CTOC에서 정적 단일 배정문 형태를 이용한 지역 변수 분리

Split Local Variables Using Static Single Assignment Form in CTOC

김기태*, 이갑래**, 유원희*
인하대학교 컴퓨터 공학부*, 김천과학대학 컴퓨터정보계열**

Ki-Tae Kim(kimkitae@inha.ac.kr)*, Kab-Lae Lee(kllee@kcs.ac.kr)**,
Weon-Hee Yoo(whyoo@inha.ac.kr)*

요약

CTOC(Classes To Optimized Classes)는 자바 바이트코드의 최적화와 분석을 위해 현재 개발 중인 프레임워크이다. CTOC는 스택 기반인 바이트코드를 최적화와 분석을 쉽게 하기위해 3-주소 형태로 변환한다. 바이트코드가 타입에 관한 정보를 포함지만 스택 기반의 동작을 수행하기 때문에 지역 변수를 위한 명시적인 타입을 갖지 못하는 경우가 발생한다. 따라서 바이트코드에서 3-주소 형태로 변환하는 과정에 사용되는 모든 변수는 정적 타입을 가져야 하는 문제점이 발생한다. 왜냐하면, 프로그램의 최적화나 분석을 위해서는 지역 변수의 타입을 알아야 하기 때문이다.

본 논문은 CTOC 프레임워크의 전반부로 바이트코드를 스택을 사용하지 않는 3-주소 형태로 변환하는 과정을 수행한다. 이 과정에서 스택 코드 형태인 CTOC-B 코드를 생성하고, 제어 흐름 그래프를 생성하여 바이트코드 수준에서 분석을 수행한다. 또한 정적 타입을 제공하기 위한 중간 과정으로 타입을 갖지 않는 CTOC-T 코드를 생성한다. 이를 위해 정적 단일 배정문 형태(Static Single Assignment Form)를 사용하여 모든 변수를 분리하고 각 변수에 새로운 이름으로 재명명하는 동작을 수행한다. 분리된 변수들은 추후 정적 타입 추론을 위한 준비 단계로 사용된다.

■ 중심어 : | CTOC | 3-주소 코드 | 자바 바이트코드 | 정적 단일 배정문 |

Abstract

CTOC(Classes To Optimized Classes) is a Java bytecode framework for optimization and analysis.

Although Java bytecode has a significant amount of type information embedded in it, there are no explicit types for local variables. However, knowing types for local variables is very useful for both program optimization and analysis. This paper is a first part of CTOC framework. In this paper, we present methods for inferring static types for local variables in a 3-address, stackless, representation of Java bytecode. We use SSA Form(Single Static Assignment Form) for spliting local variables. Splited local variables will use to prepare for static type inference.

■ Keyword : | CTOC | 3-address codes | Java Bytecodes | Static Single Assignments |

* 본 연구는 한국과학재단 목적기초연구(R05-2004-000-11694-0)지원으로 수행되었습니다.

접수번호 : #050207-002
접수일자 : 2005년 02월 07일
심사완료일 : 2005년 03월 31일
교신저자 : 김기태, e-mail : kimkitae@inha.ac.kr

I. 서 론

자바 바이트코드는 유용한 특징을 많이 가고 있지만 프로그램 분석이나 최적화를 위한 적절한 표현은 아니다. 또한 스택 기반 코드이기 때문에 수행 속도가 느린 단점이 있다[1]. 그리고 자바 컴파일러는 각 플랫폼에 독립적인 바이트코드 표현에 제약을 받기 때문에 효율적인 코드를 생성하는데 한계가 있다. 또한, 자바 가상 기계에서 동적 링크를 지원하기 위하여 고안된 상수 풀(constant pool)의 크기가 상대적으로 크다는 문제점을 갖는다. 따라서 자바 클래스 파일이 네트워크와 같은 실행 환경에서 효과적으로 실행되기 위해서는 최적화된 코드로 변환이 요구된다.

빠른 실행을 위해 최적화가 요구되는데 Bloat, Jax, 그리고 Joie와 같은 기존 연구는 바이트코드를 사용해서 저수준에서 최적화와 분석을 수행하였다[2]. 하지만 컴파일러에서 사용하는 3-주소 코드로 변환하면 더욱 효율적인 최적화를 수행할 수 있게 된다[3]. 프로그램의 이해와 분석을 위해서도 저수준의 바이트코드를 사용하는 것보다 3-주소 코드를 사용하는 것이 더욱 바람직하다.

