Static Type Assignment for SSA Form in CTOC

Ki-Tae Kim* and Weon-Hee Yoo*

Abstract: Although the Java bytecode has numerous advantages, it also has certain shortcomings such as its slow execution speed and difficulty of analysis. In order to overcome such disadvantages, a bytecode analysis and optimization must be performed. The control flow of the bytecode should be analyzed; next, information is required regarding where the variables are defined and used to conduct a dataflow analysis and optimization. There may be cases where variables with an identical name contain different values at different locations during execution, according to the value assigned to a given 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 achieve this, variables can be expressed using a static single assignment form. After transformation into a static single assignment form, the type information of each node expressed by each variable and expression must be configured to perform a static analysis and optimization. Based on the basic type information, this paper proposes a method for finding the related equivalent nodes, setting nodes with strong connection components, and efficiently assigning each node type

Keywords: *Bytecode, control flow graph, Static Single Assignment, Static Type Assignment*

1. Introduction

Although a bytecode has numerous useful features, it is a stack-based code [1,2]. With a slow execution speed, it has the disadvantage of not being an adequate expression for program analysis or optimization. There is also the problem of a relatively large size of constant pool – which was designed to support dynamic linking in the JVM. Therefore, in order for the Java class file to be effectively executed under an execution environment such as a network, it is necessary to convert it into an optimized code.

A control flow analysis of the bytecode should first be first performed for bytecode analysis and optimization [3,4]. After performing a control flow analysis, there is a requirement of which one should be aware, wherein the variables are defined and used in order to perform the data flow analysis and optimization. There may be cases where variables with an identical name contain different values at different locations, according to the value assigned to a variable in each location. Therefore, in order to statically determine the value and type, variables must be separated according to assignment. In order to achieve this, the Static Single Assignment (SSA) Form is used [5]. After transformation into an SSA Form, the information must be configured for each node displaying each variable and expression, in order to perform a static analysis and optimization. This paper describes the process of finding the related nodes and setting a type for each node, based on the basic type of information.

Manuscript received October 16, 2006; accepted March 13, 2007.
Corresponding Author: Ki-Tae Kim
*Dept. of Computer Science & Information Engineering, Inha University, Incheon, Korea (kkt@inha.ac.kr, whyoo@inha.ac.kr)

The structure of this paper is as follows. Chapter 2 explains the process of creating a CFG as well as the BNF for expressing the tree within the block; Chapter 3 describes the process of generating the SSA Form; Chapter 4 describes the process of establishing types; Chapter 5 describes the experiment and its results, and, finally, Chapter 6 contains the conclusion as well as suggestions for future research.

2. Control Flow Graph

Fig. 1 displays the example source program used in this paper. The implementations of the CFG, SSA Form and Type settings are presented using this example.

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

```
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
```

Fig. 1. (a) Source (b) Bytecode

Decompiling Fig. 1(a) using *javap –c* yields the results displayed in Fig. 1(b). *ifeq 11* in Fig. 1(b) is the bytecode command to the branch to where Label 11 is *"if equal"*. 11: is the label indicating the command offset.

Copyright ⓒ 2007 KIPS (ISSN 1738-8899)

2.1 Basic Block

The Basic Block is created for generating the CFG. The Basic Block consists of commands that do not affect the control flow in the CFG. A *leader* that affects the control flow within the program is found to create the Basic Block. The Basic Block has the form of locating a label that can identify each block in the first statement, and a branch statement is located in the last statement for transferring control to another block. The location where the label is added refers to the *lineNumbers* array that exists in the *.class* file. The line number information of the source program that has been in the *Line Number Table*, as well as information regarding the start location, is recorded within the *lineNumbers* array. In order to create a block with label information, a new block is created using label information.

The created CFG is written with a directed graph, and each node in the graph consists of a Basic Block. In order to create the CFG efficiently, the *entry node*, *exit node* and *initial node* are added as three basic nodes. *The entry node* is used to notify the entry point of the graph, and the *exit node* is used to alert the graph's exit point. The *initial node* is used to initialize information such as the parameters used in the graph. Each block used in the CFG includes label information.

2.2 Tree-Type Statements

Statements that exist in the block are added in Tree Form. The Tree used for this experiment is called the *"expression tree"*. The *expression tree* is largely categorized into expressions derived from the *Expr* class, an abstract class, and into statements derived from the *Stmt* class, which also represents an abstract class. Fig. 2 displays a part of the BNF that constructs the *expression tree*.

```
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
```

Fig. 2. BNF

The derivative classes of *Expr* and *Stmt* are created through the BNF shown in Fig. 2. The difference between the two class types is that the classes derived from *Expr* contain the *val* field for maintaining values. Moreover, each *Expr* has a *type* field for expressing the type. In this paper, static types are configured for each field, in order to propagate the node type using the *type* field. *Stmt* simply expresses the Tree Form.

