韓國 컴퓨터情報學會 論文誌
第11卷 第4號, 2006. 9.

2006-11-4-1-11

바이트코드를 위한 정적 단일 배정문 기반의 정적 타입 추론

김 지 민*, 김 기 태**, 김 제 민***, 유 원 희****

Static Type Inference Based on Static Single Assignment for Bytecode

Kim Ji Min*, Kim Ki Tea**, Kim Je Min***, Yoo Weon Hee****

요 약

바이트코드는 많은 장점을 가지고 있으나 수행 속도가 느리고 프로그램의 분석과 최적화에 알맞은 표현은 아니다. 분석과 최적화를 위하여 바이트코드를 정적 단일 배정문(SSA Form)으로 변환이 수행되어야 한다. 그러나 바이트코드에서 SSA Form으로 변환 시 어떤 변수는 타입 정보를 상실한다. 이를 해결하기 위하여 본 논문에서는 바이트코드에 대한 확장된 제어 흐름 그래프를 생성한다. 또한 정적으로 분석하기 위해 제어 흐름 그래프를 SSA Form으로 변환한다. SSA Form으로 변환을 위하여 지배자, 직접 지배자, 지배자 경계, ∅-함수, 재명명 등 많은 정보에 대한 계산을 수행한다. 생성된 SSA Form에 알맞은 타입을 선언하기 위해서 다음과 같은 동작들을 수행한다. 먼저 클래스의 호출 그래프와 상속 그래프를 생성한다. 그리고 각 노드에 대한 정보를 수집한다. 수집된 정보를 기반으로 동등한 타입의 노드를 찾고 강 결합 요소로 설정한 후 각 노드에 타입을 효율적으로 설정하는 방법을 제안하였다.

Abstract

Although bytecode has many good features, it has slow execution speed and it is not an ideal representation for program analysis or optimization. For analysises and optimizations, bytecode must be translated to a Static Single Assignment Form(SSA Form). But when bytecode is translated a SSA Form it has lost type informations of some variables. For resolving these problem in this paper, we create extended control flow graph on bytecode. Also we convert the control flow graph to SSA Form for static analysis. Calculation about many informations such as dominator, immediate dominator, dominance frontier, Φ-Function, renaming are required to convert to SSA Form. To obtain appropriate type for generated SSA Form, we proceed the followings. First, we construct call graph and derivation graph of classes. And the we collect information associated with each node. After finding equivalence nodes and constructing Strongly Connected Component based on the collected informations, we assign type to each node.

▸ Keyword : 정적타입추론(Static Type Inference), 정적단일배정문(Static Single Assignment), CFG(Control Flow Graph), 바이트코드(Bytecode)

• 제1저자 : 김지민
• 접수일 : 2006.08.11, 심사일 : 2006.08.17, 심사완료일 : 2006.09.23
* 인하대학교 컴퓨터정보공학과 석사 ** 인하대학교 컴퓨터정보공학과 박사
*** 인하대학교 컴퓨터정보공학과 석사 ****인하대학교 컴퓨터정보공학과 교수
※본 논문은 2004년 인하대학교의 지원에 의하여 연구되었음.(INHA-31513)

Ⅰ. 서 론

자바는 수행속도가 느리다는 커다란 단점을 가지고 있다. 수행속도를 증가시키기 위해서는 바이트코드에 대한 최적화가 요구된다[1]. 바이트코드는 유용한 특징을 많이 갖지만 스택을 기반 코드로서 수행 속도가 느리고 프로그램 분석이나 최적화에 적절한 표현은 아니다. 그러므로 바이트코드를 프로그램 분석과 최적화에 알맞은 형태로 변화시킬 필요가 있다[2]. 이를 위하여 확장된 데이터흐름 분석을 수행한 후 분석과 최적화에 알맞은 중간표현으로 변환 시켜야 된다[3]. 본 논문에서는 중간표현으로 정적 단일 배정문 형태(SSA Form : Static Single Assignment Form)를 이용한다[4].

SSA Form은 일반적으로 데이터 흐름 분석이나 코드 최적화를 위해 컴파일러의 중간 표현으로 많이 사용된다. SSA Form은 각 변수들이 프로그램 안에서 한번씩만 정의되는 것이다. 바이트코드에서 SSA Form으로 변환 시 각 스택변수는 지역변수로 변환이 이루어지게 된다. 이 때 하나의 스택은 여러 개의 지역변수로 변환하게 되면서 어떤 지역변수는 명확한 타입 정보를 잃어버리는 단점이 발생한다. 이를 해결하기 위해 타입 추론을 수행해야 한다. SSA Form으로 변환 될 때 각 변수에 대한 타입이 올바르게 배정되었는가는 중요한 문제이다. 왜냐하면 SSA Form에서 바이트코드로 역컴파일시 각 변수는 변수의 모든 정의와 사용에 대해 정적으로 선언된 올바른 타입을 요구하기 때문이다. 또한 변수의 타입 정보는 분석력을 향상시키기 때문이다. SSA Form으로 변경한 후 각각의 표현식은 트리 형태를 지니게 되고 각각의 노드는 변수를 표현된다.

타입 전파를 위하여 많은 연구들이 진행되고 있다. 대표적인 방법들로는 제어 흐름에 의존한 타입 계산 방법과 JumpFuction들을 이용하는 방법 그리고 제약 모델 등의 사용을 들 수 있다. 본 논문에서는 표현식과 노드에 대한 타입 전파를 위하여 기본 정보와 동등한 노드를 찾기 위한 제약 모델을 사용한다.

