정적 단일 배정 형태를 위한 정적 타입 배정에 관한 연구

A Study on Static Type Assignment for Static Single Assignment Form

김기태, 유원희
인하대학교 컴퓨터공학부

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

요약

자바 바이트코드는 많은 장점을 갖지만 수행 속도가 느리고 분석이 어렵다는 단점을 갖는다. 이를 극복하기 위해 바이트코드에 대한 분석과 최적화가 수행되어야 한다. 우선 바이트코드에 대한 제어 흐름 분석을 수행한다. 제어 흐름 분석 후 데이터 흐름 분석과 최적화를 위해서 변수가 어디서 정의되고 어디서 사용되는지에 대한 정보가 필요하다. 각 위치에서 변수에 배정되는 값에 따라 동일한 이름의 변수가 다른 위치에서 다른 값을 가지는 경우가 발생한다. 따라서 정적으로 값과 타입을 결정하기 위해서 변수는 배정되는 것에 따라 분리되어야 한다. 이를 위해 단일 배정 형태를 이용하여 표현할 수 있다. 정적 단일 배정 형태(SSA Form)로 변경한 후 정적 분석과 최적화를 위해서는 각 변수와 표현식이 나타내는 각각의 노드에 타입 정보를 설정해야 한다.

본 논문은 타입에 대한 기본 정보를 바탕으로 관련된 동등한 노드를 발견하고 강 결합 요소로 설정한 후 각 노드에 타입을 효율적으로 설정하는 방법을 제안한다.

■ 중심어 : | 자바 바이트코드 | 제어 흐름 그래프 | 정적 단일 배정 형태 | 정적 타입 배정 |

Abstract

Although the Java bytecode has numerous advantages, there are also shortcomings such as slow execution speed and difficulty in analysis. In order to overcome such disadvantages, bytecode analysis and optimization must be performed. First control flow of the bytecode should be analyzed, after which information is required regarding where the variables are defined and used to conduct data flow analysis and optimization. There may be cases where variables with an identical name contain different values at a different location during the execution according to the value assigned to a variable in each location. Therefore, in order to statically determine the value and type, the variables must be separated according to allocation. In order to do so, the variables can be expressed using a static single assignment form. After the transformation into a static single assignment form, the type information of each node expressed by each variable and expression must be configured to perform static analysis and optimization. Based on the basic type information, this paper proposes a method for finding related equivalent nodes, setting the nodes with strongly connection components and efficiently assigning each node the type.

■ Keyword : | Java Bytecodes | Control Flow Graph | Static Single Assignment Form | Static Type Assignment |

* 본 논문은 정부(교육인적자원부)의 재원으로 한국학술진흥재단의 지원을 받아 수행된 연구입니다.(R05-2004-000-11694-0)

접수번호 : #051121-002
접수일자 : 2005년 11월 21일
심사완료일 : 2006년 01월 17일
교신저자 : 김기태, e-mail : kimkitae@inha.ac.kr

I. 서 론

바이트코드는 많은 유용한 특징을 갖지만 스택 기반 코드이기 때문에 수행 속도가 느리고 프로그램 분석이나 최적화에 적절한 표현은 아니라는 단점을 갖는다[1][2]. 또한 자바 가상 기계에서 동적 링크를 지원하기 위하여 고안된 상수 풀(constant pool)의 크기가 상대적으로 크다는 문제점도 갖는다. 따라서 자바 클래스 파일이 네트워크와 같은 실행환경에서 효과적으로 실행되기 위해서는 최적화된 코드로 변환이 요구된다[3-5].

바이트코드의 분석과 최적화를 위해 가장 먼저 바이트코드에 대한 제어 흐름 분석을 수행한다. 제어 흐름 분석 후 데이터 흐름 분석과 최적화를 위해 변수가 어디서 정의되고, 어디서 사용되는지에 대한 정보가 필요하다. 각 위치에서 변수에 배정되는 값에 따라 동일한 이름의 변수가 다른 위치에서 다른 값을 가질 수 있다. 따라서 정적으로 값과 타입을 결정하기 위해서 변수는 배정에 따라 분리해야 한다[6-8]. 이를 위해 SSA Form(정적 단일 배정 형태)를 사용한다[9]. SSA Form로 변경한 후 정적 분석과 최적화를 위해서는 각 변수와 표현식을 나타내는 각 노드에 타입에 대한 정보를 설정해야 한다. 본 논문은 타입에 대한 기본 정보를 바탕으로 관련된 노드를 발견하고 각 노드에 타입을 설정하는 과정을 설명한다.