이를 위해 우리는 CTOC라 부르는 최적화 프레임워크를 개발 중에 있다[4, 5]. CTOC는 클래스 파일을 입력으로 받아 다양한 분석과 최적화를 수행하는 프레임워크이다. CTOC는 스택 기반 바이트코드를 최적화와 분석을 위해 3-주소 형태로 변환한다. 3-주소에서 사용되는 모든 변수는 정적 타입을 가져야 한다. 하지만 바이트코드의 모든 명령어가 명시적인 타입을 가지는 것은 아니기 때문에 변환 과정에서 정적인 타입에 관한 문제가 발생하게 된다. 이 문제를 해결하기 위해 CTOC에서는 변환 과정에서 생성된 타입 없는 3-주소 코드에 정적 단일 배정 형태(SSA Form: Static Single Assignment Form)를 적용하여 분리 가능한 모든 변수 형태로 나눈 후 각 변수에 새로운 이름을 재명명한다[6]. 올바른 정적 타입을 위해 타입 전파 그래프를 생성하여 타입 추론 과정을 수행하여 CTOC에서 정적 타입 추론의 문제를 해결한다.

본 논문은 변환 과정 중 정적 단일 배정 형태를 이용하여 지역 변수를 나누고, 새로운 이름을 재명명하는 과정에 대해 보인다.

논문의 구성은 2장은 최적화 프레임워크인 CTOC에 관한 전체적인 설명과 클래스 파일에서 CTOC-B코드로의 변환 과정을 보인다. 3장은 타입을 갖지 않는 CTOC-T 코드에서 사용되는 변수에 정적 단일 배정 형태를 적용하여 분리 가능한 변수들로 나누는 과정을 상세히 보인다. 4장에서는 결론과 향후 과제를 이야기한다.

II. 관련 연구

1. CTOC(Classes To Optimized Classes)

CTOC는 현재 개발 중인 최적화 프레임워크의 이름이다. CTOC의 전체적인 구성도는 [그림 1]과 같다.

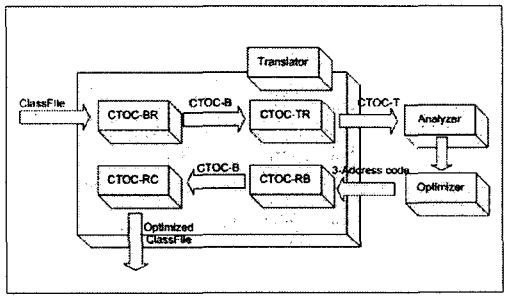

그림 1. CTOC 구성도

[그림 1]에서 보이는 것과 같이 클래스 파일(.class)을 입력 받아 여러 단계 변환 수행한 후 분석과 최적화 과정을 통해 최적화된 클래스 파일을 생성하는 것이 이 프레임워크의 목적이다. 최적화된 클래스 파일을 생성하기 위해 CTOC에서는 클래스 파일을 바이트코드 수준에서 최적화를 적용하고, 3-주소 형태로 변환한 후 다시 최적화를 적용한다. 최적화된 코드로 변환하기 위해 CTOC-BR(CTOC-Bytecode tRanslator), CTOC-TR (CTOC-Three address tRanslator), CTOC-RB (CTOC-Return Bytecode), 그리고 CTOC-RC(CTOC-Return Classfile)와 같은 변환기를 이용한다. 본 논문에서는 3-주소 형태의 변환을 위해 CTOC-BR과 CTOC-TR에 대한 부분을 설명한다. 나머지 부분은 추후에 다시 논한다.

2. 클래스 코드에서 CTOC-B 코드로 변환

CTOC에서 가장 먼저 수행되는 동작은 자바 바이트코드에서 CTOC-B(CTOC-Bytecode)코드로 변환이다. 바이트코드 수준의 분석을 수행하기 위해 기존 바이트코드에 새로운 정보를 추가한 CTOC-B 코드를 생성한다. CTOC-B 코드를 생성하기 위해 CTOC-BR를 이용한다. CTOC-BR의 동작은 [그림 2]와 같다.

