저작자표시 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에만 하여 자유롭게

- 이 저작물의 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.
- 이 저작물을 작성할 수 있습니다.
- 이 저작물을 업리 목적으로 이용할 수 있습니다.

다음과 같은 조건을 따라서 합니다.

BY: 저작자표시. 권하는 원저작자를 표시하여야 합니다.

- 권하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내야 합니다.
- 저작권자로부터 범도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 범위에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 더하하기 쉽게 요약한 것입니다.

Disclaimer ☞

collection
A Practical Application of the Method for Translation into Static Single Assignment Form
A Practical Application of the Method for Translation into Static Single Assignment Form

이 논문은 석사학위 논문으로 제출함.

仁荷大學校 大學院
 컴퓨터・情報工學科

杜麗丹
이 논문을 다니의 석사 학위 논문으로 인정함.

2007년 8월 6일

[인하대 학교장의 서명]

[부서장의 서명]

[위원의 서명]
Abstract

Static single assignment (SSA) form is an intermediate representation which encodes information about data flow and control flow that compilers use to facilitate program analysis and optimization. In SSA form each variable is defined by a single operation and each use of a variable is reached by one assignment. The obvious single property simplifies data flow. Thus it becomes increasingly popular in language processors.

This research illustrates the practical application of the method for translation into SSA form for Java bytecode level analysis and optimization. To construct the control flow graph and dominator tree, and to calculate dominance frontiers are required related work. In the process of translation there are two separate steps: placing $\emptyset$-functions and renaming variables. In the first step, three variations minimal SSA, semi-pruned SSA and pruned SSA are all implemented. In the second step each use of variables and arguments in $\emptyset$-functions are renamed. Experimental results efficiently show that SSA form linearly increases the size of the original program and pruned SSA always generates the least number of $\emptyset$-functions among three variations.
Contents

Abstract .......................................................................................................................... i
Contents .......................................................................................................................... ii

Chapter 1 Introduction ................................................................................................. 1
Chapter 2 Related work ................................................................................................. 5
  2.1 Generating control flow graph ............................................................................. 6
  2.2 Constructing dominator tree ............................................................................... 11
  2.3 Calculating dominance frontier ......................................................................... 13
Chapter 3 Translation into SSA form .......................................................................... 16
  3.1 Placing $\emptyset$-functions ..................................................................................... 16
    3.1.1 Iterated dominance frontier ......................................................................... 16
    3.1.2 Three variations ............................................................................................. 19
      3.1.2.1 Minimal SSA form .................................................................................. 20
      3.1.2.2 Semi-pruned SSA form ......................................................................... 22
      3.1.2.3 Pruned SSA form .................................................................................. 25
  3.2 Renaming variables ............................................................................................... 30
Chapter 4 Experimental Results and Evaluation ......................................................... 36
Chapter 5 Conclusion ................................................................................................... 40
References ...................................................................................................................... 42
Acknowledgements ....................................................................................................... 44
List of Figures

[Figure 2.1] Method test() in source code ........................................... 6
[Figure 2.2] Control flow graph for test() ............................................ 9
[Figure 2.3] Flow graph for test() ..................................................... 10
[Figure 2.4] Dominator tree for test() ................................................ 13

[Figure 3.1] Flow of use and definition of x, y and z .............................. 20
[Figure 3.2] Minimal SSA for test() .................................................... 22
[Figure 3.3] Semi-pruned SSA for test() ............................................. 25
[Figure 3.4] Pruned SSA for test() ...................................................... 30
[Figure 3.5] Pruned SSA form based CFG for test() ............................. 35

[Figure 4.1] Comparison between CFG lines and SSA lines .................. 37
List of Tables

[Table 2.1] Calculation of Dom[i] ................................................................. 12
[Table 2.2] Immediate dominators for test() .............................................. 12
[Table 2.3] Dominance frontiers for test() .................................................. 15

[Table 3.1] Iterated dominance frontiers for variable z ............................... 19
[Table 3.2] Nonlocal variables detecting process for test() ......................... 24
[Table 3.3] Live variables analysis for x, y and z ........................................ 29
[Table 3.4] Rename process for z in pruned SSA ......................................... 32

[Table 4.1] Example programs ...................................................................... 36
[Table 4.2] Comparison between CFG and SSA nodes ................................. 37
[Table 4.3] Lines in CFG and three variations ............................................. 38
List of Algorithms

[Algorithm 2.1] Computing dominance frontiers ........................................... 14

[Algorithm 3.1] Computing iterated dominance frontiers ............................. 17
[Algorithm 3.2] Placing $\emptyset$-functions ..................................................... 21
[Algorithm 3.3] Detecting non-local variables .................................................. 23
[Algorithm 3.4] Analyzing live variables ......................................................... 28
[Algorithm 3.5] Renaming variables ............................................................... 31


Chapter 1 Introduction

There are some benefits of using a machine-independent intermediate form. Retargetting can be facilitated; a machine-independent code optimizer can be applied to the intermediate representation. In the process of translating a source program into a target program, a compiler may construct a sequence of intermediate representations.

Static single assignment (SSA) form is an intermediate representation which encodes information about data flow and control flow that compilers use to facilitate program analysis and optimization. The value of a variable is the contents of the memory cell or cells associated with the variable. Variables are names of values. Each variable must have a location to store its value, and one variable can hold multiple values throughout its lifetime.

Two obvious basic properties of SSA form are given: each variable is defined by a single operation in the code—hence the term static single assignment; each use of variable refers to a single definition. To distinguish between the definitions of a variable, the SSA name for the variable is used, denoted with a subscript, e.g. Vi. If multiple definitions reach a use, a special operation called ∅-function is inserted into the merge point where different control flow paths meet. The ∅-function returns the value of its argument that corresponds to the control flow path that was taken to get to the assignment statement containing the ∅-function [1].

∅-functions provide the compiler with information about the flow of
values. Compiler can use this information to improve the quality of the code that it generates. The name space eliminates any issues related to the lifetime of a value. Since each value is defined in exactly one instruction, it is available along any path that proceeds from that instruction. These properties simplify and improve many optimization techniques.

Non-SSA form intermediate code is put into SSA form, which can be optimized in various ways, and then translated back out of SSA form. In the process of translation into SSA form many questions will be raised. For instance, if we consider simple variable splitting, it is obvious that many new temporary variables are introduced. Redundant $\emptyset$-functions should be considered to be removed away.

As far back as 1970 it is known that Shapiro et al. proposed the precursor of SSA form which satisfied the property that each definition of a variable is reached by exactly one assignment to that variable. By inserting explicit $\emptyset$-functions, SSA form leads to simpler formulations of works like that are based on the precursor. It was only in 1989 however an efficient and practical algorithm for computing SSA was first formulated by Cytron et al. [2,3]. Since that time, there have been valiant attempts to improve the efficiency of basic SSA form. An obvious literature is in 1995 Sreedhar and Gao [4] employed a novel program representation – the DJ graph hence they introduced a linear time algorithm for placing $\emptyset$-nodes. Also DAS and Ramakrishna [5] proposed an iterative algorithm for $\emptyset$-function computation using DJ graphs with respect to a basic block instead of a definition. In
2002 Ramalingam [6] used loop-nesting forests for calculating the iterated dominance frontier. But the Cytron et al. algorithm has remained one of the most useful and efficient in practice.