본 논문의 구성은 다음과 같다. 2장은 CFG(제어 흐름 그래프) 생성 과정과 블록 내의 트리 표현을 위한 BNF에 대해 설명한다. 3장은 SSA Form의 생성과정에 대해 설명한다. 4장에서는 타입을 설정하는 과정에 대해 기술한다. 5장에서는 간단한 실험을 수행한다. 6장에서는 결론과 향후 계획을 논한다.

II. CFG

바이트코드를 위한 CFG를 생성하기 위해 제일 먼저 해당 명령어를 기본 블록(basic block)으로 나눈다. 이후 관련 기본 블록들을 간선(edge)으로 연결하여 CFG를 완성한다. [그림 1]은 본 논문에서 사용할 예제 소스 프로그램을 나타낸다. 이 예제를 이용해 CFG와 SSA Form, 그리고 타입 설정에 대해 보인다.

```
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 1. (a) 예제 프로그램 (b)바이트코드

[그림 1](a)를 javap -c를 이용하여 역컴파일한 결과는 [그림 1](b)와 같다. [그림 1](b)에서 #9는 상수 풀의 인덱스이고, Method java/lang/Object."<init>":()V 내용을 갖는다. ifeq 11의 의미는 "만약 같은 경우"라면 11번 라벨이 있는 곳으로 분기하라는 바이트코드 명령어이다. 11: 은 명령어의 오프셋을 알려주는 라벨이다.

1. 기본 블록 생성하기

CFG를 생성하기 위해 우선 기본 블록을 생성한다. 기본 블록은 CFG에서 제어 흐름에 영향을 주지 않는 명령어들로 구성된다. 프로그램 내에서 제어 흐름에 영향을 주는 리더(leader)를 찾아 기본 블록을 작성한다[6]. 바이트코드에서의 리더가 될 수 있는 명령어는 [표 1]과 같다.

표 1. 리더로 사용 가능한 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

기본 블록은 첫 문장에 각 블록을 구분할 수 있는 라벨을 위치시키고, 마지막 문장엔 다른 블록으로 제어를 전달하는 분기문을 위치시키는 형태를 갖는다. 라벨이 추가되는 위치는 클래스 파일 내에 존재하는 lineNumbers 배열을 참조한다. lineNumbers 배열 내부에는 라인 번호 테이블에 있던 원시 프로그램의 라인 번호 정보와 시작 위치에 대한 정보가 기록되어 있다. 라벨 정보를 가진 블록을 생성하기 위해, 라벨 정보를 이용하여 새로운 블록을 생성한다.

생성된 CFG는 유향 그래프(directed graph)로 작성되고, 그래프의 각 노드는 기본 블록으로 이루어진다. CFG를 효율적으로 생성하기 위해서는 시작 노드(entry node), 종료 노드(exit node), 그리고 초기화 노드(initial node)를 기본 노드로 추가한다. 시작 노드는 그래프의 진입 지점을 알리기 위해 사용되고, 종료 노드는 그래프의 진출 지점을 알리기 위해 사용된다. 초기화 노드는 그래프에서 사용되는 매개변수와 같은 정보들의 초기화를 위해 사용된다. CFG에서 사용되는 각 블록은 라벨 정보를 포함한다.

2. 트리 형태 구문 설정

블록에 존재하는 문장들은 트리 형태로 추가한다. 이를 위해 적절한 BNF를 정의한다. 사용된 트리를 표현 트리(expression tree)라 하는데 기본 블록 내부에서 각 명령어를 3-주소 형태의 문장으로 표현하기 위해 사용한다. 표현 트리는 커다랗게 추상 클래스인 Expr 클래스로부터 파생된 표현식과 역시 추상 클래스인 Stmt 클래스로부터 파생된 문장으로 나눠진다. [그림 2]는 표현 트리를 구성하는 BNF의 일부를 나타낸다.

[그림 2]의 BNF를 통해 작성된 Expr과 Stmt의 파생 클래스를 생성한다. 이 두 종류의 클래스의 차이는 Expr로부터 파생된 클래스들은 값을 유지할 수 있는 value 필드를 가진다는 것이다. 또한 각 Expr은 타입을 표현하기 위해 type 필드도 가진다. 본 논문에서는 이 type 필드를 이용해 각 노드에 정적인 타입을 설정하고 이를 이용해 관련 노드에 타입을 전파한다. Stmt는 단순히 3-주소 형태의 문장을 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 2. BNF 코드 중 일부

[그림 1](b)의 바이트코드를 분석하고 필요한 정보를 추가하여 CFG를 생성한다. [그림 3]은 생성된 CFG를 나타낸다.