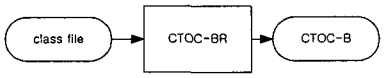

그림 2. CTOC-BR의 동작

[그림 2]를 보면 클래스 파일을 CTOC-BR의 입력으로 받아들여 CTOC-B 코드로 변환한다. 테스트를 위해 CTOC-BR에 입력으로 사용된 자바 소스는 [그림 3]과 같다.

```
public class Test {
    A a;
    boolean condition;
    public int test(){
        A a;
        int sum = 0;
        if(condition){
            int z = 5;
            a = new B();
            sum += z++;
        } else {
            String s = "four";
            a = new C(s);
        }
        return a.value + sum;
    }
}
```

그림 3. 입력으로 사용된 자바 소스

[그림 4]는 CTOC-BR을 통해 생성된 CTOC-B 코드이다. CTOC-B의 특징은 바이트코드와 유사하게 스택에 관련된 정보를 가지고 있지만 3-주소 형태에서 필요한 타입을 위해 각 명령어마다 타입이 주어진 형태이다. 그리고 getfield[Test.condition]처럼 [ ] 안에는 기존의 바이트코드에서는 상수 풀의 위치를 알려주는 #숫자가 적혀있는 것 대신 상수 풀의 내용을 가져와 3-주소로 변

```
public int test() {
        word this, sum, z;
        this := @this : Test;

        iconst_0
        istore_2
        load.r this
        getfield [Test.condition]
        ifeq [label 29]
        ...
        new [B]
        dup
        invokenonvirtual [B.<init>]
        ...
        astore_1

[label 41] load.r a
        getfield [A.value]
        ...
        ireturn
}
```

그림 4. CTOC-B 코드

환 시 다시 상수 풀을 참조하는 일을 줄였다. 명령어 중에 분기와 관련된 명령어를 기본 블록의 리더로 제어 흐름 그래프를 생성하여 바이트코드 수준에서 분석 하였다. [그림 5]는 [그림 4]에 대해 생성된 제어 흐름 그래프이다.

```
메소드 test을 CFG 리스트

기본 블록 리스트 : 0 로부터 6까지의 블록 N1
전 노드(Predecessors) :
후 노드(Successors) : [29] [9]
 0: iconst_0
 1: istore_2
 2: aload_0
 3: getfield [Test.condition]
 6: ifeq [label_29]
 ...

기본 블록 리스트 : 35 로부터 37까지의 블록 N6
전 노드(Predecessors) : [29]
후 노드(Successors) : [40]
 35: dup
 36: aload_3
 37: invokenonvirtual [C.<init>]

 ...
기본 블록 리스트 : 41 로부터 47까지의 블록 N8
전 노드(Predecessors) : [18] [40]
후 노드(Successors) :
 41: aload_1
 42: getfield [A.value]
 45: iload_2
 46: iadd
 47: ireturn
```

그림 5. 제어 흐름 그래프(CFG)

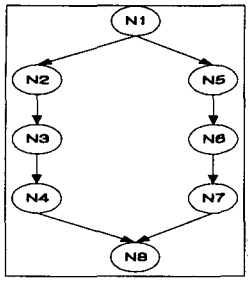

그림 6. 그래프 표현

[그림 5]를 흐름 분석을 위해 그래프 형태로 나타내면 [그림 6]과 같다. [그림 6]에서 분기는 [그림 5]의 6: ifeq [label_29] 부분에서 ifeq 명령에 의한 분기이다. 따라서 분기가 발생하는 부분까지를 리더로 해서 [그림 6]과 같은 그래프가 생성된다. 제어 흐름 그래프는 변수를 나누거나 타입을 위해 다음 부분에서도 사용된다. Ni에 해당하는 것은 각각의 노드 즉 기본 블록을 의미하며, 특히 N1과 같은 노드를 분기 노드라 하고 N8과 같은 노드를 병합 노드라 한다. 이러한 특징은 정적 단일 배정(SSA Form)을 다룰 때 중요한 역할을 한다.