A key step in translating into SSA form is to place $\varnothing$-functions. About the process of placing $\varnothing$-functions there are three different variations proposed. The original paper of Cytron et al. presented an algorithm for constructing SSA form code for a procedure, this version of SSA is called minimal SSA [2,3]. A subsequent paper of Choi and Cytron [7] presented a more complex and expensive algorithm that produces a smaller version of SSA, called pruned SSA. In 1998, Briggs et al. [8] improved third trade-off variation of SSA form between minimal SSA and pruned SSA, called semi-pruned SSA form.

This thesis is a practical efficient application of SSA form for java bytecode level research, the design and implementation of Cytron et al.’s method for translation into SSA form, as well as an application of fundamental ideas to yield useful intermediate representations suitable for other classes of analysis. These research is to design a platform translator for Java bytecode level analysis and optimization [9].

The flow of control information is included in control flow graphs, the dominator information is included in dominator trees. Therefor, control flow graphs and dominator trees should be constructed in advance. Using dominance frontier to calculate iterated dominance frontiers, then for each variable $\varnothing$-functions are placed into them. Each use of global variables
should be renamed, each argument in placed $∅$-functions should be renamed. In the implementation process, it is shown clearly that data flow (use and definition chain) information and control flow information becomes more compact because of single property of SSA form. Hence SSA form facilitate analysis of program and optimization.

The rest of this thesis is organized as follows. Chapter 2 provides the related work before translation into SSA form, including constructing control flow graph and dominator tree, calculating dominance frontier. Chapter 3 introduces the two steps of translation: placing $∅$-functions and renaming variables. Whereas the evaluation of the experimental results is shown in chapter 4. In the last chapter the conclusion will be given.
Chapter 2 Related Work

Java source code is compiled into bytecodes, i.e. .java file is compiled into *.class file, so that it can be executed by Java Virtual Machine (JVM). Java class or classes are edited and java methods are modeled. Java methods have a lot of stuff: names, codes, parameters and exceptions etc. In the modeled process, method variables and parameters are represented by local variables referenced by JVM instructions. A JVM instruction can be created from an opcode and an operand.

JVM instructions are partitioned into basic blocks. A basic block is a sequence of consecutive instructions in which flow of control enters at the beginning and leaves at the end. Each basic block has an associated expression tree which represents nested nature of code. An expression tree consists of two kinds of nodes representing arithmetic operations, method invocation, stack manipulation and exception handler etc. One is expressions which have value associated with it; the other is statements which have no value associated with it. VarExpression represents an expression that accesses local or stack variables. Each VarExpression has an integer index associated with it. It has two subclasses: LocalExpression and StackExpression. There are some commonly used statements such as GotoStatements, IfStatements and so on. When translation into SSA form, special statements called $\emptyset$ -Statements are nodes in expression trees.
2.1 Generating control flow graph

Since control flow information is encoded by SSA form, control flow graph for a code should be generated in advance. Flow of control information is added to the set of basic blocks, making up a program, constructing a directed graph called a control flow graph (CFG). The nodes in a control flow graph are basic blocks. When constructing control flow graph, entry block and exit block are added.

```
public class Example {
    int test(boolean b) {
        int r,s,t;
        int x,y,z;
        if ( b ) {
            x = 1;
            r = x;
            y = 2;
            z = 3;
            if ( y > 1 ) s = y;
        } else {
            x = 100;
            r = x;
            y = 200;
            z = 300;
            if ( y > 3 ) s = y;
        }
        t = z;
        return t;
    }
}
```

Figure 2.1 Method test() in source code

A substantial example is considered here in order to illustrate and
understand the process more clearly. Figure 2.1 gives the method test() in
Java source code. Method test() includes three conditional expressions: three
nested if-statements.

The associated control flow graph for method test() is generated in Figure
2.2. Basic block is represented as <BL_number HD = L_number CFGType
BlockType>. BL_number means the first label of this block is L_number, its
offset to code array is “number”. HD means the header of this CFG. There
are several block types: IN for an entry block; OUT for an exit block; INIT
block is for initialization of method’s name and parameters. If one block is
none of the above three types no block type is displayed. Meanwhile there
are three control flow graph types: nonheader graph, irreducible graph or
reducible graph. For instance, <BL_63 HD = L_62 NON INIT> means the first
label in block 63 is label 63: the loop header of this block is block 62. The
type of it’s control flow graph is nonheader. This is a block for initialization
of method’s name and parameters. Block 62 is added as the entry node and
block 64 as the exit node to construct control flow graph.

When a method’s local variable is referenced, if it is allocated at the
stack, it is represented as StackTypeIndex_Version or _undef; otherwise it is
represented as LocalTypeIndex_Version or _undef. Type includes integer,
character, long, short etc. _Version means this variable have a version of all
defined variables. _undef means a variable has not been given a designed
name. Local5_14 represents the type of local variable x is integer, its index
to loaded data is 5 and its version is 14. Local5_undef after label L_7
represents unnamed variable x which is used to assign to another variable. Each variable has its prototype LocalTypeIndex. Variable y and z in source code are privately represented as Locali6 and Locali7.

When a variable is assigned a constant or other variables, it is presented as an expression statement: evaluation(LocalTypeIndex_Version := a constant or variable). For instance, evaluation(Local5_14 := 1). If statements can change control flow of code. The true and false targets refer to two blocks. For example, in block 4 if (Local6_undef <= 1) is true, control goes to block 54; if false, control flows to block 22.
Figure 2.2 Control flow graph for test()
Three conditional expressions: if-statements consists of the control flow in method test(). First conditional branch if is located at block 0, branch then is located at block 28 and branch else is located at block 4. In block 4 there is another nested if-statement. Hence two out flow paths are generated, one path is from block 4, 22 to 54: the other is from block 4 to 54 directly. The corresponding flow graph is shown in Figure 2.3.

From Figure 2.3, it is easy to figure out that there are totally five control flow paths. Control may flow along one path: block 62, 63, 0, 4, 22, 54 to 64. Starting from block 62, 63, 0, 28, 51, 54 to 64 is another of the paths. Block 54 is the merging point from block 4, 22, 28 and 51, where variable z (Locali7) is assigned to variable t and the value of t is returned.
2.2 Constructing Dominator Tree

A useful way of presenting dominator information is a tree, called dominator tree, in which the initial node is the root, and each node dominates only its descendants in the tree. In the dominator tree, the parent of each node is its immediate dominator. For the root node, its dominator is itself. The following data flow equation (1) is to calculate each block’s dominators.

\[ \text{Dom}(X) = \{X\} \cup \bigcap_{P \in \text{Predecessors}(X)} \text{Dom}(P) \]  

(1)

In this equation, each block X is a dominator of itself. All dominators of each predecessor P of block X in the graph are operated with logical AND.