본 논문의 구성은 다음과 같다. 2장은 기존 타입 전파 방법의 종류와 바이트코드에 알맞게 확장된 제어흐름그래프(CFG : Control Flow Graph)생성에 대하여 설명한다. 3장은 SSA Form으로의 변환과정을 설명한다. 4장에서는 타입 전파 순서 및 타입 전파를 위한 타입 제약 모델과 알고리즘에 대하여 기술한다. 5장에서는 본 논문에서 제시한 타입 전파 기법을 이용한 실험을 수행한다. 6장에서는 결론과 향후 계획을 논한다.

Ⅱ. 기존 타입 전파 방법과 CFG

2.1 기존 타입 전파 방법

타입 분석은 재귀적 데이터 구조 표현에서는 결정 불가능한 문제라고 증명되었다. 재귀적 데이터 구조 표현뿐만 아니라 많은 부분에서 정적 타입이 추론되지 못하는 경우가 있다. 그럼에도 불구하고 타입 추론에 대한 많은 연구가 이루어지고 있다. Gil and Itai는 타입 추론을 위하여 새로운 행렬을 개발하였다[5]. 만약 구체적인 타입(Concrete Type)이 대다수의 변수에 배정되어진다면 전체프로그램에 대한 분석이 가능하다고 서술하고 있다. Pande와 Ryder는 변수의 구체적인 타입을 결정할 수 있는 알고리즘을 제안하였다[6]. 그들은 프로시져간 흐름 간선으로 구성된 프로시져간 표현을 제안하였고 이것을 이용하여 프로시져간 타입 정보에 대한 전파를 가능하게 한다고 주장하였다. Lenart, Sadler 그리고 Gupta는 변수의 구체적 타입을 결정하기 위하여 여러 가지의 Jumpfuction을 제안하였다[7]. 이러한 Jumpfuction은 프로시져 진입점에서의 상수 값 또는 구체적인 타입 집합을 전파하는데 사용된다. 위에 제시된 방법들은 대부분 많은 양의 새로운 정보를 추가 시키거나 새로운 분기들을 발생시켜 프로그램 분석 비용과 시간 이용을 높이고 있다. 이러한 단점을 보안하기 위하여 많은 타입 제약 모델들이 제안되어져 왔다. 그 중에서 본 논문에서는 Palsberg와 Schwartzbach가 제안한 타입 제약 모델을 기반으로 하여 본 논문에서 사용하는 표현식에 알맞게 변형한다[8].

2.2 CFG

바이트코드 자체를 분석하는 것은 분석 비용이나 시간 비용에서 많은 단점을 내포하고 있다. 그러므로 분석에 용이한 형태로 변환하기 위해서는 먼저 바이트코드에 알맞게 확장된 CFG를 구성해야 된다[9]. 아래 〈그림 1〉은 사용될 예제 프로그램과 예제 프로그램에 대한 바이트코드를 나타내고 있다.

```
class A extends Object{
}

class B extends A{
public String
toString(){
String s = "class B";
return s;  }}

class C extends A{
public String
toString(){
String s = "class C";
return s;  }}

public class Temp2 {
public String f(boolean
bol){
A a = new A();
A b = new B();
A c = new C();
   if(bol)
  a = b;
   else
  a = c;
String s = a.toString()
;
  return s;  }}
```

```
.....
int f(boolean);
new LA;
dup
invokespecial <Method
LA;.<init> ()V>
astore Local$2
new LB;
dup
invokespecial <Method
LB;.<init> ()V>
astore Local$3
new LC;
dup
invokespecial <Method
LC;.<init> ()V>
astore Local$4
iload bol$1
ifeq label_34
aload b$3
astore a$2
goto label_37
aload c$4
astore a$2
aload a$2
invokevirtual <Method
Ljava/lang/Object;.toString
()Ljava/lang/String;>
astore Local$5
aload s$5
areturn
```

그림 1.예제 프로그램과 바이트코드
Fig 1. Example Program and Bytecode

바이트코드에 알맞게 확장된 CFG를 생성하기 위해서는 바이트코드 자체를 기본 블록단위로 나누어야 한다. 프로그램 내에서 제어 흐름에 영향을 주는 리더(Leader)를 찾아 기본 블록을 생성한다. 그 후 프로그램에 대한 분석을 용이하게 하기 위하여 각 문장에 대한 라벨을 추가시킨다. 기본 블록에 라벨을 추가시키기 위하여 클래스 파일 내에 존재하는 LineNumbers배열을 참조하게 된다[10].

생성된 제어 흐름 그래프는 유향 그래프로 작성되고, 그래프의 각 노드는 기본 블록으로 이루어진다. 제어 흐름 그래프를 효율적으로 생성하기 위해서는 시작 블록(entry block), 종료 블록(exit block), 그리고 초기화 블록(initial block)를 기본 블록으로 추가한다. 시작 블록은 그래프의 진입 지점을 알리기 위해 사용되고, 종료 블록은 그래프의 진출 지점을 알리기 위해 사용된다. 초기화 블록은 그래프에서 사용되는 매개변수와 같은 정보들의 초기화를 위해 사용된다.