III. 타입이 없는 3-주소 코드로 변환

1. 타입을 가지지 않는 CTOC-T코드로 변환

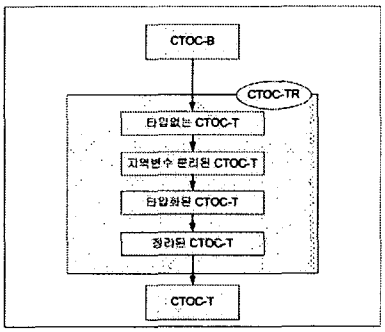

그림 7. CTOC-TR의 동작

[그림 7]과 같이 CTOC-TR은 CTOC-B 코드를 CTOC-T(CTOC-Three address code) 코드로의 변환기이다. CTOC-T코드는 3-주소 형태의 중간코드이다. CTOC-B에서 CTOC-T로 변환은 [그림 7]과 같은 단계를 거친다. 3-주소 형태로 만들기 위해서는 우선 CTOC-B 코드를 타입이 없는 3-주소 형태로 변환한다. 첫 번째 과정을 통해서 생성된 코드는 [그림 8]과 같은 변수에 타입이 결정되지 않은 임시 형태이다.

```
1: public int test(){
2:      untyped this, sum, z, stack$0, stack$1, stack$2;
3:
4:          l0 := @this: Test;
5:          $stack0 = 0;
6:          l2 = $stack0;
7:          $stack0 = l0;
8:          $stack0 = $stack0.condition;
9:          if $stack0 == 0 goto label0;
10:         $stack0 = 5;
11:         l3 = $stack0;
12:         $stack0 = new B;
13:         $stack1 = $stack0;
14:         specialinvoke $stack1.<init>();
15:         l1 = $stack0;
16:         $stack0 = l2;
17:         $stack1 = l3;
18:         l3 = l3 + 1;
19:         $stack0 = $stack0 + $stack1;
20:         l2 = $stack0;
21:         goto label1;
22:
23: label0:
24:         $stack0 = "four";
25:         l3 = $stack0;
26:         $stack0 = new C;
27:         $stack1 = $stack0;
28:         $stack2 = l3;
29:         specialinvoke $stack1.<init>($stack2);
30:         l1 = $stack0;
31:
32: label1:
33:         $stack0 = l1;
34:         $stack0 = $stack0.value;
35:         $stack1 = l2;
36:         $stack0 = $stack0 + $stack1;
37:         return $stack0;
38: }
```

그림 8. untyped 타입을 가진 코드

[그림 8]은 타입을 가진 3-주소 코드로 가기위한 중간 단계이다. 이 단계는 아직 타입이 결정되지 않은 상태이다. 따라서 단순히 untyped라는 임시 타입을 변수들에게 제공한다. 또한 더 이상 스택 코드를 사용하지 않기 위해 스택의 위치와 관련된 임시 스택 변수 생성한다. 임시 스택 변수는 $stack*i* 의 형태이다. $는 스택과 관련된 임시 변수라는 의미를 가지고 있으며, 스택을 의미하는 stack이란 이름을 사용하며 숫자는 스택의 위치를 의미한다. 이러한 스택의 위치는 stackHeight()메소드에 의해 스택 높이를 계산하여 구할 수 있다. [표 1]은 [그림 4]의

CTOC-B코드가 [그림 8]의 임시 CTOC-T코드로 변환되는 모습을 보인다.

표 1. (a) CTOC-B 코드 (b) 임시 CTOC-T 코드

load,r a getfield [A.value] iload a iadd ireturn	33: $stack0 = l1; 34: $stack0 = $stack0.value; 35: $stack1 = l2; 36: $stack0 = $stack0 + $stack1; 37: return $stack0;