The traversal algorithm through control flow graph is depth first search. The dominators is concretely represented by a vector of bits, \( \text{dom}[i] \), i is the traversed order. The high limit of i is the number of blocks in the graph, \( \text{Graph.size()} \). For root node, \( \text{dom}[0] \) is assigned by 0, i.e. itself; for others (\( i = 1,2,\ldots \)), \( \text{dom}[i] \) is initialized by the whole blocks (0,1,2,\ldots,\( \text{Graph.size()}\)-1).

For method test(), the traversed block order starting from 0 to 8 is block 62, 63, 0, 28, 51, 54, 64, 4 and 22. The value of \( \text{dom}[i] \) is calculated as shown in Table 2.1. Compared the second calculated value with the first one, the value set of block 54 is changed from (0,1,2,3,5) to (0,1,2,5), i.e. from (62,63,0,28,54) to (62,63,0,54). The third pass is seen to produce no changes, so the values after second time yield the relation dominators of method test().
Each block $X$ has a unique immediate dominator, $IDom(X)$. For immediate dominator, there are some properties: $IDom(X)$ is unequal to $X$; $IDom(X)$ dominates $X$; $IDom(X)$ does not dominate any other dominator of $X$. $IDom(X)$ is calculated using the following equation (2):

$$IDom(X) = Dom(X) - \bigcup_{D \in Dom(X)} Dom(D) - \{X\} \tag{2}$$

In this equation, each block $X$ is not $IDom(X)$. All dominators of $Dom(X)$ should be excluded by $IDom(X)$.

### Table 2.2 Immediate dominators for test()

<table>
<thead>
<tr>
<th>Block $X$</th>
<th>(62)</th>
<th>(63)</th>
<th>(0)</th>
<th>(28)</th>
<th>(51)</th>
<th>(54)</th>
<th>(64)</th>
<th>(4)</th>
<th>(22)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$IDom(X)$</td>
<td>(62)</td>
<td>(63)</td>
<td>(0)</td>
<td>(28)</td>
<td>(51)</td>
<td>(54)</td>
<td>(64)</td>
<td>(4)</td>
<td>(22)</td>
</tr>
</tbody>
</table>

All $IDom(X)$s for test() are listed in Table 2.2. The dominator tree for test() can be drawn as shown in Figure 2.4. From this picture, it is clearly to figure out that block 0 should be three blocks’ dominator: 4, 54 and 28. Block 28 should be the dominator of block 51, which is a branch in nested if-statement. Meanwhile block 62, 63 and 0 are dominators of the all nodes which follow block 0 in the dominator tree.
2.3 Calculating Dominance Frontier

Given a block X in CFG, the dominance frontier of X, DF(X), is a set of blocks Ys, which is the first block (in the series of edges from block X) that is not dominated by block X. To compute the dominance frontier linear in the size mapping, two intermediate sets $DF_{\text{local}}(X)$ and $DF_{\text{up}}(Z)$ for each block are defined such that holds the equation (3).

$$DF(X) = DF_{\text{local}}(X) \cup \left( \bigcup_{Z \in \text{Children}(X)} DF_{\text{up}}(Z) \right)$$

(3)

Algorithm 2.1 is to calculate dominance frontier using a bottom-up traversal. Based on the equation (3), $DF_{\text{local}}(X)$ are calculated from each successor of X, Y. When immediate dominator of Y, IDom(Y), is not equal to X, Y is a dominance frontier of X. For $DF_{\text{up}}(Z)$, children of block X in dominator tree are assigned to block Z. First is to calculate DF(Z). For each block Y in DF(Z), when the immediate dominator of Y is unequal to X, then
Y is one of dominance frontiers of X.

| Input: control flow graph and dominator tree  |
| Output: Dominance frontiers for each X         |
| procedure calculateDF                           |
| begin                                          |
| 1: for each X in a bottom-up traversal of the dominator tree do |
| 2: \( \text{DF}(X) := \emptyset \)            |
| 3: for each \( Y \in \text{Succ}(X) \) do    |
| 4: if \( \text{idom}(Y) \neq X \) then        |
| 5: \( \text{DF}(X) := \text{DF}(X) \cup \{Y\} \) |
| 6: fi                                          |
| 7: od                                          |
| 8: for each \( Z \in \text{Children}(X) \) do |
| 9: calculateDF(Z)                              |
| 10: for each \( Y \in \text{DF}(Z) \) do      |
| 11: if \( \text{idom}(Y) \neq X \) then        |
| 12: \( \text{DF}(X) := \text{DF}(X) \cup \{Y\} \) |
| 13: fi                                         |
| 14: od                                         |
| 15: od                                         |
| 16: end                                        |

Algorithm 2.1 Computing dominance frontiers

The dominance frontiers for example code can be tabulated in Table 2.3. All successors of block X are responding to control flow graph rather than dominator tree. Children of block X means blocks in dominator tree. All blanks in the Table 2.3 mean no block for that column.
The dominance frontier for each block is listed in the last column. Block 62 and 64 have no dominance frontier. For block 63, 0, and 54, the calculated dominance frontier is block 64. For block 28, 51, 4 and 22, the calculated dominance frontier is block 54. As described in the beginning of this chapter, block 64 is added as an exit block together with entry block 62 in order to construct CFG.
Chapter 3 Translation into SSA Form

The method for translating a non-static single assignment form into static single assignment form is to place $\varnothing$-functions for variables, and then to rename the variables and arguments in $\varnothing$-functions.

3.1 Placing $\varnothing$-functions

The key step to translate into SSA form is to add a $\varnothing$-function for every variable at each merge point, that is each node in the control flow graph with more than one predecessor. $\varnothing$-functions are referred to as merge operators or choice operations. Defining a $\varnothing$-function for a variable in a block means adding a definition of the variable in the block. A $\varnothing$-function generally has as many arguments as it has incoming predecessor edges. Each argument corresponds to the name of the relevant variable along a particular control flow path.

3.1.1 Iterated dominance frontier

Where to place $\varnothing$-functions has been proved by Cytron et al. [2,3]: the set of nodes that need $\varnothing$-functions for any variable $V$ is the iterated dominance frontier $DF^+(S)$, where $S$ is the set of assignments for $V$. The dominance frontier of $S$ is extended from the mapping relation between the dominance frontier and nodes as $DF(S) = \bigcup_{X \in S} DF(X)$. The iterated
dominance frontier, \( DF(S) = \lim_{i \to \infty} DF_i(S) \), is the limit of the increasing sequence of sets of nodes.