```
⟨block_16⟩
   label_16
⟨block_17⟩
   label_17
      INIT Local_ref0_0 Locali1_1
      goto label_0
⟨block_0⟩
   label_0
      eval (Local_ref2_2 := 1)
   label_2
      if0 (Local_ref1_UDef == 0) then ⟨block_11⟩ else ⟨block_6⟩
⟨block_6⟩
   label_6
      eval (Local_ref2_6 := 2)
      goto label_13
⟨block_11⟩
   label_11
      eval (Local_ref2_4 := 3)
      goto label_13
⟨block_13⟩
   label_13
      return Local_ref2_UDef
⟨block_18⟩
   label_18
```

그림 3. 생성된 CFG

[그림 3]의 CFG는 전체 7개의 기본 블록으로 생성되고 블록 내부의 문장들은 트리 형태로 추가된 상태이다. 이 CFG는 SSA Form으로 변경에서 입력으로 사용된다.

III. SSA Form 구현

일반적으로 변수를 정적으로 다루기 위해서는 변수를 정의(definition)와 타입(type)에 따라 분리해야 한다. 왜냐하면, 동일한 변수일지라도 정의와 사용에 따라 다른 위치에서 다른 값이나 다른 타입을 가질 수 있기 때문이다. 따라서 정적으로 값과 타입을 결정하기 위해서 변수는 배정되는 것에 따라 분리되어야한다. 이를 위해 본 논문에서는 SSA Form를 이용한다. SSA Form로 변환하는 단계는 [표 2]와 같다.

표 2. SSA Form 변환 단계

단계	내용
1단계	CFG에서 각 변수에 대한 정보 획득과 지배자 트리 구하기
2단계	지배자 경계(DF) 구하기
3단계	Ø-함수 추가하기
4단계	변수 재명명하기
5단계	반복적인 지배자 경계(DF+)가 위치하는 블록에 Ø-문장 추가하기

1. CFG에서 변수 정보 수집과 지배자 트리 구하기

SSA Form로 변환하기 위해서는 우선 앞에서 작성된 CFG를 방문하면서 프로그램 내에서 정의 되었거나 사용되는 변수에 대한 정보를 수집해야 한다. 이를 위해 CFG에서 블록별로 해당 문장에 대해 생성된 트리를 깊이 우선(depth first) 방식으로 방문하여 각 블록 내에서 정의(def)되었거나 사용(use)되는 변수를 찾는다. 각 블록을 살펴보면 해당 블록에서 정의된 변수에 대한 정보를 찾을 수 있다. 이러한 정보는 해당 노드가 변수를 다루는 VarExpr 클래스로부터 파생된 문장에 존재한다. 변수에 대한 정보는 블록에 변수가 존재하면 해당 변수를 같은 이름을 가진 변수들과 함께 유지한다.

CFG의 병합(join)지점에 Ø-함수를 삽입하기 위해서는 지배자 경계(DF : Dominance Frontier)를 구해야 한다. DF를 구하기 위해서는 먼저 지배자 트리를 생성해야 한다.

초기 노드는 루트이고 각 노드는 그 자손들을 지배하는 구조를 가진다. CFG에서 초기 노드로부터 노드 n까지 도달할 때 모든 경로가 노드 d를 거쳐야 한다면 노드 d는 n을 지배한다고 한다. 이러한 지배관계를 표현하기 위해 각 블록의 지배 정보를 갖는 배열인 dom[i]를 이용하여 각 블록의 지배자를 구한다. 이때 더 이상 dom[i]에 변화가 없을 때까지 계속 수행하여 각 dom[i]의 값을 구한다. 지배자를 구하기 위해 Aho의 알고리즘을 변형하여 이용하였다[6]. [표 3]은 dom[i]가 생성되는 과정을 나타낸다.

표 3. dom[i]를 구하는 과정

블록	초기화 후	1회 수행 후	2회 수행 후	3회 수행 후
0(16)	{0}	{0}	{0}	{0}
1(17)	{0,1,2,3,4,5,6}	{0,1}	{0,1}	{0,1}
2(0)	{0,1,2,3,4,5,6}	{0,1,2}	{0,1,2}	{0,1,2}
3(6)	{0,1,2,3,4,5,6}	{0,1,2,3}	{0,1,2,3}	{0,1,2,3}
4(13)	{0,1,2,3,4,5,6}	{0,1,2,3,4}	{0,1,2,4}	{0,1,2,4}
5(18)	{0,1,2,3,4,5,6}	{0,5}	{0,5}	{0,5}
6(11)	{0,1,2,3,4,5,6}	{0,1,2,6}	{0,1,2,6}	{0,1,2,6}

[표 3]에서 1회 수행 후 변경된 내용을 굵게 표현하였다. 이 경우 dom[i]에 변화가 발생하였기 때문에 계속 수행한다. 3회 수행 후에는 더 이상 아무런 변화가 없기 때문에 각 블록에 대해 dom[i]의 집합을 구하였다. dom[i]는 지배자에 대한 정보이기 때문에 자기 자신에 대한 정보도 포함한다.

DF를 구하기 위해서는 블록간의 지배 관계를 알아야 한다. 이를 위해 현재 블록에 대한 인덱스를 구한 후, 현재 블록의 직접 지배자 찾기를 수행한다. 직접 지배자란 초기 노드에서 n까지 경로에서 노드 n의 마지막 지배자를 의미한다.

직접 지배자를 구하기 위해서는 그래프의 노드들에 해당하는 블록들을 찾아서 현재 블록의 전위 순서를 변수 i에 가져온다. 현재 i가 루트인가를 확인한 후 루트가 아닌 경우라면 직접 지배자를 찾는다. 직접 지배자를 계산하는 식은 (1)과 같이 현재 블록의 지배자에서 현재 블록의 지배자를 지배하는 것을 제거한 후 현재 블록을 제거한 결과와 같다.