Fig. 3 displays the created CFG.

```
<block_16 entry>
   label_16

<block_17 init>
   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 exit>
   label_18
```

Fig. 3. The created CFG

The CFG in Fig. 3 is created by a total of seven Basic Blocks, and the statements inside the blocks are added in Tree Form. The CFG is used as input in the SSA Form.

3. Implementation of the SSA Form

Generally, in order to statically handle a variable, the variable has to be split according to its *definition* and *type*, since a single variable may contain different values or types at different locations according to its *definition* and *use*. Therefore, in order to statically determine the *value* and *type*, the variable must be split according to its assignment.

3.1 Variable Information and Dominator Tree

In order to execute conversion to the SSA Form, the CFG created earlier must be visited to collect information regarding the variables *def* and *use* in the program. To achieve this, the tree created for the corresponding statement for each block in the CFG must be visited in the *depth-first* to find the variables *def* and *use* in each block.

The dominance frontier must be acquired to insert the ϕ-function into the *join node* of the CFG. The *dominator tree* must first be created in order to obtain the *dominance frontier*.

Furthermore, to obtain the *dominance frontier*, the dominance relationship between the blocks must be known, to which end the index for the current block must be acquired and the search for the *immediate dominator* of the current block performed.

To find the *immediate dominator*, the blocks corresponding to the graph nodes are sought to retrieve the pre-order of the current block to the variable *i*. Check if *i* is currently the root. If not, directly find the dominator. The formula for calculating the dominator is presented in (1). From the dominator of the current block, eliminate what is dominating the current block's dominator, from which the current block is eliminated.

$$\text{idom} := \text{dom(block)} - \text{dom(dom(block))} - \text{block} \quad (1)$$

3.2 Dominance Frontier

After the CFG and the *dominator tree* have been created, the *dominance frontier* is calculated by inserting the ϕ-function. The location of ϕ-function insertion is called the *dominance frontier*. If the *dominance frontier* is denoted as DF[n], it can be expressed by (2).

$$DF[n] = DF_{local}[n] \cup DF_{up}[c] \quad (2)$$

$DF_{local}[n]$ in (2) denotes the set of nodes before n that are not directly dominated by node n. $DF_{up}[c]$ denotes the set of *dominance frontier* nodes that are not dominated by nodes indirectly dominated by node n.

3.3 Iterated Dominance Frontier

When the location of insertion for the ϕ-function is determined through the *dominance frontier*, the ϕ-function is inserted. In a structural language, there can be multiple places where ϕ-functions can be inserted, since in structural language, *join nodes* are created in specific locations, and the ϕ-function is generally inserted where the *join nodes* are located. However, most branches in the Java bytecode consist of conditional *if* branches and unconditional *goto* branches that can be expressed simply.

If the definition for the corresponding variable exists in another block but not in the current block, the variable is defined using information in the other block. In order to achieve this, the *iterated dominance frontier* is recalculated using the *dominance frontier* value calculated earlier. The *dominance frontier* acquired from the iteration is the location where the ϕ-statement is to be inserted.

3.4 Rename

This paper expresses the statement with the ϕ-function inserted as *PHIStmt*. Rename the two *Local_ref2_UDef* variables present in the *PHIStmt* created. First check whether there is a definition of the corresponding variable *Locali2* in each block using the CFG and the *dominator tree*. In the case of *<block_0>,* the definition of *Local_ref2_2* exists. Configure the information regarding the existing variable on the top of the field to signify the top of the stack. From the block corresponding to the top block's successor, again find the definition for the corresponding block. If the variable's definition exists, information regarding the new variable is inserted on top of the stack. For *<block_11>*, the new value of *Local_ref2_4* exists on top of the stack. For *<block_13>*, *Local_ref2_UDef*, the operand of one of the *PHIStmts* is from *<block_11>*. The corresponding variable is renamed as *Local_ref2_4* present in the current stack. The result so far indicates *Local_ref2_10 := Phi(Local_ref2_4, Local_ref2_UDef).* The result of renaming for the variables of all blocks is *Local_ref2_10 := Phi(Local_ref2_4, Local_ref2_6)*. This process establishes a clear name for each variable of the *PHIStmt* added.