Input: A control flow graph with entry block; a set of blocks \( S \), calculated \( DF(X) \)
Output: iterated dominance frontier \( DFplus(S) \)
procedure calculateDFplus
begin
1: \( DFplus(S) := \emptyset \)
2: inworklist := \( S \)
3: worklist := \( S \)
4: while (worklist \neq \emptyset ) do
5: remove the first block \( X \) from worklist
6: calculateDF(X)
7: for each \( Y \in DF(X) \) do
8: \( DFplus(S) := DFplus(S) \cup \{ Y \} \)
9: if \( Y \notin inworklist \) then
10: inworklist := inworklist \cup \{ Y \}
11: worklist := worklist \cup \{ Y \}
12: fi
13: od
14: od
end

Algorithm 3.1 Computing iterated dominance frontiers

Algorithm 3.1 describes the process to compute iterated dominance frontier. The actual implementation of \( S \), the set of assignment blocks for variable \( V \), is a Collection. The computation of \( DFplus(S) \) is an efficient HashSet. Inworklist is assigned by a Hashset as a temporary set of variables. Worklist is assigned by an efficient LinkedList whose elements are
inworklist HashSets. This algorithm is performed once for each variable V.
In the while loop, one of the blocks – the first block X in worklist is
removed. DF(X) have been calculated in chapter 2.3 as a Collection. Method
iterator() iterates over all elements in DF(X). Each block Y is obtained by
next() method and is added into DFplus(S). If Y is not contained by
inworklist, added to inworklist and worklist. The iteration will be ended
when the result of worklist.hasNext() becomes empty.

For method test(), variable z (Locali7) is considered and analyzed here. In
both block 28 and 4 variable z is defined. S for variable z is the two blocks.
Worklist is \( \{28,4\} \) now. Block 28 is removed from worklist at first and
worklist is changed to be \( \{4\} \). DF(\( \{28\} \)) is already known to be \( \{54\} \) in
Table 2.3. Block 54 is added into DFplus(S), DFplus(S) equals to \( \{54\} \). It is
needed to add \( \{54\} \) into inworklist and worklist since \( \{54\} \) has not been
contained by them. Inworklist is changed to be \( \{(28,4,54)\} \) and worklist
\( \{(4,54)\} \). \( \{4\} \) is removed from worklist at the next time. DF(\( \{4\} \)) is \( \{54\} \).
DFplus(S) remains no change. Continually taking \( \{54\} \) away from inworklist,
DF(\( \{54\} \)), i.e. \( \{64\} \) should be added into both DFplus(S), inworklist and
worklist. Then DFplus(S) is \( \{(54,64)\} \), whereas inworklist remains no change
and worklist is empty since DF(\( \{64\} \)) is empty in Table 2.3.

Table 3.1 gives the process of calculating iterated dominance frontiers for
z (Locali7). After four loops, worklist becomes empty while the final
DFplus(\( \{4,28\} \)) are set \( \{54,64\} \). The computing process for variable x and y
is quite same with z. In Figure 2.3, block 54 is the merge point of two
different paths.

Table 3.1 Iterated dominance frontier for variable z

<table>
<thead>
<tr>
<th></th>
<th>Initialization</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>(28,4)</td>
<td>(28,4)</td>
<td>(28,4)</td>
<td>(28,4)</td>
<td>(28,4)</td>
</tr>
<tr>
<td>Block X</td>
<td></td>
<td>(28)</td>
<td>(4)</td>
<td>(54)</td>
<td>(64)</td>
</tr>
<tr>
<td>Y=DF(X)</td>
<td></td>
<td>(54)</td>
<td>(54)</td>
<td>(64)</td>
<td></td>
</tr>
<tr>
<td>Inworklist</td>
<td></td>
<td>(28,4)</td>
<td>(28,4,54)</td>
<td>(28,4,54,64)</td>
<td>(28,4,54,64)</td>
</tr>
<tr>
<td>Worklist</td>
<td></td>
<td>(28,4)</td>
<td>(4,54)</td>
<td>(54)</td>
<td>(64)</td>
</tr>
<tr>
<td>DFplus(S)</td>
<td></td>
<td>(54)</td>
<td>(54)</td>
<td>(54,64)</td>
<td>(54,64)</td>
</tr>
</tbody>
</table>

3.1.2 Three Variations

Another side of the process of placing $\emptyset$-functions is which variables need $\emptyset$-functions. Three proposed variations answer this question: minimal SSA form, semi-pruned SSA form and pruned SSA form.

Part of code in Figure 2.2 is referenced to explain the three variations. Only variables x, y and z are considered and their separate prototype are Locali5, Locali6 and Locali7. In the source code, two if-statements are nested into the first if-statement. A flow graph is drawn to show the use and definition information of x, y and z in Figure 3.1. The three variables are all defined in block 4 and 28, which privately belong to two different control flow paths. The use block of x is the same with its definition block at each flow path. Block 22 and 51 use y those are different with blocks that define it. Whereas z is only used at the merge point of the two flow paths, block 54.
3.1.2.1 Minimal SSA Form

Minimal SSA is the basic form. It inserts $\emptyset$-functions at every point where control flows merge together. Minimal means that the definition is a "minimal" description, but the greatest number of $\emptyset$-functions will be inserted among the three variations of SSA form. This placement may introduce some unnecessary $\emptyset$-functions.

After the iterated dominance frontiers have been calculated, algorithm 3.2 is to place $\emptyset$-functions into them. First is to get set of definition blocks $S$ for variable $V$. $\text{DFplus}(S)$ can be calculated by calling the algorithm 3.1. The exit block is the one added in order to construct the control flow graph, therefore no $\emptyset$-function is placed at it. When translation into SSA form, $\emptyset$-functions are placed before any statements in a block.
Input: A control flow graph, calculated iterated dominance frontier DFplus(S)
Output: iterated dominance frontier with \(\emptyset\)-functions

procedure placePhiFunctions
begin
1: for each variable V is contained in CFG do
2: \(S := \text{definitionblocks}(V)\)
3: calculateDFplus(S)
4: for each basic block \(X \in \text{DFplus}(S)\) do
5: if \(X \neq \{\text{exit}\}\) then
6: get target for V and put it at the left side of \(\emptyset\)
7: for each \(Y \in \text{Predecessors}(X)\) do
8: get the operand name of V
9: put the operand mapping with Y at the right side of \(\emptyset\)
10: od
11: fi
12: od
13: od
end

Algorithm 3.2 Placing \(\emptyset\)-functions

\(\emptyset\)-functions are considered as special assignment statements. The target of V, a SSA name with union version, is implemented as an object of VarExpression and located at the left side of \(\emptyset\)-statements. Each operand with \_undef postfix, mapped with block Y by a HashMap data structure, is put at the right side of \(\emptyset\)-statements as an argument. The number of arguments equals to the number of block X’s predecessor edges.

For variable \(z\) (Locali7), the target is Locali7_35 which constructs its \(\emptyset\)-statement at the left side: Locali7_35 := PHI(). X is block 54, there are four incoming edges, its predecessor blocks are block 28, 51, 4 and 22. Hence four arguments will be inserted. The positions of four arguments in the \(\emptyset\)
function are mapped with their own block Y, and separately inserted as four steps. In this example, the first argument is mapped with block 28, hence \( \text{Locali7}_{35} := \text{PHI} (\text{Locali7}_{\text{undef}}) \) is generated. Then the second is mapped with block 4; the third is mapped with block 51; the last argument is mapped with block 22. Finally, the \( \emptyset \)-function has four arguments:

\( \text{Locali7}_{35} := \text{PHI} (\text{Locali7}_{\text{undef}}, \text{Locali7}_{\text{undef}}, \text{Locali7}_{\text{undef}}, \text{Locali7}_{\text{undef}}) \).