idom := dom(block) - dom(dom(block)) - block (1)

이렇게 결정된 직접 지배자는 [표 4]와 같다.

표 4. 직접 지배자

블록(전위 순서)	16(0)	17(1)	0(2)	6(3)	13(4)	18(5)	11(6)
직접 지배자	∅	0	1	2	2	0	2

<block_16>과 같은 경우는 루트에 해당하기 때문에 루트의 직접 지배자는 존재하지 않기 때문에 ∅로 표시된다. 구해진 idom을 현재 블록의 부모로 설정하여 지배 관계에 대한 정보를 저장한다. [그림 5]는 지배 관계에 대해 생성된 지배자 트리이다.

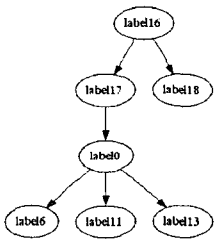

그림 5. 지배자 트리

2. DF 구하기

CFG와 지배자 트리가 생성된 후 ∅-함수 삽입을 위해 DF를 계산한다. 반복적인 while 문이나 if 문과 같은 구조에서 문장은 CFG에서 병합되는 경로가 존재하며 그 위치에 ∅-함수가 삽입된다. 일반적으로 루프가 시작되는 곳에서 ∅-함수가 삽입되는 위치를 알 수 있는 방법은 없다. 따라서 모든 노드는 자신이 지배하지 않는 노드의 위치를 가지고 있을 필요가 발생한다. 이러한 ∅-함수를 삽입할 위치를 DF라고 한다. 즉, DF는 CFG에서 모든 노드 n에 대해서 필요한 ∅-함수를 삽입할 위치를 말한다. DF를 DF[n]으로 표현할 때 (2)와 같은 식으로 표현된다.

$$DF[n] = DF_{local}[n] \cup \bigcup_{c \in children[n]} DF_{up}[c] \qquad (2)$$

(2) 식에서 $DF_{local}[n]$은 노드 n에 의해서 직접 지배되지 않는 n의 앞 노드 집합을 의미한다. $DF_{up}[c]$는 노드 n에 의해 간접 지배되는 노드들에 의해 지배되지 않는 DF의 노드 집합을 의미한다. [표 5]는 생성된 DF이다.

표 5. 지배자 경계

블록(Block)	선행자(succ)	직접지배자(idom)	지배자 경계(DF)
16	17, 18	∅	∅
17	0	16	18
0	6, 11	17	18
6	13	0	13
13	18	0	18
18	∅	16	∅
11	13	0	13

[표 5]에서 보듯이 [그림 3]의 <block_13>과 <block_18>이 DF로 계산되었다. <block_18>의 경우엔 종료 노드에 해당하기 때문에 ∅-함수를 삽입하지 않는다. 따라서 ∅-함수가 삽입 될 수 있는 위치는 <block_13>이 된다. CFG에서 보듯이 <block_13>은 병합되는 노드가 위치한 곳이기도 하다.

3. ∅-함수 추가하기

DF를 통해 ∅-함수가 삽입될 위치가 결정되면 ∅-함수를 삽입한다. 구조적인 언어는 ∅-함수가 삽입될 수 있는 위치가 여러 곳에 발생할 수 있다. 왜냐하면 구조적인 언어에서는 특정 위치에 병합 노드가 발생하기 때문에 병합이 이루어지는 곳에 일반적으로 ∅-함수가 삽입된다. 하지만 자바의 바이트코드의 경우엔 대부분의 분기가 조건 분기인 if와 무조건 분기인 goto에 의해 이루어지기 때문에 간단하게 표현될 수 있다.

현재 블록에 해당 변수에 대한 정의가 존재하지 않고 다른 블록에 정의가 존재하는 경우엔 다른 블록에 존재하는 정보를 이용하여 해당 변수를 정의한다. 이를 위해 이전에 계산된 DF 값을 이용해서 반복적인 DF(DF+ : iterated dominance frontier)를 다시 계산한다. 반복적으로 구해진 DF가 바로 실제 ∅-문장이 삽입될 위치가 된다.

4. 이름 바꾸기

본 논문에서는 ∅-함수가 삽입된 문장을 PhiStmt로 표현하였다. ∅-함수가 삽입될 위치를 찾은 후 그곳에

PhiStmt를 추가 한다. 생성된 PhiStmt에 존재하는 2개의 Local_ref2_UDef 변수에 대해 재명명을 수행한다. 우선 각 블록에서 해당 변수 Locali2에 대한 정의가 각 블록에 존재하는가를 확인한다. 이때 CFG와 지배자 트리를 이용한다. <block_0>는 Local_ref2_2에 대한 정의가 존재한다. 존재하는 변수에 대한 정보를 스택의 꼭대기를 의미하는 필드인 top에 설정한다. 이 블록의 후행자에 해당하는 블록에서 다시 해당 변수에 대한 정의를 찾는다. 이때 변수에 대한 정의가 존재하면 다시 스택의 꼭대기에 새로운 변수에 대한 정보를 넣게 된다. <block_11>의 경우 스택의 꼭대기에 새로운 값인 Local_ref2_4 값이 존재하게 된다. <block_13>의 경우 PhiStmt중 하나의 피연산자인 Local_ref2_UDef가 <block_11>로부터 온 경우인데, 해당 변수를 현재 스택에 존재하는 Local_ref2_4 값으로 대체해서 이름을 바꾼다. 여기까지 진행한 후 결과는 Local_ref2_10 := Phi(Local_ref2_4, Local_ref2_UDef)이다. 모든 블록에 대해 변수에 대한 재명명을 수행한 후 결과는 Local_ref2_10 := Phi (Local_ref2_4, Local_ref2_6)이 된다. 이 과정을 통해 추가된 Ø-문장의 각 변수에 대해 명확한 이름이 설정된다.

5. 파이-문장 추가하기