3.5 Adding PHIStmt

Add the corresponding statement to the block where *PHIStmt* is to be inserted. Currently all statements exist in the form of the linked list. Therefore, add *PHIStmt* which was created after the label statement *label_13*. Fig. 4 displays the graph after *PHIStmt* has been added.

```
<block label_16 entry>
    label_16

<block label_17 init>
    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 exit>
    label_18
```

Fig. 4. Static Single Assignment Form

As displayed in Fig. 4, all blocks and variables within the blocks are assigned new names, according to the definitions and usage of the variables. The modified version number becomes important data for type inference and optimization.

4. Type Assignment

As presented in Fig. 4, in the completed SSA Form, all blocks and block variables are assigned new names according to the variables *def* and *use*. However, since not all nodes have type information, types must be assigned to those nodes without types using the existing information.

4.1 Finding the Equivalent Node

In order to assign types to nodes whose variables are present, the Basic Blocks of the CFG with the SSA Form must be visited according to the *pre-order,* to find the equivalent nodes. The Basic Block's statements are constructed in Tree Form. Therefore, after finding the Basic Block, the corresponding tree must be visited in order to examine the nodes constituting each statement, as presented in Fig. 4. In *IfZStmt* of *<block_0> if0(Local_ref1_1==0) then <block_11> else <block_6>, Local_ref1_1* is the case of *VarExpr*. If the corresponding node is *VarExpr*, then the corresponding expression should be verified to see if it was defined earlier so as to define the expression and find an equivalent node. For the case in Fig. 4, *Locali1_1* of *<block_17>* is equivalent with *Local_ref1_1*, the type of which is not defined in *<block_0>*. Since *<block_17>* is an initialization node, it has information regarding the format parameters. Therefore, rather than ref signifying that the type is yet to be determined, it has *i* signifying the integer type. The algorithm for obtaining an equivalent node is presented in Algorithm 1.

Algorithm 1. equivalent node

```
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
```

In Algorithm 1, *equiv* is assigned as *HashMap*. According to the algorithm, *equiv* of *VarExpr* becomes {*Local_ref1_1* = [*Local_ref1_1,Locali1_1], Locali1_1= [Local_ref1_1, Locali1_1*]}. In other words, *Local_ref1_1* and *Locali1_1* are the equivalent nodes and are assigned identical types in the future.

Table 1 displays the entire *equiv* for the equivalent nodes created according to Algorithm 1.

Table 1. Information of the equivalent nodes

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]

The locations where equivalent nodes can be created are trees with nodes having variables or constants. As presented in Fig. 2, these nodes can be created in *VarExpr, PhiStmt*, and *StoreExpr,* which include the *ExprStmt, IfZeroStmt, PhiJoinStmt, ReturnExprStmt* statements in the BNF. The reason for equivalent 1: and 3: being created is to prevent redundant creation, since equivalent nodes can exist in other blocks.

4.2 Assigning a Number to Each Node

After finding the equivalent nodes through the SSA Form, a number is assigned to each node. When setting types, there can be several nodes of the same type, and a number is to be assigned to each node to distinguish the nodes. Based on the *TreeVisitor* class, all trees of the Basic Blocks are visited *depth-first*, and the count field value is assigned to each node. For Fig. 4, the entire node is 31. In other words, the tree consists of 31 nodes in total.

4.3 Type Settings

Set a type in an appropriate location. In Fig. 4, examine the statement in *<block_11>: ExprStmt : eval (Local _ref2_4 := 3) {15}*. {15} denotes the node number, which is the count value. The *ExprStmt* structure is displayed in Fig. 5.

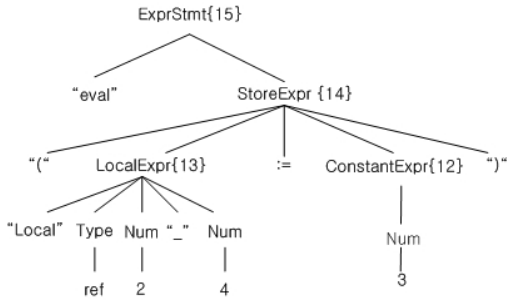

Fig. 5. ExprStmt

First insert the node currently being visited into the stack. Then find a node equivalent to the current node of *ExprStmt*. At this point, only the node is the equivalent node. If there is a child node, visit this node to find a node equivalent to the child node. According to Fig. 5, the next child node visited is *StoreExpr*. If an equivalent node equiv is found with *StoreExpr : (Local_ref2_4 := 3){14}*, it will be similar to 6: in Table. 1. Therefore, an equivalent node of *ConstantExpr : 3{12}, StoreExpr : (Local_ref2_4 := 3){14}, LocalExpr : Local_ref2_4{25}, LocalExpr : Local_ref2_4{13}* can be obtained. There are two cases of the equivalent type *LocalExpr*. Since these are different nodes, they are distinguished by attaching an index to the count field value calculated earlier.