The process of variable \( x \) and \( y \) is the same with \( z \), their target names are \( \text{Locali5}_{40} \) and \( \text{Locali6}_{50} \). Another two unnecessary \( \emptyset \)-functions are inserted for variable \( r \) (Locali2) and \( s \) (Locali3). Figure 3.2 gives the final inserted \( \emptyset \)-functions in minimal SSA form for test() method.

```plaintext
<BL_54 HD = L_62 NON>
  \text{Locali6}_{50} := \text{PHI} (\text{Locali6}_{\text{undef}}, \text{Locali6}_{\text{undef}}, \text{Locali6}_{\text{undef}}, \text{Locali6}_{\text{undef}})
  \text{Locali2}_{45} := \text{PHI} (\text{Locali2}_{\text{undef}}, \text{Locali2}_{\text{undef}}, \text{Locali2}_{\text{undef}}, \text{Locali2}_{\text{undef}})
  \text{Locali5}_{40} := \text{PHI} (\text{Locali5}_{\text{undef}}, \text{Locali5}_{\text{undef}}, \text{Locali5}_{\text{undef}}, \text{Locali5}_{\text{undef}})
  \text{Locali7}_{35} := \text{PHI} (\text{Locali7}_{\text{undef}}, \text{Locali7}_{\text{undef}}, \text{Locali7}_{\text{undef}}, \text{Locali7}_{\text{undef}})
  \text{Locali3}_{30} := \text{PHI} (\text{Locali3}_{\text{undef}}, \text{Locali3}_{\text{undef}}, \text{Locali3}_{\text{undef}}, \text{Locali3}_{\text{undef}})
```

Figure 3.2 Minimal SSA for test()

### 3.1.2.2 Semi-pruned SSA Form

Briggs et al. proposed an intermediate form named semi-pruned SSA form between minimal SSA and pruned SSA [8]. Semi-pruned SSA form requires any "non-local" variable to be found in advance. When a variable \( V \) used in a basic block has not been defined before this use in the basic block, \( V \) is recognized as a non-local variable, and nonlocal flag is set true.
Algorithm 3.3 Detecting nonlocal variables

In Algorithm 3.3, flag variable defined is an object of BitSet with the number of blocks in CFG and initialized to false. Each real occurrence of variable R is iteratively checked. Block X can be gotten by calling getBlock() method. The index i of block X in CFG is achieved by preorder. Then defined.set(i) method sets the ith bit true if R is defined in X; otherwise, defined.get(i) equals to false, it means R is occurred and used in block X but is not defined in block X before this use. Therefore R is not a local variable, flag nonlocal is set to true. When flag nonlocal is true it is needed to place ∅-function for the variable in iterated dominance frontiers. This algorithm is related to the total number of variable’s uses and definitions that need to be processed. In semi-pruned SSA form, algorithm
3.3 should be executed before placing $\phi$-functions.

Actually, only for used variables it needs to be detected as non local variables or not. Table 3.2 shows the detecting results of used variables of $x$ (Locali5), $y$ (Locali6) and $z$ (Locali7) and ignores the defined variables of them in test() method.

**Table 3.2 Non local variables detecting process for test()**

<table>
<thead>
<tr>
<th>Use of variables</th>
<th>Used block</th>
<th>Defined block</th>
<th>Nonlocal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locali5_undef</td>
<td>(4)</td>
<td>(4)</td>
<td>false</td>
</tr>
<tr>
<td>Locali5_undef</td>
<td>(28)</td>
<td>(28)</td>
<td>false</td>
</tr>
<tr>
<td>Locali6_undef</td>
<td>(22)</td>
<td>(4)</td>
<td>true</td>
</tr>
<tr>
<td>Locali6_undef</td>
<td>(51)</td>
<td>(28)</td>
<td>true</td>
</tr>
<tr>
<td>Locali7_undef</td>
<td>(54)</td>
<td>(4,28)</td>
<td>true</td>
</tr>
</tbody>
</table>

In the control flow from entry (62), (63), (0), (4), (22), (54) to exit (64), variable $y$ (Locali6) are defined in block 4 and used in block 22, therefor variable Locali6_undef in block 22 is a non local variable. So the situation in the control flow from entry (62), (63), (0), (28), (51) to exit (64) is the same, variable Locali6_undef in block 51 is a non local variable. Variable $z$ (Locali7_undef) used in block 64 is defined in block 4 or 28, it is a non local variable. $\phi$-functions for two variables $y$ (Locali6) and $z$ (Locali7) are needed and placed them in the iterated dominance frontier (54) using algorithm 3.2, $\phi$-function for local variable $x$ (Locali5) in minimal SSA is deleted. Here the target names for $y$ and $z$ becomes Locali6_35 and Locali7_30. Figure 3.3 shows the semi-pruned SSA form for test() method.
3.1.2.3 Pruned SSA Form

In minimal SSA form, the dominance frontier correctly captures the flow of values to determine where to insert $\emptyset$-functions but ignores the data flow facts – knowledge about the lifetimes of values from analyzing their definitions and uses. Choi et al. [7] proposed another variation – pruned SSA form. Pruned SSA form needs to perform liveness analysis to define the set of variables that are live on entry to the block and to remove those $\emptyset$-functions for some dead variables.

A variable is live on an edge if there is a directed path from that edge to a use of the variable that does not go through any definition [7]. Liveness information can be calculated from use and definition as following equations (4) and (5) [10]:

$$In(X) = Use(X) \cup (Out(X) - Def(X)) \quad (4)$$

$$Out(X) = \bigcup_{S \in \text{Successors}(X)} In(S) \quad (5)$$

In these equations, $In(X)$ means a variable is live-in at a basic block $X$ if it is live on any of the in edges of that block; $Out(X)$ means it is live-out at a basic block $X$ if it is live on any of the out edges of the block. $Use(X)$ is the set of all variables used in block $X$. $Def(X)$ is the set of all variables defined in block $X$. The boundary condition is specified as no variable is
live on exit from the program: In((exit)) = ∅. In pruned SSA form, if variable V is live-in into block X, then insert ∅-functions for variable V; otherwise no ∅-functions for it.

In public method block(), two member data block.Use and block.Def defined as Set structure, represent Use(X) and Def(X). Each variable V is represented as an object of VarExpression class. The block X that a real occurrence variable is being can be assigned using getBlock() method. If the real occurrence of variable is defined, then its prototype is added into Set block.Def; otherwise its prototype is added into Set block.Use. After each real occurrence of variable R in CFG is iteratively checked, two Sets block.Use and block.Def for every block have been calculated.