III. SSA Form으로 변환

3.1 트리 형태 구문 설정

분석과 최적화를 위하여 CFG에 의하여 생성된 블록에 존재하는 문장들을 트리 형태의 3-주소 형식 문장으로 변환 시켜야 된다. 이때 사용되는 트리를 표현트리(Expression Tree)라 부른다. 표현트리를 만들기 위한 BNF는 아래 〈그림 2〉와 같다.

```
Expr → ConstantExpr | DefExpr | StoreExpr | ShiftExpr
ConstantExpr → ' Id ' | Num + F | Num
                + L  |  Num
StoreExpr → ( MemExpr := Expr )
DefExpr → MemExpr
NewExpr → New Type name
MemExpr → MemRefExpr  |  VarExpr
VarExpr → LocalExpr  |  StackExpr
LocalExpr → Stack  |  Local  Type  Num
Stmt → ExprStmt | InitStmt | JumpStmt | LabelStmt |
StackManipStmt
LabelStmt → Label
ExprStmt → eval  Expr
InitStmt → INIT  LocalExpr( )
JumpStmt → GotoStmt | IfStmt | ReturnExprStmt
IfStmt → IfZeroStmt
GotoStmt → goto  Block
IfZeroStmt → if 0 (  Expr  == | != | > | >=  |
   < |<=  null  |  0 )  then Block else Block
ReturnExprStmt → return  Expr
StackManipStmt → ( LocalExpr, LocalExpr ) := dup(
LocalExpr )
Block → <block  Label  >
Label → label_Num
```

그림 2.변환을 위한 BNF
Fig 2. BNF for Translation

BNF는 크게 두 가지로 분류되어진다. 하나는 Expr에서 파생되는 표현식으로 이 표현식은 값과 타입을 갖고 있다. 다른 하나는 Stmt에서 파생되는 표현식으로 이 표현식은 3-주소 형태를 표현하기 위한 것이다. 〈그림 1〉의 코드를 BNF와 필요한 정보를 추가 시켜 CFG를 생성하면 〈그림 3〉과 같다. 〈그림 3〉의 CFG는 7개의 기본 블록이 생성되고 블록 내부의 문장들은 트리 형태로 추가되어 진다. CFG에서 각 노드는 하나의 블록을 의미하고 블록 내부는 여러 개의 부트리로 이루어져 있

다. 각 부트리는 하나의 문장을 표현하고 있고, 그 부트리의 노드는 각각의 변수들을 나타내고 있다. 3-주소 형태 문장들의 타입 정보를 나타내는 부분에서 대부분 타입이 아직 결정되어지지 않았다는 의미의 'undef'로 표기되어진다. 이유는 바이트코드에서 사용된 하나의 스택 변수가 3-주소 코드형태로 변형 시 여러 개의 서로 다른 지역변수로 변환되면서 타입 정보들을 상실하게 되기 때문이다. 또한 읽어 들인 정보들이 주소 참조의 내용을 가지고 있으므로 타입 정보가 명확치 않기 때문이다.

```
<block 47 source>
label_47
<block 48  init>
label_48
INIT Local_ref0_0  Locali1_1
goto label_0
<block 0>
label_0
eval (S_ref0_2 := new LA;)
(S_ref0_4, S_ref1_6) := dup(S_ref0_undef)
eval S_ref1_undef.<init>()
eval (Local_ref2_8 := S_ref0_undef)
label_8
eval (S_ref0_9 := new LB;)
(S_ref0_11, S_ref1_13) := dup(S_ref0_undef)
eval S_ref1_undef.<init>()
eval (Local_ref3_15 := S_ref0_undef)
label_16
eval (S_ref0_16 := new LC;)
(S_ref0_18, S_ref1_20) := dup(S_ref0_undef)
eval S_ref1_undef.<init>()
eval (Local_ref4_22 := S_ref0_undef)
label_25
if0 (Locali1_undef == 0) then <block 34 >
else <block 29>
<block 29>
label_29
eval (Local_ref2_30 := Local_ref3_undef)
goto label_37
<block 34>
label_34
eval (Local_ref2_25 := Local_ref4_undef)
goto label_37
<block 37>
label_37
eval (Local_ref5_27 :=
local_ref2_undef.toString())
label_43
return Local_ref5_undef
<block 49 sink>
label_49
```

그림 3. 생성된 CFG
Fig 3. Created CFG

3.2 지배자 경계 구하기

SSA Form으로 변형하기 위해서는 CFG를 방문하면서 지배자 경계를 찾아야한다. 지배자 경계(DF : Dominator Frontier)는 Ø-함수가 삽입될 위치다. Ø-함수는 if문이나 while문 같이 CFG에서 제어가 병합되는 경우에 사용되어 진다. 지배자 경계를 찾기 위해서는 지배자와 직접 지배자에 대한 정보가 필요하다. 지배자를 구하기 위해 Aho의 알고리즘을 변형하여 이용한다[11]. 아래 〈그림 4〉는 알고리즘에 의하여 생성된 지배자 트리이다.

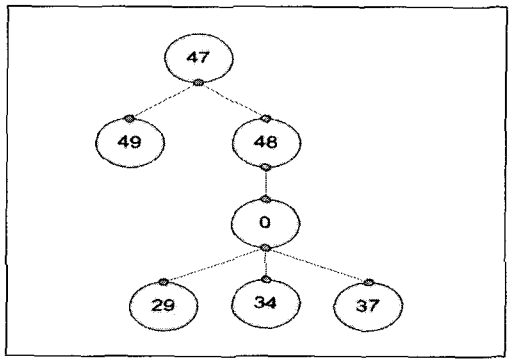

그림 4.지배자 트리
Fig 4. Dominator Tree

〈그림 4〉에서 각 원은 기본 블록들을 나타내고 있다. 또한 분석의 용이하게 하기 위하여 시작 블록, 종료 블록, 그리고 초기화 블록을 추가 시켰는데 이는 각각 47번, 49번, 그리고 48번 블록이다. 지배자를 구하였다면 지배자 정보를 이용하여 직접지배자를 구할 수 있다. 직접지배자를 구하기 위해서는 그래프의 노드들에 해당하는 블록들을 찾아서 현재 블록의 전위 순서를 구한다. 직접 지배자를 구하는 식은 아래 식(3.1)과 같다.

idom := dom(block)-dom(dom(block))-block ··· (3.1)

〈표 1〉는 위 방법들을 통하여 얻게 된 직접 지배자를 나타내고 있다.