2. 정적 단일 배정문 형태를 사용하여 변수 분리

타입 없는 3-주소 코드에 정적인 단일 타입을 주기위해서는 일반적으로 변수들을 정의(define)와 사용(use)에 따라 분리한다. 왜냐하면 동일한 변수라도 정의와 사용에 따라 다른 위치에서 다른 타입과 다른 값을 가질 수 있기 때문이다. 예를 들면 [그림 8]의 코드에서 라인 5:를 보면 $stack0에 처음에는 정수 값 0이 배정되지만 라인 12:에서는 $stack0에 new B가 배정된다. 즉, 각 위치에서 배정되는 값에 따라 동일한 이름의 변수가 다른 위치에서 다른 타입의 값을 가질 수 있게 된다. 따라서 정적으로 타입을 결정하기 위해서는 변수는 배정되는 것에 따라 분리되어야 한다. 이를 위해 본 논문에서는 정적 단일 배정문 형태를 사용한다. 정적 단일 배정문 형태는 일반적으로 데이터 흐름 분석이나 코드 최적화를 위해 컴파일러의 중간 표현으로 사용된다. 정적 단일 배정문 형태는 각 변수들이 프로그램 안에서 한 번씩만 정의되는 것이다. 정적 단일 배정문 형태로 변환에서 가장 중요한 것은 원시 프로그램으로부터 같은 이름의 변수들이 서로 관련이 없을 때 변수 이름을 재명명하는 것이다. 정적 단일 배정문을 구하는 일반적인 순서는 [표 2]와 같다. [표 2]와 같은 과정을 통해서 변수 이름을 재명명한 후에 필요한 경우 최적화를 수행하여 사용되지 않는 변수를 알아내어 죽은 코드 제거를 수행하여 최적화를 구현한다.

표 2. 정적 단일 배정문 형태 변환 과정

단계1:	CFG에서 지배자 트리(dominator tree)를 구한다.
단계2:	지배자 트리와 CFG를 보면서 지배자 경계(DF : Dominator Frontier)를 구한다.
단계3:	지배자 경계를 이용하여 Ø-함수를 삽입한다.
단계4:	지배자 트리를 순행하면서 변수의 이름을 재명명한다.

2.1 지배자 트리 구하기

[표 2]의 변환 과정 중 첫 단계로 앞에서 생성한 제어 흐름 그래프를 통해 지배자 트리(dominator tree)를 먼저 생성한다. 지배자 트리는 흐름 그래프에서 초기 노드로부터 노드 n까지 도달할 때 모든 경로는 d를 거쳐야 한다면, 노드 d는 노드 n을 지배하며 이러한 지배 구조를 트리 형태로 나타낸 것이다. 지배자 트리를 생성하는 순서는 먼저 제어 흐름 그래프에서 깊이 우선 신장 트리(DFST: depth-first spanning tree)를 구한다. 깊이 우선 신장 트리를 이용하여 Aho가 제안한 지배자 계산 알고리즘을 이용하면 손쉽게 지배자 트리를 생성할 수 있다[3]. 본 논문에서는 집합을 이용하는 방법을 사용하여 지배자 트리를 생성하였다[7].

2.2 지배자 경계 구하기

제어 흐름 그래프와 지배자 트리를 생성한 후 Ø-함수 삽입을 위해 지배자 경계(DF: dominance frontier)를 구한다[6]. 반복적인 while 문이나 if문과 같은 구조에서 문장은 흐름 그래프에서 병합(join) 되는 경로가 존재하며 그 위치에 일반적으로 Ø-함수가 삽입된다. 루프가 시작되는 곳에서 Ø-함수를 삽입할 위치를 알 수 있는 방법은 없다. 따라서 모든 노드는 자신이 지배하지 않는 노드의 위치를 가지고 있을 필요가 있다. 이러한 Ø-함수를 삽입할 위치를 지배자 경계라 한다. 즉, 지배자 경계는 흐름 그래프에서 모든 노드 n에 대하여 필요한 Ø-함수를 삽입할 위치를 말한다. 지배자 경계는 DF[n]으로 표현하며 지배자 경계를 구하기 위해서는 2가지 집합을 이용한다.

* DFlocal[n] : 노드 n에 의해 직접 지배되는 않는 n의 앞 노드집합
* DFup[n] : 노드 n의에 의해 간접 지배되는 노드에 의해 지배되지 않는 지배자 경계의 노드 집합

이 두개의 집합으로부터 지배자 경계를 구하는 식은 다음과 같이 표현 한다.

$$DF[n] = DFlocal[n] \cup \bigcup_{c \in children[n]} DFup[c]$$

지배자 경계를 구하는 알고리즘은 [표 3]과 같다.