```
⟨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 (Local_ref2_2 := 1)
   label_2
      if0 (Local_ref1_1 == 0) then ⟨block_11⟩ else ⟨block_6⟩
⟨block label_6⟩
   label_6
      eval (Local_ref2_6 := 2)
      goto label_13
⟨block label_11⟩
   label_11
      eval (Local_ref2_4 := 3)
      goto label_13
⟨block label_13⟩
   label_13
      Locali2_10:=Phi(Local_ref2_4, Local_ref2_6)
      return Local_ref2_10
⟨block label_18⟩
   label_18
```

그림 6. 완성된 SSA Form

생성된 PhiStmt를 삽입해야 하는 블록에 추가한다. 현재 기존의 모든 문장은 연결 리스트 형태로 존재한다. 따라서 라벨 문장 즉, label_13 다음에 생성된 PhiStmt를 추가하게 된다. [그림 6]은 PhiStmt가 추가된 후의 그래프를 나타낸다.

[그림 6]에서 보듯이 모든 블록과 블록내의 변수들은 변수들의 정의와 사용에 따라 새로운 버전 번호를 갖게 된다. 이렇게 변형된 정보는 타입 추론과 최적화를 위한 중요한 자료가 된다.

IV. 타입 배정

[그림 6]과 같이 완성된 SSA Form는 모든 블록과 블록내의 변수들이 정의와 사용에 따라서 새로운 이름을 갖게 된다. 하지만 모든 노드가 타입에 대한 정보를 갖고 있지 않기 때문에 기존에 존재하는 정보를 이용하여 타입이 결정되지 않은 노드에 타입을 설정해야 한다.

1. 동등한 노드 찾기

변수가 존재하는 노드에 타입을 설정하기 위해서는 [그림 6]과 같은 SSA Form를 가진 CFG의 기본 블록을 전위 순서로 방문하면서 동등한 노드를 찾아야 한다. 기본 블록의 문장들은 트리 형태로 구성되어 있다. 따라서 기본 블록을 찾은 후에는 해당 트리를 방문하여 각각의 문장을 구성하는 노드를 살펴보게 된다. 예를 들면 <block_0>에 있는 IfZStmt인 if0 (Local_ref1_1 == 0) then <block_11> else <block_6>에서 Local_ref1_1은 VarExpr인 경우이다. 해당 노드가 VarExpr인 경우라면 해당 표현식이 이전에 정의 되었는가를 확인한 후에 표현식의 정의를 이용하여 현재 표현식과 동등한 노드를 찾는다. [그림 6]의 경우엔 <block_17>의 Locali1_1이 <block_0>에서 타입이 결정되지 않은 Local_ref1_1과 동등하다. <block_17>은 초기화 노드이기 때문에 매개변수로 들어온 변수들에 대한 정보를 갖는다. 따라서 타입이 아직 결정되지 않았다는 의미의 ref가 아닌 이미 결정된 정수형 타입을 의미하는 i를 가진다. 동등한 노드를 구하는 알고리즘은 [알고리즘 1]과 같다.

알고리즘 1. 동등한 노드의 집합 구하기