표 1 직접 지배자
Table 1. Immediate Doninator

블록	47	49	48	0	29	34.	37.
idom	Null	47	47	48	0	0	0

노드 n의 DF를 DF[n]으로 표현할 때 DF[n]을 구하기 위해서는 $DF_{local}[n]$과 $DF_{up}[n]$을 알아야 한다. $DF_{local}[n]$은 노드n에 의해서 직접 지배되지 않는 n의 앞노드 집합을 의미한다. $DF_{up}[c]$는 노드 n에 의해 간접 지배되는 노드들에 의해 지배되지 않는 DF의 노드 집합을 의미한다. 아래 (식 3.2)는 DF를 구하는 식이고, 〈표 2〉는 지배자 경계를 나타내고 있다.

$$DF[n] = DF_{local}[n] \cup \bigcup_{c \in children[n]} DF_{up}[c] \quad \cdots\cdots\cdots\cdots (3.2)$$

표 2. 지배자 경계(DF)
Table 2.. Dominator Frontier

블록	47	49	48	0	29	34	37
DF	Null	Null	49	49	37	37	49

3.3 파이 문장 추가하기

DF를 통하여 Ø-함수가 삽입될 위치가 정해졌다면 Ø-함수를 삽입한다. 만약 현재 블록에 해당 변수에 대한 정의가 존재 하지 않고 다른 블록에 정의가 존재한다면 다른 블록의 정보를 이용하여 변수를 정의한다. 이를 위해 반복적으로 DF 값을 계산한다. 반복적으로 구해진 DF가 실제 Ø-문장이 들어갈 위치가 된다. Ø-문장을 위하여 본 논문에서는 PhiStmt를 추가한다. 생성된 PhiStmt에 존재하는 두 개의 Local_ref2_Undef변수에 대하여 재명명을 수행한다. 예제에서 보면 PhiStms가 추가될 블록은 37번 블록이다. 37번 블록에서 Local_ref2_Undef변수에 대한 두 종류의 서로 다른 타입의 정의가 내려온다. 37번 블록에서 PhiStmt에 의하여 정의될 Local_ref2_Undef를 위하여 스택 사용 정보를 이용하여 37번 블록으로 내려올 서로 다른 정의를 가진 Local_ref2_undef변수에 대한 재명명을 수행하면 Local_ref2_30와 Local_ref2_25로 재명명 이 이루어진다. 새로이 삽입될 Loca_ref2_Undef변수에 대한 재명명은 위의 방법과 동일하게 수행하게 된다.

결과적으로 37번 블록에는 Local_ref2_39 := Phi(Local_ref2_30, Local_ref2_25)문장이 들어가게 된다. 본 논문에서는 분석을 용이하게 하기 위하여 각 변수의 정의가 어디로부터 왔나를 알려주기 위하여 각 정의가 생성된 라벨 정보를 앞에 표기해준다. 아래 〈그림 5〉는 완성된 SSA Form을 보여주고 있다. 〈그림 5〉에서 보듯이 모든 블록과 블록 내 사용된 변수들은 정의와 사용에 따라 새로운 번호를 갖게 된다. 이러한 정보는 정적 타입 추론 및 최적화에 중요한 정보로 사용되어진다.

```
<block label_48 >
label_48
INIT Local_ref0_0 Locali1_1
goto label_0
<block label_0>
label_0
eval (S_ref0_2 := new LA;)
(S_ref0_4, S_ref1_6) := dup(S_ref0_2)
eval S_ref1_6.<init>()
eval (Local_ref2_8 := S_ref2_4)
label_8
eval (S_ref0_9 := new LB;)
(S_ref0_11, S_ref1_13) := dup(S_ref0_9)
eval S_ref1_13.<init>()
eval (Local_ref3_15 := S_ref0_11)
label_16
eval (S_ref0_16 := new LC;)
(S_ref0_18, S_ref1_20) := dup(S_ref0_16)
eval S_ref1_20.<init>()
eval (Local_ref4_22 := S_ref0_18)
label_25
if0 (Locali1_1 == 0) then <block label_34>
else <block label_29>
<block label_29>
label_29
eval (Local_ref2_30 := Local_ref3_15)
goto label_37
<block label_34>
label_34
eval (Local_ref2_25 := Local_ref4_22)
goto label_37
<block label_37>
label_37
Local_ref2_39 :=
          Phi(label_29=>Local_ref2_30,
      label_34=>Local_ref2_25)
eval (Local_ref5_27 := Local_ref2_39.toString())
label_43
return Local_ref5_27
<block label_49>
label_49
```