표 3. 지배자 경계를 구하는 알고리즘

```
입력 : 지배자 트리의 각 노드
출력 : 각 노드 n의 지배자 경계 집합
방법 :
 computeDF[n] =
      S ← { }
(1)   for each node y in succ[n]
        if dominate(y) ≠ n
          S ← S U {y}                    // DFlocal[n] 계산
(2)   for each child c of in the dominator tree
          computeDF[c]
          for each element w of DF[c]
            if n does not dominate w
              S ← S U {w}                // DFup[c] 계산
      DF[n] ← S
```

[표 3]에서 (1)은 현재 노드 n에서의 지배자 경계 집합을 구하는 부분이며, (2)는 현재 노드 n의 자식 노드에 대한 지배자 경계 집합을 구하는 부분이다. [표 3]의 알고리즘을 이용해 구한 DF[n]을 통해서 Ø-함수가 삽입될 위치를 결정한다.

2.3 Ø-함수 삽입

DF[n]을 통해 Ø-함수를 삽입할 위치가 결정되면 삽입할 위치에 Ø-함수를 삽입 한다. 구조적인 언어는 Ø-함수가 삽입되는 위치가 여러 곳에 발생할 수 있다. 왜냐하면 구조적인 언어에서는 특정 위치에 병합 노드가 발생하기 때문에 병합이 이루어지는 곳에 일반적으로 Ø-함수가 삽입 된다. [그림 9]는 구조화된 문장에서 병합 노드의 위치를 보여준다. 검은색으로 표시된 곳에 Ø-함수가 삽입된다. CTOC-T의 구문에서는 IF와 GOTO에 의해 분기와 반복이 발생하기 때문에 아주 간단하게 처리할 수 있다. IF문의 경우 Ø-함수 삽입은 [그림 10]과 같다.

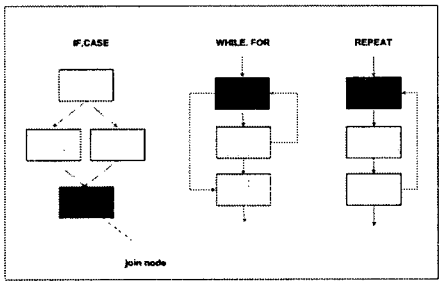

그림 9. 흐름 그래프에서 병합 노드 위치

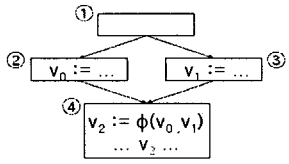

그림 10. Ø-함수 삽입

[그림 10]의 경우 ②, ③에서 v_0, v_1에 대한 배정이 발생하게 되면 아래의 병합 노드에 Ø-함수를 추가하여 v_2 ← Ø(v_0, v_1) 형태로 만들어 이 노드의 다음부터는 v_2 변수를 사용하게 된다. 실제로 이 경우 삽입된 함수는 v ← Ø(v, v) 형태이다. v_2 ← Ø(v_0, v_1) 형태는 재명명이 이루어진 후의 모습이지만 여기서 설명을 돕기 위해 표현하였다. CTOC-TR에서는 프로그램 상에 조건문이 발생할 경우 Ø-함수가 1개 이상 발생할 수 있기 때문에 Ø-함수의 노드를 함수 한 개당 하나씩 만들면 그래프가 복잡해진다. 따라서 하나의 노드에 Ø-함수를 모두 삽입하도록 하였다.

[표 4]는 Ø-함수 삽입 알고리즘이다. [표 4]의 Ø-함수 삽입 알고리즘에서Aorig[n]은 각 노드 n에 들어 있는 좌변수 x이며 defsites[x]의 배열 변수는 좌변수 x를 가지고 있는 노드의 집합을 의미한다. DF[n]는 노드 n의 지배자 경계 집합이며 Aphi[Y] 지배자 경계 노드 Y의 Ø-함수 집합을 의미한다. 즉, Aphi[Y]가 실제로 필요한 출력 결과 값이다.

표 4. Ø-함수 삽입 알고리즘

```
입력: 제어흐름 그래프에서의 각 노드에 대한 모든 변수
출력: Ø-함수가 삽입된 SSA 추상 구문 트리
방법:
Place-Ø-Functions =
   for each variable x in Aorig[n]
      defsites[x] ← defsites[x] U {n}
   for each variable x
      W ← defsites[x]
      while W not empty
         remove some node n from W
         for each Y in DF[n]
           if x not in Aphi[Y]
              insert the statement x ← Ø(x, x, ... , x) at the
              top of block Y, where the Ø-function has
              as many arguments as Y has predecessors
                 Aphi[Y] <- Aphi[Y] U {x}
                 if x not in Aorig[Y]
                    W <- W U {Y}
```