```
Input : node1, node2 ∈ Node
Output : equiv ∈ HashMap
procedure makeEquiv(node1, node2)
begin
    s1 = equivalent(node1);
    s2 = equivalent(node2);
    if s1 != s2 do
        s1.addAll(s2);
        iter = s2.iterator();
        while iter.hasNext() do
            n = iter.next();
            equiv.put(n, s1);
        endwhile
    fi
end
```

[알고리즘 1]에서 equiv는 HashMap 클래스로 되어있다. VarExpr의 equiv는 알고리즘에 의해 {Local_ref1_1 = [Local_ref1_1, Locali1_1], Locali1_1=[Local_ref1_1, Locali1_1]}이 된다. 즉 Local_ref1_1과 Locali1_1은 동등한 노드이고 추후에 같은 타입을 배정 한다. 생성된 동등한 노드에 대한 전체 equiv는 [표 6]과 같다.

표 6. 생성된 동등한 노드에 대한 정보

```
* SSA Graph 를 통해 생성된 equivalence HashMap
1 : [2, Local_ref2_6, (Local_ref2_6 := 2), Local_ref2_6]
2 : [Local_ref2_10, Local_ref2_10, Local_ref2_10 :=
  Phi(label_6=Local_ref2_6, label_11=Local_ref2_4)]
3 : [2, Local_ref2_6, (Local_ref2_6 := 2), Local_ref2_6]
4 : [Local_ref1_1, Locali1_1]
5 : [2, Local_ref2_6, (Local_ref2_6 := 2), Local_ref2_6]
6 : [Local_ref2_4, (Local_ref2_4 := 3), 3, Local_ref2_4]
7 : [Local_ref1_1, Locali1_1]
8 : [Local_ref2_10, Local_ref2_10, Local_ref2_10 :=
  Phi(label_6=Local_ref2_6, label_11=Local_ref2_4)]
9 : [Local_ref2_10, Local_ref2_10, Local_ref2_10 :=
  Phi(label_6=Local_ref2_6, label_11=Local_ref2_4)]
10 : [Local_ref2_4, (Local_ref2_4 := 3), 3, Local_ref2_4]
11 : [Local_ref2_4, (Local_ref2_4 := 3), 3, Local_ref2_4]
12 : [2, Local_ref2_6, (Local_ref2_6 := 2), Local_ref2_6]
13 : [Local_ref2_4, (Local_ref2_4 := 3), 3, Local_ref2_4]
```

동등한 노드가 발생할 수 있는 위치는 변수 또는 상수를 가진 노드가 있는 트리인데 이것에 해당하는 것은 [그림 2]의 BNF에서 ExprStmt, IfZeroStmt, PhiJoinStmt, ReturnExprStmt 문장이 포함하는 VarExpr, PhiStmt, 그리고 StoreExpr 등에서 발생할 수 있다. 1:과 3:처럼 동등한 equiv가 발생하는 이유는 다른 블록에 동등한 노드가 존재할 수 있기 때문에 중복으로 생성된 것이다.

2. 각 노드에 번호 설정하기