Algorithm 3.4 iteratively solves the equations (4) and (5) using a backward algorithm. Boolean variable change is initialized to true and used to find a fixed point for the while loop. Block.In and block.Out are defined as two Sets data structure for In(X) and Out(X) and initialized as empty. Flag size represents the number of blocks in CFG. Each block X in CFG is iteratively accessed in postorder. At the beginning of the while loop, the original block.In is saved as block.OldIn in order to find out any change between them. Successors Ys of block X is an iterator object and each Y is iteratively checked by calling next() method. Each variable V in In(Y) is added to Out(X) when it has not yet been contained by In(X). In this algorithm, each element in four Sets block.Use, block.Def, block.In and block.Out is the prototype of variable V rather than the real occurrence of it.
To calculate In(X), first is to add each variable that has not been
tained by set block.In. A temporary set block.Temp (i.e. Temp(X)) refers
to block.Out in order to calculate the difference between block.Out and
block.Def. When variable V in block.Def has been contained by block.Temp,
V should be removed away from block.Temp. Finally block.In is calculated
by combining variables both in block.Temp and block.In. When size
becoming zero, each block.In for whole blocks in CFG has the same set
ements with block.oldIn, i.e. the fixed point for the while loop is achieved.

Input: A control flow graph, calculated Use(X) and Def(X).
Output: In(X) and Out(X)

procedure analyzeLiveVariables
begin
  1: change := true
  2: for each basic block X \in CFG do
  3:     In(X) := \emptyset
  4:     Out(X) := \emptyset
  5:     od
  6: while change == true do
  7:     size := number of CFG basic blocks
  8:     for each X \in CFG and X != {exit} do
  9:         OldIn(X) := In(X)
 10:     for each block Y \in Succ(X) do
 11:         for each variable V \in In(Y)
 12:             if V \in Out(X) then
 13:                 Out(X) := Out(X) \cup V
 14:             fi
 15:     od
 16:     od
end
Algorithm 3.4 Analyzing Live Variables

The live variable calculation is global. This algorithm is related to the number of blocks, the number of edges in the control flow graph, and the total number of variable’s uses and definitions that need to be processed.
Table 3.3 Live variables analysis for x, y and z

<table>
<thead>
<tr>
<th>Block X</th>
<th>Succ(X)</th>
<th>Use(X) Def(X)</th>
<th>1st</th>
<th>2nd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Out(X)</td>
<td>In(X)</td>
</tr>
<tr>
<td>9</td>
<td>(64)</td>
<td></td>
<td></td>
<td>Local7</td>
</tr>
<tr>
<td>8</td>
<td>(54)</td>
<td>Local6</td>
<td>Local7</td>
<td>Local6</td>
</tr>
<tr>
<td>7</td>
<td>(22)</td>
<td>Local5</td>
<td>Local6</td>
<td>Local5</td>
</tr>
<tr>
<td></td>
<td>(54)</td>
<td>Local5</td>
<td>Local6</td>
<td>Local5</td>
</tr>
<tr>
<td>6</td>
<td>(4)</td>
<td>Local6</td>
<td>Local7</td>
<td>Local6</td>
</tr>
<tr>
<td></td>
<td>(22,54)</td>
<td>Local6</td>
<td>Local7</td>
<td>Local6</td>
</tr>
<tr>
<td>5</td>
<td>(51)</td>
<td>Local6</td>
<td>Local7</td>
<td>Local6</td>
</tr>
<tr>
<td></td>
<td>(54)</td>
<td>Local6</td>
<td>Local7</td>
<td>Local6</td>
</tr>
<tr>
<td>4</td>
<td>(28)</td>
<td>Local5</td>
<td>Local6</td>
<td>Local5</td>
</tr>
<tr>
<td></td>
<td>(51,54)</td>
<td>Local5</td>
<td>Local6</td>
<td>Local5</td>
</tr>
<tr>
<td>3</td>
<td>(0)</td>
<td>Local5</td>
<td>Local6</td>
<td>Local5</td>
</tr>
<tr>
<td></td>
<td>(4,28)</td>
<td>Local5</td>
<td>Local6</td>
<td>Local5</td>
</tr>
<tr>
<td>2</td>
<td>(63)</td>
<td>Local5</td>
<td>Local6</td>
<td>Local5</td>
</tr>
<tr>
<td></td>
<td>(0)</td>
<td>Local5</td>
<td>Local6</td>
<td>Local5</td>
</tr>
<tr>
<td>1</td>
<td>(62)</td>
<td>Local5</td>
<td>Local6</td>
<td>Local5</td>
</tr>
<tr>
<td></td>
<td>(63,64)</td>
<td>Local6</td>
<td>Local6</td>
<td>Local5</td>
</tr>
</tbody>
</table>

For method test0, live variables for each block X is backward calculated as listed in Table 3.1. Variables x (Local5), y (Local6) and z (Local7) are considered only. Use((64)), Def((64)), Out((64)) and In((64)) are always empty for it is an exit block. Out(X) are calculated before In(X). After the second calculation, there is not any change between In(X) and OldIn(X) for each block X. Hence the iteration is fixed.

Table 3.2 shows that only variable z (Local7) lives in block 54. It means that only one $∅$-function for variable z is inserted in the iterated dominance frontier. $∅$-functions for variable x and y in minimal SSA are deleted in pruned SSA. The target name of z is Local7_30. Four arguments are placed
into the $\varnothing$-Statement. The pruned SSA form for test() is shown in Figure 3.4:

\[
\text{Locali7}_{30} := \text{PHI}(\text{Locali7}_{\text{undef}}, \text{Locali7}_{\text{undef}}, \text{Locali7}_{\text{undef}}, \text{Locali7}_{\text{undef}})
\]

Figure 3.4 Pruned SSA form for test()

### 3.2 Renaming Variables

In our practical method, after the $\varnothing$-functions are placed, each argument in $\varnothing$-functions is renamed by distinct definitions coming from different incoming control flow edges; each use of variable \( V \) is renamed to use the most recent definition of \( V \). The control flow graph is traversed in preorder fashion, beginning from the entry block.

In algorithm 3.5 a stack is used to keep trace of the most recent definition of each variable. This algorithm is considered as several main steps. First of all is to get the $\varnothing$-Statement in block \( X \) since $\varnothing$-Statements are always placed at the front of a block. The target of the $\varnothing$-Statement, considered as a definition, is pushed into toposfstack. topofstack is an object of VarExpression and its elements are variables.

At the second step, each real occurrence of variables \( R \) is checked. When a definition of a variable is encountered in block \( X \), push it into toposfstack as the most recent SSA name of the variable; if a use of a variable is encountered, pop the top of the stack to this real occurrence \( R \), i.e rename \( R \)
by toposfack.

---

Input: A control flow graph with placed $\emptyset$-functions  
Output: SSA form based CFG  

procedure renameVariables  
begin  
1: get $\emptyset$-Statement in block X  
2: toposfack := the target for variable V  
3: for each real occurrence of variable $R \in$ block X do  
4: \quad if $R$ is a definition then  
5: \quad \quad toposfack := $R$  
6: \quad else  
7: \quad \quad $R := $ toposfack  
8: \quad fi  
9: od  
10: for each block $Y \in$ Successors(X) do  
11: \quad get the $\emptyset$-statement for $R$ in $Y$  
12: \quad get the current operand of the $\emptyset$-statement in block X  
13: \quad operand := toposfack  
14: od  
15: for each block $Z \in$ Children(X) do  
16: \quad renameVariables(Z)  
17: od  
end  

Algorithm 3.5 Renaming variables

At the third step, each successor $Y$ of block $X$ in CFG is iteratively visited. If there exits a $\emptyset$-Statement in block $Y$, get the operand as an object of VarExpression, and use toposfack to replace it. The mapping between block $Y$ and the position of operand is implemented calling get(Y)
method by an object of HashMap data structure. Finally, renameVariables() is recursively invoked for each child of block X in the dominator tree of its control flow graph.