The *strong connection component*, which has the form of the array list, must be obtained in order to configure the type. If the stack is not empty, remove the node located on top of the stack, acquire a node equivalent to the eliminated node, and add it to the *strongly connection* node. In the example above, since *StoreExpr : (Local_ref2_4 := 3){14}* existed on the current stack, the corresponding node is eliminated from the stack, and [*Local_ref2_4, (Local_ref2_4 := 3), 3, Local_ref2_4*], which is an equivalent node to the eliminated node, is added to the *strongly connection component*. Then, each node that corresponds to each component is visited to set the type. *ConstantExpr, VarExpr, StoreExpr,* and *PhiStmt* are nodes that can be assigned a type, as shown in Fig. 2. For example, if the *strongly connection component* currently visited is *ConstantExpr : 3{12}*, the value of the expression is first retrieved to the value field. In terms of a constant, a value can be a character or a number. Therefore, they must be processed differently according to the circumstances. The *instanceof* operator is used to verify which kind of constant it is. In the case where the value of the *instanceof* Integer is *true*, the value type is an integer. Consequently, the type can be determined from the value. Once the type has been assigned to the current *strongly connection component*, the remaining components will also be assigned with types. Therefore, the process of configuring integer types to all of the *strongly connection components* [*Local_ref2_4, (Local_ref2_4 := 3), 3, Local_ref2_4*] is performed. Similarly, node types can be assigned to the remaining nodes. Fig. 6 displays the final *static single assignment form* graph with all types assigned.

```
<block_16>
  label_16

<block_17>
  label_17
    INIT Local_ref0_0 Locali1_1
      *[Type] : type(Local_ref0_0) = LTemp;
      *[Type] : type(Locali1_1) = Z
    goto label_0

<block_0>
  label_0
  label_2
    if0 (Local_ref1_1 == 0) then <block_11> else <block_6>
        *[Type] : type(Locali1_1) = Z

<block_6>
  label_6
    eval (Local_ref2_6 := 2)
    *[Type] : type(2) = I
    *[Type] : type(Locali2_6) = I
    *[Type] : type((Locali2_6 := 2)) = I
    goto label_13

<block_11>
  label_11
    eval (Local_ref2_4 := 3)
    *[Type] : type(3) = I
    *[Type] : type(Locali2_4) = I
    *[Type] : 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)
    *[Type] : type(Locali2_6) = I
    *[Type] : type(Locali2_4) = I
    *[Type] : type(Locali2_10) = I
    return Local_ref2_10

<block_18>
  label_18
```

Fig. 6. Typed Static Single Assignment Form

In Fig. 6, [*Type*] : *type(Local_ref0_0) = LTemp;* indicates the class type located in the 0^{th} position of the local variable. Since the Java virtual machine expresses the object type as *L*, this paper also expressed the object type in the form of *LTemp* as *Local_ref_0_0*. [*Type*] : *type(Locali1_1) = Z* indicates that the first type of the first local variable is of the *boolean* type, denoted as *Z* in the JVM. [*Type*] : *type{3} = I* signifies that node 3 is of the integer type. In turn, types are statically assigned to all documents and nodes.

5. Experiment and Results

The experiment was conducted using a Pentium 4 2.4 GHz processor, with 512MB of RAM, Eclipse 3.1 was used for writing and testing CTOC, Editplus 2.11 was used to print the bytecode, and a Java compiler j2sdk1.4.2_03 was used.

We analyzed 6 case examples that could be used to compare the control flow. These data – which appeared in Don Lance's paper - are used to compare the experiment results [6]. In Table 1, simple explanations of the program that we used are presented.

Table 1. Examples

Program	Explanation
SquareRoot	Finding the square root
SumOfSquareRoot	Finding the sum of the square root of a number. That is from 1 to n
Fibonacci	Finding fibonacci number n.
QuickSort	Sorting the Integer array using Quicksort
LabelExample	Labeled break and continue program
Exceptional	Exception handling of try-catch-finally

Experiment items using the example of Table 1 include the number of lines of each program's source code, the number of bytecode lines, the number of lines after transforming the code, the number of basic blocks, the number of edges, and the total number of nodes.

