한 형태로 변경한 후, 변경된 코드에 대해 제어 흐름 분석을 수행하였다. 이때 사용되는 *CTOC*는 기존의 바이트코드에 정보를 추가하여 분석을 용이하게 하고 트리 형태의 중간 표현을 이용하여 최적화를 수행하는 프레임워크이다.

이전 연구에서는 루프에 대한 식별 없이 단순히 *eCFG*를 생성한 후 *SSA Form*을 생성하였다. 하지만 프로그램에서 수행 중에 많은 시간이 루프를 수행하는데 소비된다. 비록 루프에서 외부 코드가 증가된다 할지라도 내부 루프의 명령어를 줄일 수 있다면 프로그램의 수행시간을 크게 줄일 수 있게 된다. 따라서 프로그램 내에서 루프를 발견하는 것은 중요한 작업이라고 할 수 있다.

따라서 본 논문에서는 기존의 *eCFG*에서 루프를 찾는 방법을 보였다. 기존의 *eCFG*에서 루프를 찾기 위해서 제일 먼저 깊이 우선 탐색을 통해 방문한 노드에 대한 전위 순서와 후위 순서, 그리고 방문한 노드에 대한 정보들을 획득하였다. 이후 각 블록별로 순차 선행자와 역 선행자에 대한 정보를 *FP*와 *BP* 집합을 통해서 수집하였다. 루프 헤더를 설정하는 과정에서 *BP*를 이용하여 그래프에 루프가 존재하는지를 확인할 수 있었다. 그리고 루프가 존재하는지가 확인된 후에는 관련된 루프를 묶는 과정에서 각 루프의 헤더를 설정하였다. 이후 헤더 정보를 이용하여 루프 트리를 생성하였다. 루프 트리를 확인하면 현재의 *eCFG*에 몇 개의 루프가 존재하며 또한 어느 정도 중첩되었는지 확인할 수 있었다. 이들 루프와 루프 트리 정보를 이용하면 추후에 루프에서 적용 가능한 최적화와 분석을 좀 더 효율적으로 적용할 수 있게 된다. 추후 연구에서는 루프 트리를 이용하여 *CTOC*에서의 루프 최적화와 분석을 적용할 것이다.

참 고 문 헌

[1] Tim Linholm and Frank Yellin, 'The Java Virtual Machine Specification', The Java Series, Addison Wesley, Reading, MA, USA, Jan, 1997

[2] James Gosling, Bill Joy, and Guy Steel, 'The Java Language Specification' The Java Series, Addison Wesely, 1997

[3] 김기태, 유원희, "CTOC에서 자바 바이트코드를 이용한 제어 흐름 분석에 관한 연구", 한국콘텐츠학회 논문지 제6권 제1호, pp. 160-169, 2006

[4] 김기태, 유원희, "CTOC에서 자바 바이트코드를 위한 정적 단일 배정 형태", 정보처리학회논문지D 제13-D권 제 7호, pp. 939-946, 2006

[5] 김기태, 유원희, "정적 단일 배정 형태를 위한 정적 타입 배정에 관한 연구", 한국콘텐츠학회 논문지 제6권 제2호, pp. 117-126, 2006

[6] 김기태, 김지민, 김제민, 유원희, "CTOC에서 자바 바이트코드 최적화를 위한 Value Numbering", 한국컴퓨터정보학회논문지, 11권6호, pp. 19-26, 2006

[7] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers Principles, Techniques and Tools, Addison Wesley, 1986

[8] Andrew W. appel, Modern Compiler Implementation in Java. CAMBRIDGE UNIVERSITY PRESS, pp. 437-477, 1998

[9] Muchnick, S. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, San Francisco. 1997.

[10] Tarjan, R. E. Testing flow graph reducibility. J. Comput. Syst. Sci. 9, 355-365, 1974

[11] Havlak, P. Nesting of reducible and irreducible loops. ACM Trans. Program. Lang. Syst. 19, 4, 557-567, 1997

[12] Sreedhar, V. C., Cao, G. R., and Lee, Y. F. Identifying loops using DJ graphs. ACM Trans. Program. Lang. Syst. 18, 6, 649-658, 1996

[13] Steensgaard, B. Sequentializing program dependence graphs for irreducible programs. Tech. Rep. MSR-TR-93-14, Microsoft Research, Redmond, Wash. 1993