2.4 변수 재명명(rename)

변수 재명명 단계에서는 Ø-함수가 삽입된 그래프와 지배자 트리를 너비우선 방식으로 순행하면서 각 노드의 좌변수를 재명명한다. 이때 좌변수를 재명명하기 위해 각 변수마다 각각의 변수 스택을 두어 좌변수가 재명명될 때마다 각 변수의 스택에 저장하면서 동시에 오른쪽 연산식의 나머지 변수들은 스택에서 수행하면서 상위 노드로부터 이전에 정의되어진 변수 이름을 가지고 오면 변수 재명명 과정이 종료된다. 변수가 모두 재명명 되면 추상 구문 트리(AST) 데이터를 가진다. 재명명하는 알

표 5. 변수 재명명 알고리즘

```
입력: 지배자트리의 각 노드를 너비 우선방식으로 입력
출력: 각 노드의 재 정의된 변수로 변환된 SSA 추상 구문 트리
방법:
// 초기화 부분
    for each variable a
        Count[a] <- 0
        Stack[a] <- empty
        push 0 onto Stack[a]
// 변수 재명명 부분
Rename(n) =
    for each statement S in block n
        if S is not a phi-function
            for each use of some variable x in S
                i <- top(Stack[x])
                replace the use of x with xi in S
            for each definition of some variable a in S
                Count[a] <- Count[a] + 1
                i <- Count[a]
                push i onto Stack[a]
                replace definition of a with definition of
                ai in S
    for each successor Y of block n
        Suppose n is the jth predecessor of Y
            for each phi-function in Y
                suppose the jth operand of the
                phi-function is a
                i <- top(Stack[a])
                replace the jth operand with ai
    for each child X of n
        Rename(X)
    for each definition of some variable a in the original
    S
        pop Stack[a]
```

고리즘은 [표 5]와 같다. [표 5]의 알고리즘은 먼저 각 변수 스택을 초기화 하며 최종적으로 [그림 11]과 같이 지역 변수가 정의와 사용에 따라 나눠진 코드를 생성한다.

```
public int runningExample(){
untyped   I0, $stack0, I2, I3, $stack1, I1, $stack2,
          $stack0#2,$stack0#3,$stack0#4,$stack0#5,
          $stack0#6,$stack1#2,I3#2,$stack0#7,
          $stack0#8,I3#3,$stack0#9,$stack1#3,
          $stack0#10,$stack0#11,$stack1#4,
          $stack0#12;

          I0 := @this: Test;

          $stack0 = 0;
          I2 = $stack0;
          $stack0#2 = I0;
          $stack0#3 = $stack0#2.condition;
          if $stack0#3 == 0 goto label0;
          $stack0#4 = 5;
          I3 = $stack0#4;
          $stack0#5 = new B;
          $stack1 = $stack0#5;
          specialinvoke $stack1.(init)();
          I1 = $stack0#5;
          $stack0#6 = I2;
          $stack1#2 = I3;
          I3#2 = I3 + 1;
          $stack0#7 = $stack0#6 + $stack1#2;
          I2 = $stack0#7;
          goto label1;

label0:
          $stack0#8 = "four";
          I3#3 = $stack0#8;
          $stack0#9 = new C;
          $stack1#3 = $stack0#9;
          $stack2 = I3#3;
          specialinvoke $stack1#3.(init)($stack2);
          I1 = $stack0#9;

label1:
          $stack0#10 = I1;
          $stack0#11 = $stack0#10.value;
          $stack1#4 = I2;
          $stack0#12 = $stack0#11 + $stack1#4;
          return $stack0#12;
}
```

그림 11. 정적 단일 배정이 적용된 CTOC-T코드

[그림 11]에서 보듯이 $stack0#12은 $stack0의 변수가 12개의 서로 다른 변수로 나누어졌다는 것을 의미한다. 즉, $stack0에 대해 12번의 서로 다른 정의가 발생했다는 것이다. 생성된 새로운 변수의 이름은 다음 정의가 발생하기 전까지 유효하다. [그림 11]의 코드에 나타나는 변수들은 아직 타입이 결정되지 않은 상태이다. 따라서 아직 untyped라는 키워드를 포함하고 있다.