그림 5. 완성된 SSA Form
Fig 5. Completed SSA Form

IV. 타입 추론

4.1 클래스 계층 그래프

본 논문에서는 클래스 계층을 나타내기 위해서 호출 그래프와 클래스의 타입 정보를 갖고 있는 상속 계층 그

래프를 사용한다. 호출 그래프는 각 클래스에서 클래스 내 메소드에 대한 호출 관계를 나타낸다. 호출 그래프에서 각 노드는 클래스를 의미한다. 각 노드는 클래스 내 메소드를 포함하고 있다. 노드 간 연결되는 간선은 클래스 간 호출 관계를 나타내고 있다. 클래스에 대한 상속 계층 그래프에 각 노드는 각각의 클래스를 의미한다. 상속 계층 그래프를 구성하기 위하서는 먼저 각각의 사용될 클래스 이름을 모은다. 이 정보는 바이트코드에서 SSA Form으로 변환 시 향후 필요한 정보를 위한 데이터 구조를 생성하고 그곳에서 정보를 읽어 온다. 그리고 읽어온 클래스 이름과 클래스 타입 정보를 클래스 정보를 저장하기 위한 워크리스트에 추가 시킨다. 아래 〈그림 6〉은 예제 프로그램에 대한 호출 그래프와 클래스 상속 계층을 나태는 그래프이다.

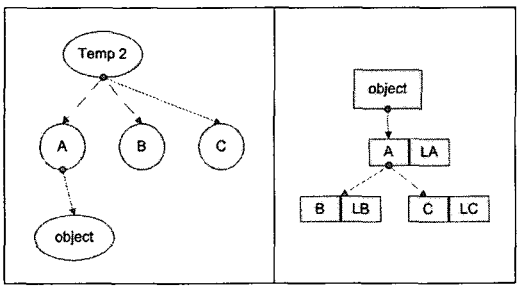

그림 6.(a)호출 그래프 (b)상속 그래프
Fig 6. (a) Call Graph (b) Hierarchy Graph

〈그림 6〉의 (a)호출 그래프에서 각각의 원은 클래스를 의미하고 각 클래스를 잇는 간선은 유향 간선으로 호출 관계를 나타내고 있다. 예제에서 보면 Temp2클래스는 A, B, C클래스를 호출한다. 〈그림 6〉의 (b)상속 그래프에서 사각형이 클래스를 나타낸다. 그리고 그 옆에 연결되어 있는 사각형은 해당 클래스 타입을 의미한다. 각 클래스를 잇는 간선들은 유향 간선으로 각 클래스간의 상속 관계를 표현하고 있다. 이렇게 타입정보를 나타내는 이유는 클래스에 대한 최대한의 타입 정보를 얻기 위함이다.

만약 새로운 클래스가 추가되어진다면 새로운 호출 그래프나 상속그래프가 그려지는 것이 아니라 클래스들의 워크리스트에는 클래스 이름 및 클래스 타입 정보가 추가된다. 예를 들어 추가될 클래스가 메서드 형태라면 그것의 이름, 매개변수와 반환 타입이 워크리스트에 저장되어 진다. 이러한 상속 계층 그래프와 워크리스트를 통하여 클래스 간 계층 정보를 얻을 수 있고 타입 전파를 위한 제약 모델에 사용이 용이해진다.

4.2 노드 추적

변수에 대한 타입을 배정하기 위해서 SSA Form으로 구성되어진 CFG의 기본 블록을 전위 순서로 방문한다. 기본 블록에 포함되어 있는 문장들은 리스트 형태로 연결되어 있으며 각 문장은 트리 형태로 이루어져 있다. 각 노드는 각각의 변수 및 표현을 나타내고 있다. 〈그림 7〉은 블록 0번의 ExprStmt문장인 eval Local_ref2_30 := Local_ref3_15의 구조를 나타내고 있다. 이 구조에 나타나듯이 아직까지는 각 변수에 대한 명확한 타입이 선언되어 있지 않음을 알 수 있다.

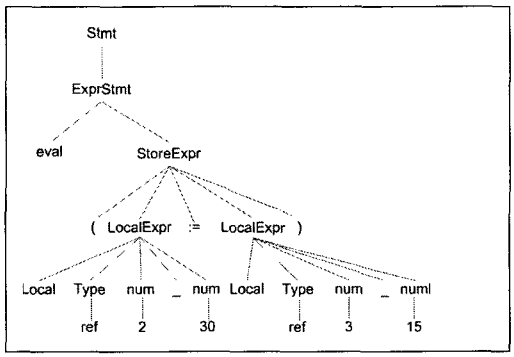

그림7. 트리형태의 문장 구조
Fig 7. Express construct of Tree Form

노드를 탐색하기 위해서 visitor를 사용한다. visitor는 가장 먼저 현재 블록에 대한 타입을 검사한다. 블록의 타입에 따라 얻어지는 정보가 다르기 때문에 현재의 블록의 타입정보가 필요하다. 앞에서 언급했듯이 블록은 크게 네 가지 형태의 블록으로 이루어져 있기 때문이다. 또한 각각에 블록에 대한 수행되는 방법이 다르기 때문이다. 블록에 대한 타입이 결정되었다면 블록 내 문장을 나타내는 트리가 있는지를 확인한다. 블록 내 트리가 없다면 그것은 시작 블록이거나 종료 블록일 것이고 트리가 존재한다면 초기화 블록이거나 소스 블록일 것이다. 만약 트리가 존재한다면 각각의 노드를 방문한다. 노드를 방문하면서 각각의 노드에 대한 정보를 수집한다. 만약 노드가 Stmt들이라면 각 Stmt를 구성하고 있는 자식 노드를 찾아가 Stmt를 구성하는 변수들에 정보를 저장한다. 위 동작은 더 이상 자식 노드가 없을 때까지 반복하여 모든 노드에 대한 정보를 저장한다.