Table 2. Results of Experiment

Program	source	bytecode	after trans	basic blocks	edges	nodes
SquareRoot	37	94	60	15	18	99
SumOfSquareRoot	38	103	63	18	19	108
Fibonacci	42	76	69	18	22	86
QuickSort	30	79	68	16	21	101
LableExample	28	51	59	13	16	58
Exceptional	41	99	149	26	29	143

In Table 2, ***source*** represents the number of lines of the source code, ***bytecode*** represents the number of lines of the bytecode generated by javap –c. ***after trans*** represents the number of code lines that are added and eliminated in transforming the original code to the basic block in CTOC. ***basic block*** represents the number of basic blocks generated by the leader for the transformed code and the basic block. ***edge*** represents the number of edges that show a relationship among the basic blocks. ***node*** represents the number of nodes used to identify instructions and statements in a basic block.

Table 3 shows the results of CFG and SSA Form after conversion.

Table 3. SSA Form results

Program	CFG lines	SSA lines	%	CFG nodes	SSA nodes	%
SquareRoot	60	63	4.76	99	117	15.38
SumOfSqure Root	63	71	11.27	108	143	24.48
Fibonacci	69	77	10.39	86	126	31.75
QuickSort	68	76	10.53	101	133	24.06
LabelExample	59	63	6.35	58	74	21.62
Exceptional	147	177	15.82	143	304	52.96

In Table 3, we can see that the number of lines and the number of nodes increase after conversion into the SSA Form.

Table 4 shows the results of static type decisions in the SSA Form.

Table 4. Example of Experiment

Program	Basic block	Statements	Type
SquareRoot	15	42	75
SumOfSquareRoot	18	47	83
Fibonacci	16	48	60
QuickSort	16	82	125
LabelExample	13	35	35
Exceptional	26	89	107

In Table 4, *Basic block* means the number of basic blocks created in the SSA Form, and *Statements* means the number of statements used in the SSA Graph. *Type* shows each node at process that static type assignment is achieved however many type decisions happen. For example, the SquareRoot program is divided into a basic block of 15, including 42 statements in tree form, and 75 type decisions occur after SSA Form conversion

6. Conclusion

The Java bytecode has recently been used as a means of intermediary expression in various fields. However, the disadvantages of a slow execution speed and difficulty in performing analysis exist. Therefore, optimization and program analysis are required to expedite the execution speed and enhance program understanding.

This paper involves the analysis of programs from the bytecode level and preparation for optimization. In order to perform analysis and optimization at the bytecode level, the CFG was first created. However, due to the bytecode characteristics, the existing control-flow analysis techniques were expanded to suit the bytecode. Furthermore, in order to perform a static analysis, the CFG was converted into the SSA form. For conversion into the SSA Form, a considerable level of calculation was carried out regarding the CFG, *dominance relationship*, *dominator tree*, *immediate dominator*, *ϕ-function*, *rename* and *dominance frontier*.

After conversion into the SSA form, the equivalent nodes were obtained and found, and then configured as *strong connection components*, and an identical type was assigned to each node. The SSA Form with the type assigned acts as an adequate input to the optimization process to be performed in later stages.

References

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

[2] James Gosling, Bill Joy, and Guy Steel, “The Java Language Specifications,” The Java Series, Addison Wesely, 1997.
[3] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilers Principles, Techniques and Tools*, Addison Wesley, 1986.
[4] Andrew W. appel, *Modern Compiler Implementation in Java*. CAMBRIDGE UNIVERSITY PRESS, 1998, pp 437-477.
[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, “Efficiently Computing the Static Single Assignment Form and the Control Dependence Graph”, March 1991, pp 451-490.
[6] Don Lance, “Java Program Analysis: A New Approach Using Java Virtual Machine Bytecodes”, http://www. mtsu.edu/~java

Ki-Tae Kim
Kim received an MS degree in Computer Science from Inha Univ. in 2001. During 2004~2006, he worked as a full-time instructor at Inha University, and he is now undertaking a doctorate course as a member of the programming environment lab at Inha Univ. His research interests include Compiler, Programming Language and the Semantic Web.

Weon-Hee Yoo
Yoo received a Ph.D. degree in Computer Engineering from Seoul University in 1985. He has been a professor at Inha Univ. since 1979. His research interests include Compiler, Programming Language, Program Slicing, and XML access control.