[14] http://www.graphviz.org/

부 록

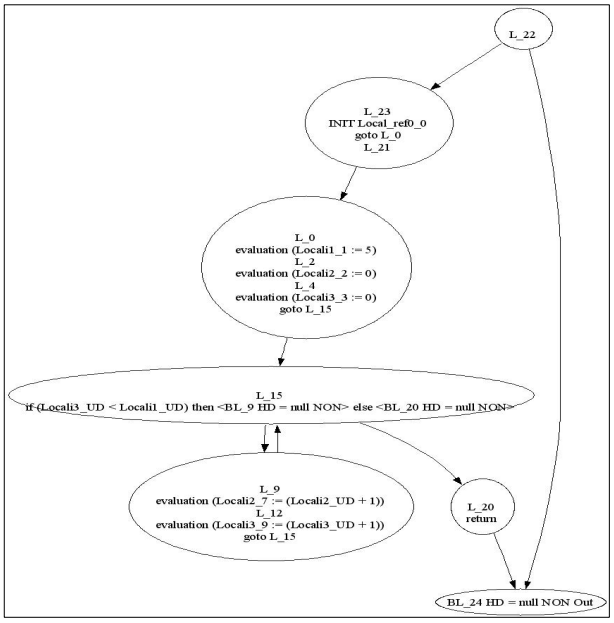
실험은 펜티엄 4 2.4GHz, 메모리 512MB를 가진 PC에서 수행하였으며, 사용한 소프트웨어는 *CTOC* 작성과 소스 테스트를 위해 자바 IDE인 eclipse 3.2를 사용하였고, 바이트코드 출력을 위해 editplus 2.11 버전을 사용하였다. 자바 컴파일러는 jdk1.5.0_09을 사용하였다. 또한 eCFG와 루프 트리 출력을 위해서 오픈 소스인 Graphviz를 사용하였다[14]. *CTOC*를 통해 필요한 정보를 .dot 파일로 생성한 후 결과를 .jpg로 생성하였다.

```
public class NestedLoop1 {
  public void loop() {
    int n = 5;
    int x = 0;
    for (int a=0; a<n; a++)
      x++;
    }
}
```

```
public class NestedLoop2 {
  public void loop() {
    int n = 5;
    int x = 0;
    for (int a=0; a<n; a++)
      for (int b=0; b<n; b++)
        for (int c=0; c<n; c++)
          x++;
  }
}
```

```
public class NestedLoop3 {
  public void loop() {
    int n = 5;
    int x = 0;
    for (int a=0; a<n; a++)
      for (int b=0; b<n; b++)
        x++;
    for (int c=0; c<n; c++)
      x++;
  }
}
```

(그림 6) 예제 (1) 단일 루프 (2) 3중 루프 (3) 2중 루프와 단일 루프



(그림 7) 예제 (1)에 대한 eCFG

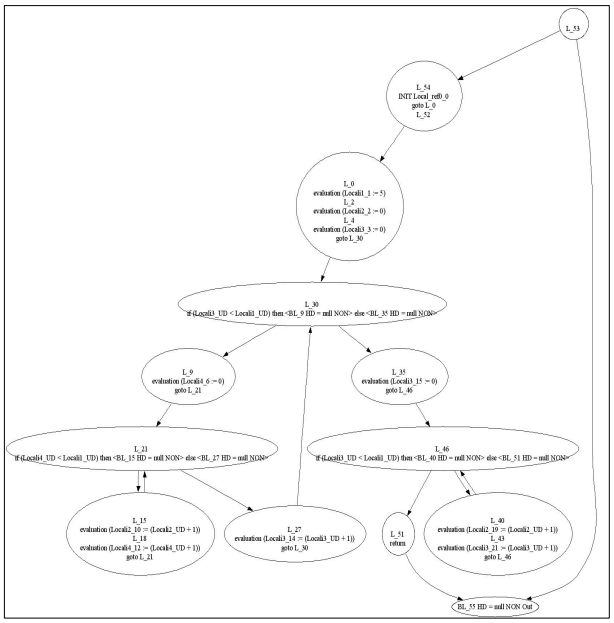
(그림 8) 예제 (2)에 대한 eCFG

테스트를 위한 예제는 다음의 세 가지이다. 첫 번째 경우는 프로그램 내에 단순한 루프가 존재하는 경우이고 두 번째는 본문의 예제 보다 한 단계 깊은 중첩이 되어 있는 3중 루프인 경우, 그리고 마지막은 프로그램 내에 서로 다른 루프가 존재하는 경우를 확인하였다.