4.3 타입이 동등한 노드 찾기

각 노드에 대한 정보를 모았다면 타입 전파를 위하여 타입이 동등한 노드를 찾아야한다. 타입 전파를 위하여 타입 제약 모델과 동등한 노드를 찾기 위한 알고리즘을 사용하였다. 논문에서 사용하는 타입 제약 모델은 기존에 Palsberg과 Schwartzbach가 제안한 모델을 변형시킨 것이다. 아래 〈표 3〉은 타입 제약 모델은 나타내고 있다.

표 3. 타입 제약 모델

Table 3. Type Constraints Model

Expression	Constraint
Id := E	⟦Id⟧⊇⟦E⟧ ∧ ⟦Id := E⟧ := ⟦E⟧
E_1 := E_2	⟦E_1 := E_2⟧ := ⟦E_2⟧
New C	⟦New C⟧ := { C }
if E_1 then E_2 else E_3	⟦E_1⟧ := Z ∧ ⟦if E_1 then E_2 else E_3⟧ ⊇ ⟦E_2⟧ ∪ ⟦E_3⟧
Id_3 := phi(Id_1, Id_2)	⟦Id_3⟧ := {the enclosing common ancestor of Id_1 and Id_2 } ∨ ⟦Id_3⟧ := {Id_1 ∨ Id_2}
(Id_3, Id_2) := dup(Id_1)	⟦Id_2⟧ := ⟦Id_1⟧ ∧ ⟦Id_3⟧ := ⟦Id_1⟧
....	....
Id	⟦Id⟧ := ⟦Id⟧

타입 제약 모델에서 Id는 변수, E는 표현식, C는 클래스를 의미한다. ⟦E⟧는 표현식 E의 타입을 의미한다. {C}는 클래스 C의 타입을 의미한다. 또한 본 논문에서는 실 매개변수와 형식 매개변수의 타입은 동등하다고 가정하고 있다. 이러한 타입 제약 모델은 타입을 계산하는데 사용되어 진다. 타입 제약을 통하여 어떠한 노드가 서로 동등한 타입을 가질 수 있는지 알았다면 타입이 동등한 노드를 찾아야 한다. 동등한 노드를 구하는 알고리즘은 〈그림 8〉과 같다.

```
Input : node1, node2 ∈ Node
Output : equiv ∈ HashMap
procedure equivalent(Node node)
 begin
  Set s = (Set) equiv.get(node);
  if (s == null)
   s = new HashSet(1);
  s.add(node);
  equiv.put(node, s);
 endif
 return s;
end
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
 endif
end
```

그림 8. 동등한 노드 계산 알고리즘

Fig 8. Equivalent node Compute Algorithm

위 알고리즘에서 equivalent 프로시져는 동등한 노드를 찾기 위하여 먼저 자기 자신을 동등한 노드에 추가 시키는 작업을 한다. makeequiv 프로시져는 자신을 제외한 동등한 노드를 찾는다. 알고리즘 1을 수행하기 위하여 SSA Form을 가진 CFG를 방문하면서 각각의 노드에 대한 정보를 수집한다. 그러나 이미 각 노드에 대한 정보를 가지고 있으므로 저장된 정보를 이용하면 동등한 노드를 찾을 수 있다. 동등한 노드가 발생할 수 있는 위치는 변수 또는 상수를 가진 노드가 있는 트리인데 이것에 해당하는 것은 〈그림 2〉의 BNF에서 보면 ExprStmt, IfZeroStmt, PhiJoinStmt, ReturnExprStmt 문장이 포함하는 VarExpr, PhiStmt 그리고 StoreExpr 등에서 발생할 수 있다. 기본적으로 StoreExpr과 PhiStmt는 변수들의 타입 정보를 전파한다.

4.4 타입 전파

타입을 설정하기 위하여 동등한 노드를 발견 후 각 노드에 번호를 설정한다. 이는 타입 설정 시 동일한 형태의 노드가 하나 이상 존재 할 수 있기 때문에 각 노드별로 번호를 설정하여 서로 다른 노드 간에 구분을 위해서이다. 예제 코드의 경우 전체 노드의 수는 82개가 된다.