SSA Form의 CFG를 통해서 각 변수나 상수에 대해 동등한 노드를 발견한 후 각 노드에 번호를 설정한다. 이는 추후에 타입을 설정할 때 동일한 형태의 노드가 여러 개 존재할 수 있는데 각 노드별로 번호를 설정하여 서로 다른 노드로 구분하기 위해서이다. TreeVisitor 클래스를 통해 기본 블록의 모든 트리를 깊이 우선으로 방문하면서 각 노드의 count 필드의 값을 설정한다. [그림 6]의 경우 설정된 전체 노드의 수는 31이 된다. 즉 전체 31개의 노드로 이루어졌다는 의미이다.

3. 타입 설정하기

적당한 위치에 타입을 설정한다. 타입 설정의 예로 [그림 6]에서 <block_11>의 문장 ExprStmt : eval (Local_ref2_4 := 3) [15]경우를 살펴본다. [15]는 노드의 count 값을 나타낸다. ExprStmt 구조는 [그림 7]과 같다.

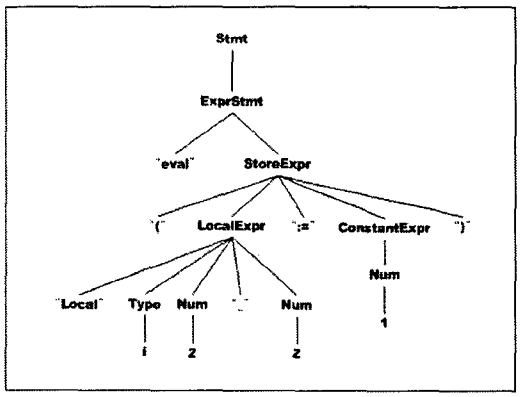

그림 7. ExprStmt의 구조

처음엔 현재 방문 중인 노드를 스택에 넣는다. 그 후 현재 노드인 ExprStmt와 동등한 노드를 찾는다. 현재는 단지 자신 만이 동등한 노드이다. 자식 노드가 있는 경우엔 자식 노드를 방문하여 자식 노드와 동등한 노드를 찾는다. 다음에 방문된 자식 노드는 [그림 7]을 보면

StoreExpr 노드인 경우이다. StoreExpr : (Local_ref2_4 := 3)[14]와 동등한 노드인 equiv를 찾으면 위의 [표 6]에서 6:과 같은 경우이다. 따라서 ConstantExpr : 3[12], StoreExpr : (Local_ref2_4 := 3)[14], LocalExpr : Local_ref2_4[25], LocalExpr : Local_ref2_4[13] 인 동등한 노드를 얻을 수 있다. 동등한 타입의 LocalExpr이 두 가지 경우가 나타난다. 이들을 서로 다른 노드이기 때문에 서로 구별하기 위해서 앞에서 계산한 count 필드 값에 해당하는 인덱스를 붙여 구분하였다.

타입을 설정하기 위해서는 현재 노드의 강 결합 요소를 구해야 하는데 강 결합 요소는 배열 리스트의 형태를 가진다. 스택이 비워있지 않다면 스택의 꼭대기에 위치한 노드를 제거하고, 제거된 노드와 동등한 노드를 구해 강 결합 요소에 추가한다. 위의 예에서는 현재 스택에 StoreExpr : (Local_ref2_4 := 3)[14]이 존재했기 때문에 스택으로부터 해당 노드를 제거하고 이 노드와 동등한 노드인 [Local_ref2_4, (Local_ref2_4 := 3), 3, Local_ref2_4]를 찾아 강 결합 요소에 추가한다. 그 후 각 컴포넌트에 해당하는 노드를 하나씩 방문하여 타입을 설정한다. 타입이 설정 가능한 노드로는 [그림 2]의 BNF에서 ConstantExpr, VarExpr, StoreExpr, PhiStmt인 경우이다. 예를 들어 현재 방문한 강 결합 요소가 ConstantExpr : 3[12] 인 경우라면, 우선 해당 표현식의 값을 value 필드로 가져온다. 상수의 경우엔 value가 문자인 경우도 있고 숫자인 경우도 있다. 따라서 경우에 따라서 서로 다르게 처리해야 한다. 어떤 종류의 상수인가를 확인하기 위해서 instanceof 연산자를 이용한다. if(value instanceof Integer)를 사용하서 참인 경우라면 value 타입이 정수인 경우이다. 결국 값으로부터 타입을 결정할 수 있다는 의미이다. 현재 강 결합 요소에 타입이 결정되면 나머지 강 결합 요소도 같은 타입이 설정되게 된다. 따라서 강 결합 요소인 [Local_ref2_4, 3, (Local_ref2_4 := 3), Local_ref2_4]에 모두 정수 타입을 설정하는 과정을 수행한다. 나머지 노드들도 이와 같은 방법을 수행하면 모든 노드에 타입을 설정할 수 있게 된다. [그림 8]은 최종적으로 타입이 설정된 SSA Form의 그래프를 나타낸다.