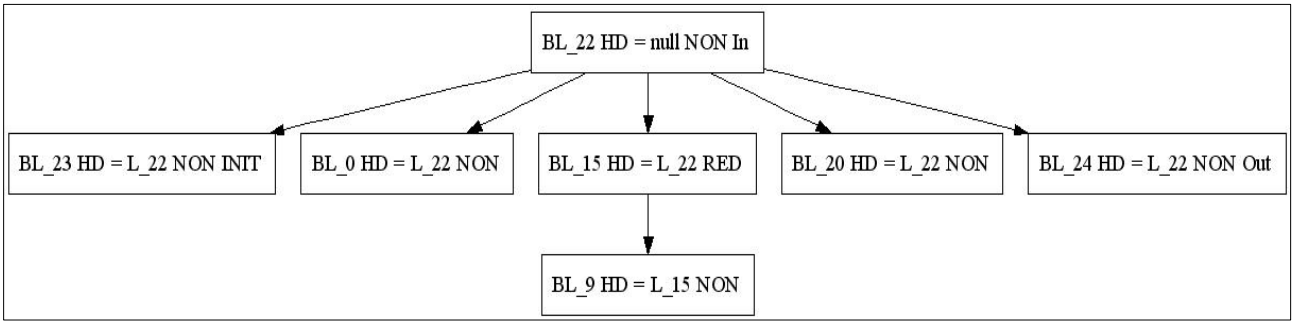
(그림 6)은 (1) 단일 루프, (2) 3중 루프, (3) 2중 루프와 단일 루프에 대한 자바 소스이다.

(그림 7), (그림 8), (그림 9)는 각각 예제 (1), (2), (3)에 대한 *eCFG*를 나타낸 것이다. 그림을 보면 루프 구조에 대해 전반적으로 파악할 수 있지만, 만약 더욱 복잡한 경우라면 그래프의 형태에 따라 이해하기 어려운 경우도 발생할 수 있다.

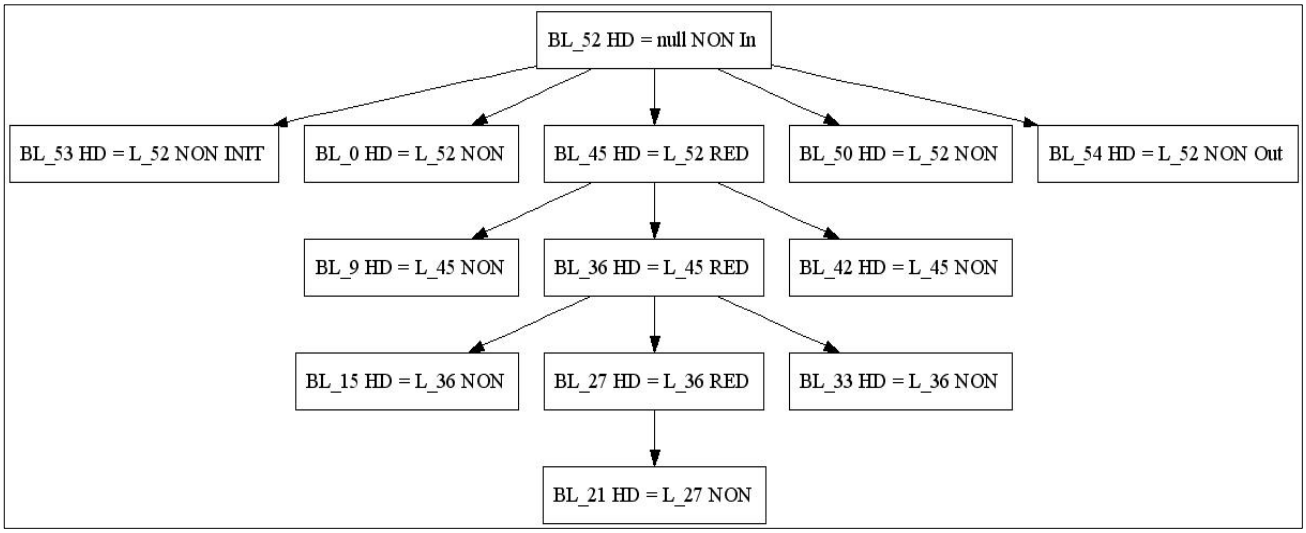
(그림 10), (그림 11), (그림 12)는 예제 (1), (2), (3)에 대한 루프 트리를 표현하고 있다. 루프 트리를 살펴보면 *eCFG*보다 쉽게 루프들의 관계를 이해할 수 있다. 특히 (그림 12)의 경우 <*BL_30*>, <*BL_9*>, <*BL_21*>, <*BL_27*>로 이루어진 외부 루프와 <*BL_21*>, <*BL_15*>로 이루어진 내부 루프 그리고 <*BL_46*>, <*BL_40*>로 이루어진 또 다른 내부 루프에 대해 쉽게 확인할 수 있다.