In Figure 2.2 there are uses of variables x, y and z, which are represented as Locali5 undef in block 28 and 4, Locali6 undef in block 51 and 22, Locali7 undef in block 54. The rename process for variable x and y are similar with z. For variable z (Locali7) it is illustrated in pruned SSA for method test() in Table 3.4.

**Table 3.4 Renaming process for z in Pruned SSA**

<table>
<thead>
<tr>
<th>Block X</th>
<th>Top of stack</th>
<th>Real occurrence R in X</th>
<th>Succs(X)</th>
<th>Renamed statement and variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>(62)</td>
<td></td>
<td></td>
<td>(63,64)</td>
<td></td>
</tr>
<tr>
<td>(63)</td>
<td></td>
<td></td>
<td>(0)</td>
<td></td>
</tr>
<tr>
<td>(0)</td>
<td></td>
<td></td>
<td>(28,4)</td>
<td></td>
</tr>
<tr>
<td>(54)</td>
<td>Locali7_30</td>
<td>Locali7 undef</td>
<td>(64)</td>
<td>Evaluation(Local4_10 := Locali7_30)</td>
</tr>
<tr>
<td>(28)</td>
<td>Locali7_7</td>
<td>Locali7_7</td>
<td>(54)</td>
<td>PHI(Local7_7,Local7 undef, Locali7 undef,Locali7 undef)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(54)</td>
<td>Locali7_30 := Locali7_7,Locali7_18, Locali7 undef,Locali7 undef)</td>
</tr>
<tr>
<td>(51)</td>
<td>Locali7_7</td>
<td></td>
<td>(54)</td>
<td>PHI(Local7_7,Local7_18, Locali7 undef,Locali7 undef)</td>
</tr>
<tr>
<td>(4)</td>
<td>Locali7_18</td>
<td>Locali7_18</td>
<td>(54)</td>
<td>Locali7_30 := Locali7_7,Locali7_18, Locali7 undef,Locali7 undef)</td>
</tr>
<tr>
<td>(22)</td>
<td>Locali7_18</td>
<td></td>
<td>(54)</td>
<td>PHI(Local7_7,Local7_18, Locali7 undef,Locali7 undef)</td>
</tr>
<tr>
<td>(64)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The recursive starts from entry block 62 in dominator tree. In block 54, one $\emptyset$-function is encountered and the target value Locali7_30 is assigned to topofstack. Afterwards, the use of $z$ Locali7_undef in statement Evaluation := (Locali4_10 := Locali7_30) is renamed by topofstack, Locali7_30. In block 28, a definition of $z$ Locali7_7, is encountered and push it into topofstack. Block 54 is one successor block of (28), the detected $\emptyset$-function in it is Locali7_30 := PHI(Local7_undef, Local7_undef, Local7_undef, Local7_undef). The first argument is mapped with block 28 and rename it with Local7_7. The $\emptyset$-function is replaced as Locali7_30 := PHI(Local7_7, Local7_undef, Local7_undef, Local7_undef). Block 51 is the sequential visited child block, mapping with the third argument. The $\emptyset$-function is renamed as Locali7_30 := PHI(Local7_7, Local7_undef, Local7_7, Local7_undef). So do other two arguments. The final $\emptyset$-function is Locali7_30 := PHI(Local7_7, Local7_18, Local7_7, Local7_18).

Each variable is examined in test(). Additionally, in (0) Locali1_undef for variable b is changed into Locali1_1; in (54) Locali4_undef for variable t is changed into Locali4_10. Comparison with Figure 2.2, it is clear that each use of variables is replaced by it’s recent definition name in Figure 3.5.
<BL_62 HD = null NON In>
  L_62
<BL_63 HD = L_62 NON INIT>
  L_63
    INIT Local_ref0_0 Local1_1
    goto L_0
  L_61
<BL_0 HD = L_62 NON>
  L_0
    if0 (Local1_1 == 0)
      then <BL_28 HD = L_62 NON>
    else <BL_4 HD = L_62 NON>
<BL_4 HD = L_62 NON>
  L_4
    evaluation (Locali5_14 := 1)
  L_7
    evaluation (Locali2_16 := Locali5_14)
  L_10
    evaluation (Locali6_17 := 2)
  L_13
    evaluation (Locali7_18 := 3)
  L_16
    if (Locali6_17 <= 1)
      then <BL_54 HD = L_62 NON>
    else <BL_22 HD = L_62 NON>
<BL_22 HD = L_62 NON>
  L_22
    evaluation (Locali3_21 := Locali6_17)
    goto L_54
<BL_28 HD = L_62 NON>
  L_28
    evaluation (Locali5_3 := 100)
  L_32
    evaluation (Locali2_5 := Locali5_3)
L_35
Figure 3.5 Pruned SSA form based CFG for test()
Chapter 4 Experimental Results and Evaluation

The experiments are presented to evaluate the method for translation into SSA form and to compare three variations: minimal SSA, semi-pruned SSA and pruned SSA form. The experiments are carried out on Pentium 4 2.0 GHz processor, 512MB Ram. The source programs and test programs are run in Eclipse 3.2.1, Java compiler j2sdk1.4.2_09 is used.

In our experiments, four other programs in Java source code: SquareRoot, Fibonacci, BubbleSort and LableExample are tested together with example program referred in chapter 2 and 3. Table 4.1 gives the explanation of the four programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SquareRoot</td>
<td>To compute square root of a number</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>To compute fibonacci progression where the next value is the sum of the current and previous values</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>To perform a series of passes over items in an array in non decreasing order</td>
</tr>
<tr>
<td>LabelExample</td>
<td>To use label after break and continue statement</td>
</tr>
</tbody>
</table>

Table 4.2 shows the statistic numbers of expression tree nodes for control flow graph and for three variations of SSA form based control flow graph. The target values and parameters in placed $\emptyset$-statements are nodes added in expression trees, which rise the original nodes.
Table 4.2 Comparison between CFG and SSA nodes

<table>
<thead>
<tr>
<th>Program</th>
<th>CFG nodes</th>
<th>Minimal nodes</th>
<th>Semi-pruned nodes</th>
<th>Pruned nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SquareRoot</td>
<td>100</td>
<td>143</td>
<td>9</td>
<td>143</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>87</td>
<td>127</td>
<td>8</td>
<td>127</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>102</td>
<td>150</td>
<td>12</td>
<td>134</td>
</tr>
<tr>
<td>LabelExample</td>
<td>60</td>
<td>76</td>
<td>4</td>
<td>72</td>
</tr>
<tr>
<td>Test</td>
<td>79</td>
<td>109</td>
<td>5</td>
<td>91</td>
</tr>
</tbody>
</table>

Figure 4.1 Comparison between CFG and minimal SSA nodes

When translation into SSA form, minimal SSA is the basic form proposed by Cytron et al. Figure 4.1 gives a comparison chart between CFG nodes and minimal SSA nodes of the five tested programs in Table 4.2. The columns of minimal SSA nodes in the chart are higher than those of CFG nodes greatly. The CFG nodes divided by the increased nodes in minimal SSA form equals the increasing node rate. Then among the five tested programs, the highest increasing rate is about 47.5% in BubbleSort program; even the lowest increasing rate is about 26.6% in LabelExample program.