```
⟨block_16⟩
   label_16
⟨block_17⟩
   label_17
      INIT Local_ref0_0 Locali1_1
         *[타입 결정] : type(Local_ref0_0) = LTemp;
         *[타입 결정] : type(Locali1_1) = Z
      goto label_0
⟨block_0⟩
   label_0
   label_2
      if0 (Local_ref1_1 == 0) then ⟨block_11⟩ else ⟨block_6⟩
         *[타입 결정] : type(Locali1_1) = Z
⟨block_6⟩
   label_6
      eval (Local_ref2_6 := 2)
         *[타입 결정] : type(2) = I
         *[타입 결정] : type(Locali2_6) = I
         *[타입 결정] : type((Locali2_6 := 2)) = I
      goto label_13
⟨block_11⟩
   label_11
      eval (Local_ref2_4 := 3)
         *[타입 결정] : type(3) = I
         *[타입 결정] : type(Locali2_4) = I
         *[타입 결정] : type((Locali2_4 := 3)) = I
      goto label_13
⟨block_13⟩
   label_13
      Local_ref2_10:=Phi(label_11=Local_ref2_4,
label_6=Local_ref2_6)
         *[타입 결정] : type(Locali2_6) = I
         *[타입 결정] : type(Locali2_4) = I
         *[타입 결정] : type(Locali2_10) = I
      return Local_ref2_10
⟨block_18⟩
   label_18
```

그림 8. 타입이 설정된 SSA Form의 그래프

[그림 8]에서 [타입 결정] : type(Local_ref0_0) = LTemp; 는 지역 변수의 0번째 위치하는 클래스 타입을 나타내는 것이다. 자바 가상 머신에서는 객체 타입을 L로 표현하기 때문에 여기서도 객체 타입을 타나내기 위해 LTemp 형태로 Local_ref_0_0을 표현하였다. [타입 결정] : type(Locali1_1) = Z는 지역 변수 첫 번째 해당하는 타입은 불리언 타입이라는 것을 나타낸다. 자바 가상 기계에서는 불리언 타입을 Z로 표현한다. [타입 결정] : type(3) = I는 노드 3의 타입이 정수하는 것을 타나내는 것이다. 이렇게 해서 모든 문장과 노드에 정적으로 타입을 설정하게 된다.

V. 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 7]은 실험에 사용될 프로그램에 대한 간단한 설명이다.

표 7. 사용 예제와 간단한 설명

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

[표 8]은 SSA Form 생성 후 정적 타입 결정이 수행된 후 결과를 보여준다.

표 8. 타입 배정 후 실험 결과

프로그램	기본 블록(ea)	문장(no.)	타입 결정(no.)
Test	7	17	12
SquareRoot	11	42	75
SumOfSquareRoot	11	47	83
Fibonacci	12	48	60
QuickSort	24	82	125
LableExample	14	35	35
Exceptional	30	89	107

[표 8]에서 기본 블록(ea)는 SSA Form 과정에서 생성된 기본 블록의 개수를 나타내고, 문장(no.)은 SSA 그래프에서 사용되는 전체 문장의 개수를 나타낸다. 타입 결정(no.)은 정적 타입 배정이 수행되는 과정에서 각 노드별로 타입 결정이 몇 번 발생하는 가를 나타낸다. 예를 들면, [그림 8]에서 보듯이 Test 프로그램은 SSA Form 생성 후 7개의 기본 블록으로 나뉘고, 17개의 문장을 트리 형태로 포함하고 있으며, 12번 타입 결정이 발생하는 것을 알 수 있다.

VI. 결 론

최근 여러 분야에서 자바 바이트코드가 중간 표현으로 사용된다. 하지만 실행 속도가 느리고 분석하기 어렵다는 단점이 존재한다. 따라서 빠른 실행 속도와 프로그램의 이해를 높이기 위해서는 최적화와 프로그램 분석이 요구된다.

본 논문에서는 바이트코드 수준에서 프로그램을 분석하고 최적화를 할 수 있는 준비를 수행하였다. 바이트코드 수준에서 분석과 최적화를 수행하기 위해서는 우선 CFG를 생성하였다. 하지만 바이트코드의 특성 때문에 기존의 제어 흐름 분석 기술을 바이트코드에 적합하게 확장하였다. 또한 정적인 분석을 위해 CFG를 SSA Form으로 변환하였다. 정적 단일 형태로 변환을 수행하기 위해 CFG, 지배자 관계, 지배자 트리, 직접 지배자, Ø-함수, 재명명, DF 등 많은 정보에 대한 계산을 수행하였다.

SSA Form로 변환 후에 각 노드에 적합한 타입을 설정하기 위해 동등한 노드를 구하고 동등한 노드를 발견하고 강 결합 요소로 설정한 후 각 노드에 동일한 타입을 설정하였다. 이렇게 타입이 설정된 SSA Form는 추후 수행될 최적화에 적합한 입력 형태가 된다.

향후 연구로는 클래스 간의 정보를 이용하여 수행되는 타입 추론과 최적화에 관한 연구를 계속 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] J. Gosling, Bill Joy, and Guy 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, 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