각 노드가 구분되었다면 타입을 전파한다. 타입 전파

를 위해서는 구분된 각 노드를 방문한다. 일반적으로 현재 노드의 강 결합 요소를 구해야 하는데 강 결합 요소는 배열 리스트 형태를 가진다. 강 결합 요소를 찾기 위한 스택이 비워있지 않다면 스택의 꼭대기에 위한 노드를 제거하고, 제거된 노드와 동등한 노드를 구하여 강 결합 요소에 추가한다. 그러나 만약 현재 노드에 자식 노드가 존재한다면 자식 노드에 대한 강 결합 요소를 구하고 그 후 각 컴포넌트에 해당하는 노드를 하나씩 방문하여 타입을 설정한다. 타입 설정 가능한 노드로는 〈그림 2〉의 BNF에서 ConstantExpr, VarExpr, StoreExpr, PhiStmt인 경우이다. 그러나 위의 방법은 일반적인 경우로 타입이 한번에 결정될 수 있는 경우이다. 그러나 〈그림 5〉의 Local_2_39 := Phi(label_29=〉 Local_ref2_30, label_34=〉Local_ref2_25)문장의 경우와 같이 객체 간 타입 전파가 제어 흐름에 따라 이루어진다면 타입 설정은 다음과 같이 이루어진다. 먼저 문장 자체를 강 결합 요소를 찾기 위한 스택에 넣는다. 그 후에 현재 노드와 동등한 노드가 존재하는지를 확인한다. 현재의 노드는 항상 현재 노드와 같기 때문에 동등한 노드이다. 그 후 자식 노드인 Local_ref2_30과 Local_ref2_25의 노드의 강 경합 요소를 찾아 타입을 찾아 설정한다. 성정된 Local_ref2_30의 타입은 B 타입이 되고 Local_ref2_25의 타입은 C타입이 된다. 이러한 객체간의 타입 전파인 경우 일반적으론 Local2_39에 대한 타입을 위하여 최상위 타입인 Object가 할당되어야 한다. 하지만 Object타입으로 설정은 여러 가지 문제점을 발생한다. 이를 피하기 위하여 호출 그래프와 계층 그래프를 이용하여 자식 노드들의 가장 인접한 공통 조상의 타입을 배정한다. 계층그래프에서 B 클래스와 C 클래스에 가장 가까운 공통 조상은 A이므로 Local2_39에 대한 타입은 A타입이 된다. 그러나 만약 공통 조상이 없는 경우라면 모든 객체의 최상위인 Object타입이 설정된다. 아래 〈그림 9〉은 타입 전파를 통하여 타입이 모두 설정된 SSA Form를 보여주고 있다.

```
<block label_47>
 label_47
<block label_48>
label_48
INIT Local_ref0_0 Localil_1
*[타입 결정] : (Local_ref0_0) = LTemp2;
*[타입 결정] : (Localil_1) = Z
goto label_0
<block label_0 >
 label_0
(S_ref0_2 := new LA;)
*[타입 결정] : (new LA;) = LA;
*[타입 결정] : (S_ref0_2) = LA;
*[타입 결정] : ((S_ref0_2 := new LA;)) = LA;
(S_ref0_4, S_ref1_6) := dup(S_ref0_2)
*[타입 결정] : (S_ref0_2) = LA;
*[타입 결정] : (S_ref0_4) = LA;
*[타입 결정] : (S_ref1_6) = LA;
 eval S_ref1_6.<init>()
*[타입 결정] : (S_ref1_6) = LA;
*[타입 결정] : (S_ref1_6.<init>()) = V
eval S_ref0_4
*[타입 결정] : (S_ref0_4) = LA;
label_8
eval (S_ref0_9 := new LB;)

                .........
                .........
 <block label_37>
label_37
Local_ref2_39 :=
            phi(label_29=>Local_ref2_30,
                label_34=>Local_ref2_25)
*[타입 결정] : (Local_ref2_30) = LB;
*[타입 결정] : (Local_ref2_25) = LC;
*[타입 결정] : (Local_ref2_39) = LA;
eval (Local_ref5_27 :=
                  local_ref2_39.toString())
*[타입 결정] : (Local_ref2_39) = LA;
*[타입 결정] : (Local_ref2_39.toString()) =
                Ljava/lang/String;
*[타입 결정] : (Local_ref5_27) =
                 Ljava/lang/String;
*[타입 결정] : ((Local_ref5_27 :=
Local_ref2_39.toString())) =
Ljava/lang/String;
Local_ref2_39.toString())) =
Ljava/lang/String;
label_43
return Local_ref5_27
*[타입 결정] : typeof(Local_ref5_27) =
               Ljava/lang/String;
                .........
                .........
<block label_49 >
label_49
```

그림 9. 타입이 추론된 SSA Form

Fig 9. Type is Inferred SSA Form

V. 실험 결과

실험은 펜티엄 4 2.4GHz, Windows XP professional 버전, 메모리 512MB, 자바 컴파일러는 j2sdk 1.4.2_09 에서 실험 했으며 Eclipse 3.1.1 에디터에서 테스트를 수행 하였다. 〈그림 10〉은 실행화면을 보여주고 있다.

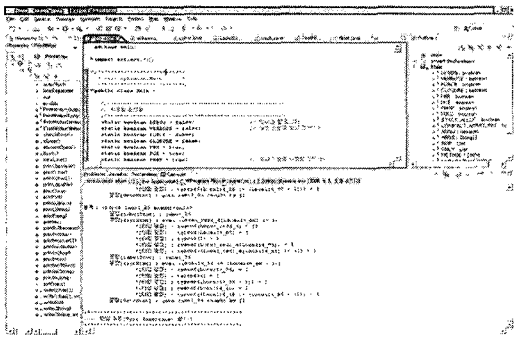

그림 10. 실험화면
Fig 10. Program Execution.

예제 프로그램으로는 Fibonacci, BobbleSort, Sieve, Hash, Ackerman 함수에 대하여 실험을 수행 하였다. 〈표 4〉에서는 예제 프로그램들에 대한 바이트코드에서 3-주소 형태의 문장으로 표현한 CFG로 변화 시 프로그램의 크기의 변화를 실험하였다. 〈표 4〉에서 소스는 실제 소스 프로그램의 라인 수, 바이트코드는 바이트코드의 라인 수 그리고 CFG는 3-주소 형태로 변환했을 때의 라인 수를 의미한다.