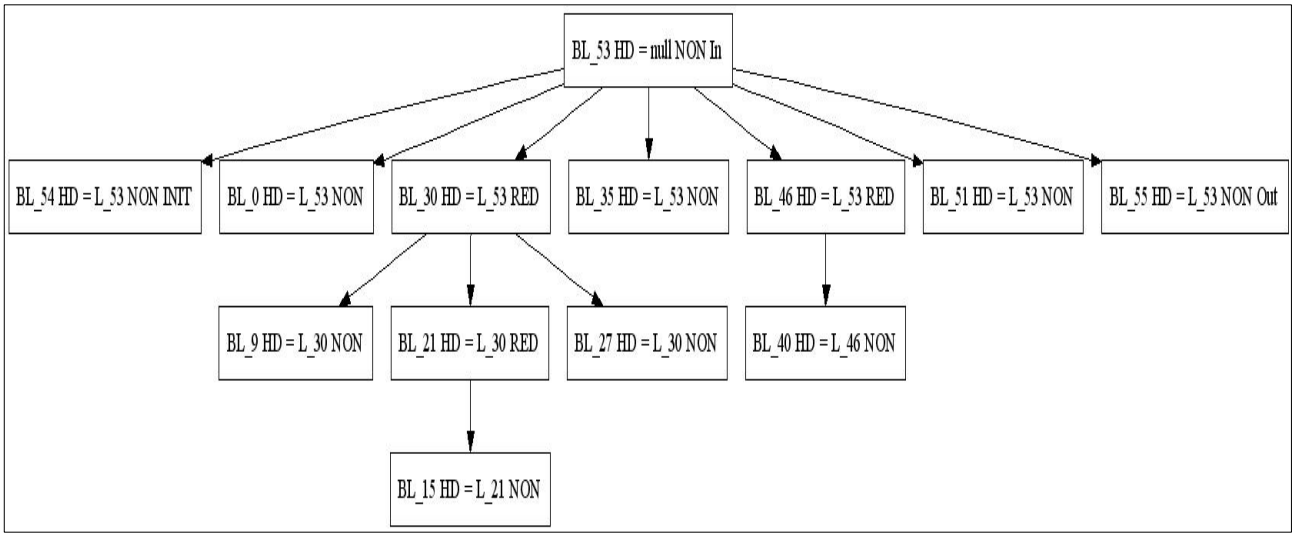
(그림 9) 예제 (3)에 대한 eCFG



(그림 10) 예제 (1)에 대한 루프 트리



(그림 11) 예제 (2)에 대한 루프 트리



(그림 12) 예제 (3)에 대한 루프 트리

김 기 태

e-mail : kkt@inha.ac.kr
1999년 상지대학교 전자계산학과(학사)
2001년 인하대학교 전자계산공학과 (공학석사)
2003년 인하대학교 전자계산공학과 (공학박사수료)
2004년~현 재 인하대학교 컴퓨터공학부 강의전임강사
관심분야 : 컴파일러, 프로그래밍언어, 정보보안

김 제 민

e-mail : jeminya@naver.com
2006년 인하대학교 컴퓨터공학부(공학사)
2006년~2008년 인하대학교 컴퓨터정보 공학과(공학석사)
2008년~현 재 인하대학교 정보공학과 박사과정
관심분야 : 컴파일러, 프로그래밍언어, 정보보안

유 원 희

e-mail : whyoo@inha.ac.kr
1975년 서울대학교 응용수학과(이학사)
1978년 서울대학교 대학원 계산학 (이학석사)
1985년 서울대학교 대학원 계산학 (이학박사)
1979~현 재 인하대학교 컴퓨터공학부 교수
관심분야 : 컴파일러, 프로그래밍언어, 병렬시스템