3. 분리된 지역 변수

[표 6]을 살펴보면 [표 1] (b)에 SSA를 적용한 후에 변수들이 분리된 모습을 보여준다.

표 6.
(a) 임시 CTOC-T 코드 (b) 분리된 CTOC-T 코드

```
33: $stack0 = l1;
34: $stack0 = $stack0.value;
35: $stack1 = l2;
36: $stack0= $stack0 + $stack1;
37: return $stack0;                    (a)
```

```
33:  $stack0#10 = l1;
34:  $stack0#11 = $stack0#10.value;
35:  $stack1#4 = l2;
36:  $stack0#12=$stack0#11+$stack1#4;
37:  return $stack0#12;                (b)
```

앞의 과정을 통해서 생성된 타입을 가지 않는 CTOC-T는 단순히 자바로부터 생성된 바이트코드가 아닌 여러 다른 언어로부터 생성된 중간 코드일 수 있기 때문에 예상치 못한 문제가 발생할 수 있으며, 추후에 정적 타입을 배정해야 하기 때문에 발생할 수 있는 여러 가지 문제가 존재한다. 일반적으로 바이트코드 검증의 문제, 인터페이스에 의한 다중 상속, 그리고 배열에 대한 타입 문제가 발생할 수 있다. 이들 문제는 정적 타입 추론과 관련된 부분에서 다시 논의한다.

Ⅳ. 결 론

최근 여러 분야에서 자바 바이트코드가 중간 표현으로 사용된다. 하지만 실행 속도가 느리다는 단점이 존재한다. 빠른 실행 속도를 위해서는 코드의 최적화가 요구되는데 스택 코드인 바이트코드를 사용해서 저수준에서 최적화와 분석을 하는 것 보다는 컴파일러에서 사용하는 전통적인 3-주소 형태의 코드로 변환하는 것이 더욱 바람직하다. 따라서 이를 위해 현재 CTOC 프레임워크를 개발 중이다. CTOC는 자바 바이트코드의 최적화와 분석을 위한 프레임워크이다. CTOC에서 스택 기반인 바이트코드는 최적화와 분석을 위해 3-주소 형태로 변환되어야 하고, 사용되는 모든 변수는 올바른 정적 타입을 가져야 하는 문제가 발생하였다.

이를 해결하기 위해서 본 논문에서는 자바 바이트코드를 스택을 사용하지 않는 3-주소 코드 형태로 변환하는 과정을 수행하였다. 이 과정에서 스택 코드 형태인 CTOC-B 코드를 생성하고, 제어 흐름 그래프를 생성하여 바이트코드 수준에서 분석을 수행하였다. 그리고 정적인 타입을 제공하기 위한 중간 과정으로 타입을 가지지 않는 CTOC-T 코드를 생성하였다. 이를 위해 정적 단일 배정 형태를 사용하였으며, 정적 단일 배정 형태를 이용하여 모든 변수를 분리해서 각 변수별로 새로운 이름으로 재명명하는 동작을 수행하였다.

추후 이루어지는 연구에서는 분리된 변수에 명확한 정적인 타입을 제공할 수 있는 타입추론 알고리즘을 제공할 것이며, 3-주소 형태의 다양한 최적화와 분석을 수행할 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] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations", Technical report, McGill University, July, 1998.

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

[4] 김경수, 김기태, 조선문, 유원희, "CTOC에서 스택 기반 코드를 효율적인 중간 코드로 변환기 설계", 제 22회 한국정보처리학회 추계발표대회 논문집 제11권, 제2호, pp.429-432, 2004.

[5] 김기태, 유원희, "CTOC에서 3-주소 코드를 위한 정적 타입 추론", 제22회 한국정보처리학회 추계발표대회 논문집 제11권, 제2호, pp.437-440, 2004.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. Kenneth Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph", pp.451-490, 1991.

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

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

이 갑 래(Kab-Lae Lee) 종신회원

- 1987년 2월 : 인하대학교 전산학과(이학사)
- 1989년 2월 : 인하대학교 전자계산공학과(공학석사)
- 1997년 3월 : 인하대학교 전자계산공학과(박사수료)
- 1993년 9월~현재 : 김천과학대학 교수

<관심분야> : 콘텐츠 보호, 웹프로그래밍, 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, 실시간 시스템, 병렬시스템