표 4. 실험 결과 1
Table 4. Experimentation Result 1

예제프로그램	소스	바이트코드	CFG
Fibonacci	42	76	54
BobbleSort	30	79	68
Sieve	26	133	61
Hash	33	92	50
Ackerman	15	33	26

〈표 4〉에서 보면 바이트코드의 크기가 가장 크다는 것을 알 수 있다. 생성된 CFG가 크기가 큰 이유는 분석과 판독을 용이하게 하기 위하여 여러 가지 정보를 삽입 했기 때문이다.

아래 〈표 5〉는 CFG에서 SSA Form으로 변환 시 발생하는 노드 개수에 대한 비교를 나타내고 있다. CFG node는 CFG에서 나타나는 노드들의 수를 나타내고 있고, SSA node는 CFG에서 SSA Form으로 변환 시 생성된 전체 노드 수를 나타내고 있다. 〈표 5〉에서 보면 SSA Form으로 변환 시 노드의 개수가 PhiStmt 만큼 증가 하는 것을 알 수 있다.

표 5. 실험 결과 2
Table 5. Experimentation Result 2

예제프로그램	CFG node	SSA node
Fibonacci	108	116
BobbleSort	117	131
Sieve	158	185
Hash	128	139
Ackerman	53	53

VI. 결론 및 향후 과제

프로그램 분석과 최적화를 위하여 바이트코드에서 SSA Form으로 변환 시 몇몇 변수는 타입 정보를 상실한다. SSA Form에 올바른 타입이 주어지는 것은 매우 중요한 일이다. 왜냐하면 SSA Form에서 바이트코드로 역컴파일시 각 변수는 변수의 모든 정의와 사용에 대해 정적으로 선언된 알맞은 타입을 요구하기 때문이다.

본 논문에서는 바이트코드에서 SSA Form으로 변형을 위하여 먼저 바이트코드에 알맞게 확장된 CFG를 구성하였다. 확장된 CFG정보와 본 논문에서 제안한 BNF를 이용하여 트리구조의 3-주소 형태 문장을 구성하였다. 또한 3-주소 형태 문장을 SSA Form으로 변환하기 위하여 CFG를 구성하였고, CFG를 이용하여 지배자와 직접 지배자를 계산하였다. 그 후 계산된 정보를 이용하여 DF를 구한 후 알맞은 위치에 Ø-함수를 삽입 하여 SSA Form을 구성하였다.

구성된 SSA Form에 알맞은 타입을 설정하기 위하여 클래스 간 호출 관계 그래프와 상속 관계 그래프를 구성하였다. 구성된 그래프들은 객체 타입 전파를 위하여 중요한 정보가 된다. 그리고 각 노드를 방문하여 노드들의

구성 정보를 수집했다. 노드들의 구성 정보를 수집 후 동등한 타입을 가질 수 있는 노드들을 발견하기 위하여 타입 제약 모델과 알고리즘을 제안 하였다. 이렇게 구해진 동등한 노드에 대한 강 결합 요소를 설정한 후 각 노드에 알맞은 타입을 설정 하였다.

향후에는 배열 구조에 대한 타입 설정 문제와 SSA Form에서의 별칭 문제에 대한 타입 기반의 해결 그리고 역컴파일 수행시의 타입 검증 및 최적화에 관한 연구를 수행 할 것이다.

참고문헌

[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] Taiana Shpeismans, Mustafa Tikir, "Generating Efficient Stack Code for Java", Technical report, University of Maryland, 1999.

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

[5] J. Gil and A. Itai. "The complexity of type analysis of object-oriented programs." In Proc. of ECOOP '98 1988.

[6] H.Pande and B.Ryde. "Static type determination for C++." IN proc of the sixth USENIX C++, Apr. 1994.

[7] Alexxander Lenart, Christoper Sandler, and Sandeep K. S. Gupta. "SSA-based Flow-Sensitive Type Analysis." In Proc. ACM'00, pp. 813-817, March 2000.

[8] Jens Palsberg and Michael I. Schwartzbach. "Object-Oriented Type Systmes." IN Proc. ACM OOPLA'91 pp. 146-161 Apr. 1991.

[9] 김경수, 유원희, "바이트코드 분석을 위한 중간코드에 관한 연구", 한국컴퓨터정보학회논문지, 제11권 제1호, pp. 107-117p, 2006(3).

[10] Joshua Engel. "Programming for the Java Virtual Machine." Addison Wesley Longman. 1999.

[11] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, "Compilers Principle, Techniques, and Tools." Addison Wesley, 1996.

저 자 소 개

김 지민

2003년 2월 : 단국대학교 전산학과 이학사

2004년 ~ 현재 : 인하대학교 컴퓨터정보공학과 석사

〈관심분야〉 컴파일러, 최적화

김 기태

1999년 2월 : 상지대학교 전산학과 이학사

2001년 2월 : 인하대학교 전자계산공학과 석사

2001년 3월 ~ 현재 : 인하대학교 컴퓨터정보공학과 박사

〈관심분야〉 컴파일러, 프로그래밍 언어, 정보보안

김 제민

2002년 2월 : 인하대학교 컴퓨터정보공학과 공학사

2006년 2월~ 현재 : 인하대학교 컴퓨터정보공학과 석사

〈관심분야〉 컴파일러, 최적화

유 원희

1975년 2월 : 서울대학교 응용수학과 이학사

1978년 2월 : 서울대학교 계산학과 이학석사

1985년 2월 : 서울대학교 계산학과 이학박사

1979년 ~ 현재 : 인하대학교 컴퓨터정보공학부 교수

〈관심분야〉 컴파일러, 프로그래밍 언어, 병렬시스템