Table 4.3 provides the statistic data about CFG lines and lines in three
variations of SSA form. Lines mean the counted number of instruction lines inside control flow graph or SSA form based control flow graph. One $\emptyset$-statement is counted as one instruction line.

Table 4.3 Lines in CFG and three variations

<table>
<thead>
<tr>
<th>Program</th>
<th>CFG lines</th>
<th>Minimal lines</th>
<th>Semi-pruned lines</th>
<th>Pruned lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>SquareRoot</td>
<td>51</td>
<td>62</td>
<td>62</td>
<td>62</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>58</td>
<td>66</td>
<td>66</td>
<td>63</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>55</td>
<td>67</td>
<td>63</td>
<td>62</td>
</tr>
<tr>
<td>LabelExample</td>
<td>47</td>
<td>51</td>
<td>51</td>
<td>50</td>
</tr>
<tr>
<td>Test</td>
<td>47</td>
<td>52</td>
<td>49</td>
<td>48</td>
</tr>
</tbody>
</table>

The situation for lines is similar with that for nodes. It is obvious that lines in SSA form based CFG increase greatly than original CFG lines. The increasing line rate is calculated as CFG lines divided by the difference between minimal SSA lines and CFG lines. The highest increasing line rate is 21.8% for BubbleSort program. The lowest increasing rate is 8.51% in LabelExample program.

It shows that after translation into SSA form based CFG the program codes including both nodes in expression tree and instruction lines in CFG are considerably increased because of inserted $\emptyset$-functions. Hence to reduce the number of $\emptyset$-functions is very important.

Table 4.2 and Table 4.3 provides experimental results among minimal SSA, semi-pruned SSA and pruned SSA form. The results shows clearly that the number of $\emptyset$-statements (lines) and nodes in minimal SSA are
always the most; those of pruned SSA are correspondingly lowest for most programs; those of semi-pruned SSA lie at the middle of them.

The numbers of $\emptyset$-statements in semi-pruned SSA for SquareRoot, Fibonacci and LabelExample program are the same with those of minimal SSA, although semi-pruned SSA cost time and space to detect non local variables. In pruned SSA, the decreased number of $\emptyset$-statements are obvious. The decreased rate is calculated as the number of $\emptyset$-statements in minimal SSA divided by the decreased number of $\emptyset$-statements between pruned SSA and minimal SSA. Because $\emptyset$-statements only for live variables in IDF are placed, the decreasing rate is about 37.5% in Fibonacci; and it is about 41.6% for BubbleSort program.

After the above comparison among three variations, it can be concluded that minimal SSA form generates greatest number of $\emptyset$-functions with minimal cost. The nodes and lines increase greatly in minimal SSA form. If someone would like to select semi-pruned SSA for trade-off because some non local variables are removed away, we can find that for SquareRoot, Fibonacci and LabelExample programs the number of $\emptyset$-functions between minimal and semi-pruned SSA is quite the same. Then we prefer to choice pruned SSA form which has fewest number of $\emptyset$-functions among three variations. Some $\emptyset$-functions for dead variables are unnecessarily since those variables are never used in iterated dominance frontiers. Although pruned SSA needs expensive global live variables analysis.
Chapter 5 Conclusion

Static single assignment (SSA) form is an efficient intermediate representation in the process that a compiler translates the source code into the target code. The Java source code is compiled into bytecode. Bytecode instructions are partitioned into basic blocks. The instructions in a basic block are represented by an expression tree. When control flow graph is translated into static single assignment (SSA) form, special statements called $\emptyset$-statements are nodes added to the expression tree and are placed into the graph. Both nodes in expression trees and instruction lines in control flow graph increases accordingly.

Each definition of a variable is modified to different SSA versions, and each use of the variable is modified to use the most recently defined variable version. Also the number of arguments in the placed $\emptyset$-function for a variable equals to the number of its value flow from predecessors. Each argument in $\emptyset$-functions is renamed by the recent definition version of a variable from its data flow. The size of program representation after translation into SSA form increases and is linear with that of original control flow graph.

The basic algorithm for placing $\emptyset$-functions is for minimal SSA form, which cost least but has greatest number of $\emptyset$-functions. Semi-pruned SSA and pruned SSA can be made by the basic algorithm for minimal SSA. Semi-pruned SSA need advance non-local variables detecting process, which
places less $\emptyset$-functions than minimal SSA but more than pruned SSA form. Pruned SSA costs relatively more expensive time and space to analyze the live variables in programs, but places minimal $\emptyset$-functions among three of them.

From the whole process, we can conclude that SSA form makes information of definition and use more compact or it is an improvement of def-use chain. Benefiting from its single assignment property and each use of variable referring to single name, SSA form simplifies data flow to efficiently facilitate and improve the analysis of program. Unrelated uses of the same variable in the source code become different variables in SSA form. Hence needless relationships are eliminated.

On the other hand, a great quantity of $\emptyset$-function is inserted so that the space of original program increases. Therefore, redundant $\emptyset$-functions for some variables should be removed to get more efficient code for better program analysis and optimization. $\emptyset$-functions for local variables in semi-pruned SSA and for dead variables in pruned SSA are not placed into iterated dominance frontiers any more. Pruned SSA is our flavor among the three variations because it efficiently decreases the number of placed $\emptyset$-functions.
References


Acknowledgements

The two years I studied and researched in Inha university is like a journey and it is a special period in my life. During these two years I have been supported and helped by many people. Now it is a good opportunity that I can express my gratitude to all of them.

The first person that I would like to express my gratitude is my direct supervisor - professor Weon-hee Yoo whom I have followed since from 2005. Professor Yoo is famous as a smart and principled researcher and respectable professor. I am full of gratitude for his instructions in my research.

Interdependence is certainly more valuable than independence. I would like to thank all the graduate students for doctorial course and master course in programming environment lab, who have shown me the most direct and practical help in my research: Ki-tae Kim, Je-min Kim, Sun-moon Jo, Kyoung-soo Kim and Young-kuk Kim.

A great sense of gratitude is devoted to my parents: Liang Du and Wei-xiang Cai, who have taught me the good things that really matter in my life. Dr Hong-wen Kang, my husband, a research scientist in climatology, have supported and inspired me a lot in my life. An excellent lovely boy Yi-yang Kang, my son, is the important one to whom I express my gratitude. His great progress in his growth have inspired me to improve myself continually.

I wish my family members and my friends share all happiness with me.
Li-dan Du
2007–08