

A PRACTICAL INTERPROCEDURAL ALIAS ANALYSIS  
FOR AN OPTIMIZING/PARALLELIZING C COMPILER

*by*  
*Maryam Emami*

School of Computer Science  
McGill University, Montreal

October 1993

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF  
MASTER OF SCIENCE

Copyright © 1993 by Maryam Emami

# Abstract

Accurate alias analysis is a crucial phase of optimizing/parallelizing compilers for languages which support pointer data structures. The result of the alias analysis is used by most of data-flow analyses and optimization phases in a compiler to produce efficient code.

In this thesis, we introduce a new approach for interprocedural alias analysis which determines the explicit *points-to* relationships between locations in an abstract stack at each program point. Two variables have a points-to relationship if one of them points, or may point, to the other one.

To perform our interprocedural analysis, we have designed a new representation to capture the call-structure of the program, called the *invocation graph*. This is similar to the traditional *call graph*, with some additional properties for the recursive function calls. The invocation graph is used to follow the exact sequence of function calls and returns in the program. In this manner, the precise information is propagated to/from a function.

This work has been integrated in the McCAT optimizing/parallelizing compiler for the C-language. The analysis handles both scalar variables and aggregate data structures (records and arrays).

The results of the points-to analysis provides a base upon which other interprocedural analyses are built. This thesis provides both experimental results and examples to demonstrate the usefulness of the points-to analysis.

# Résumé

L'analyse d'alias est une phase extrêmement importante de tout compilateur de langages supportant les pointeurs de données. Le résultat de cette analyse est en effet utilisé par le plupart des méthodes basées sur l'analyse de flots de données ainsi que de nombreuses techniques d'optimisations afin de générer du code efficace.

Dans ce rapport, on introduit une nouvelle approche pour l'analyse inter-procédurale d'alias permettant de déterminer les relations explicites de type *pointe sur* entre les éléments d'une pile abstraite à tout instant d'un programme. Deux variables sont liées par une relation de type *pointe sur* si l'une d'elles pointe, ou peut pointer, sur l'autre.

Notre analyse est basée sur une nouvelle structure de données appelée *graphe d'invocation* permettant de représenter précisément la hiérarchie d'appels et de retours de fonctions d'un programme. Cette structure est similaire aux traditionnels *graphes d'appels* tout en possédant des propriétés additionnelles pour le traitement des appels récursifs de fonctions. Le graphe d'invocation permet de déterminer la séquence exacte d'appels/retours de fonctions dans un programme. Il autorise la propagation d'une information précise vers (ou à l'extérieur) des fonctions.

Ce travail a été intégré dans le compilateur C McCAT. La méthode présentée traite de façon détaillée aussi bien les variables scalaires que les structures de données plus complexes de C, telles les tableaux ou les enregistrements.

Le contexte du travail présenté dans cette thèse est général et peut servir de base pour l'élaboration d'autres méthodes d'analyse inter-procédurales. Ce manuscrit présente à la fois des travaux expérimentaux ainsi que des exemples décrivant l'analyse de type *pointe sur* développée.

## Acknowledgements

I am thankful to Professor Laurie Hendren, my supervisor, for her technical and financial support throughout the course of my study at McGill. I am grateful for her understanding, patience, and kindness. She was always there to listen and to help. THANKS a lot Laurie.

I would like to thank W. A. Landi (currently at Siemens Corporate Research) for all the technical discussions that we had through email and for sending me the results of his method for comparisons. Further, I would like to thank Bhama Sridharan, one of my best friends, for her presence during all the stages of my study and for her technical support. I am thankful to V.C. Sreedhar for his constant support. He is the one who will never say no, the one who has an answer for every question. I am grateful for Ravi Shanker who was there cheer up people with his warm smile. I am thankful to Chris Donawa, our McCAT and ACAPS administrator, for his great administrative work. I would like to thank Rakesh Ghiya for his friendly attitude and all the technical discussions we had. Further, I am thankful to Clark Verbrugge who provided the table of his results for the general constant propagation. I am grateful for all the people who commented on my thesis and also who provided me with buggy test programs to test my software, especially V.C. Sreedhar. I would like to say a special thanks to all the ACPAS members who made such a friendly environment to work together, and, in particular Justiani, Michel Gagne, Chandrika Mukerji, Cecile Moura, Dhrubajyoti Goswami, Ana Erosa, and Luis Lozano. Special thanks are also due to dear Lise Minogue, the secretary of our department, for her kindness.

Further, I would like to thank Dr. Mehdi Mohageg, Dr. Mahmood Nourbakhsh, and Mr. Eskandar Ilkhan without whose support and help I could never have continued my studies. I am thankful to my dear parents, and to my dear sister, Mardjan, for their understanding. I am grateful for Amir Keyvan Khandani, one of my best friends, for being so patient and supportive.

I thank God who has surrounded me with such nice people.



Dedicated to my mother, Irandokht Yadollahi

# Contents

<b>Abstract</b>	ii
<b>Résumé</b>	iii
<b>Acknowledgments</b>	iv
<b>1 Introduction</b>	1
1.1 Alias Analysis . . . . .	2
1.2 Applications of Alias Analysis . . . . .	3
1.2.1 Constant Propagation . . . . .	3
1.2.2 Reaching Definitions . . . . .	4
1.2.3 Replacement of Aliased Variables . . . . .	4
1.2.4 Alias Information in Building the Call-graph . . . . .	5
1.3 McGill Compiler Architecture Testbed (McCAT) . . . . .	6
1.4 Thesis Contributions . . . . .	7
1.5 Thesis Organization . . . . .	9
<b>2 Background</b>	11
2.1 An Introduction to FIRST . . . . .	12
2.1.1 Motivations for Building FIRST . . . . .	12
2.1.2 Modifications to GNU C to Build FIRST . . . . .	13
2.2 SIMPLE Intermediate Representation . . . . .	13

2.2.1	Important Characteristics of SIMPLE . . . . .	13
2.2.2	Transformation of FIRST to SIMPLE . . . . .	15
2.2.3	An Overview of SIMPLE . . . . .	16
<b>3</b>	<b>Intraprocedural Points-to Analysis for Scalar Variables</b>	<b>32</b>
3.1	Motivating Example . . . . .	32
3.2	Our Approach . . . . .	33
3.3	Rules to Convert <i>Points-to</i> Information to <i>Alias</i> information . . . .	34
3.3.1	Justification of the Proposed Approach . . . . .	36
3.4	Notations and Algorithm . . . . .	37
3.5	Basic Statements . . . . .	38
3.5.1	Case 1 . . . . .	41
3.5.2	Case 2 . . . . .	41
3.5.3	Case 3 . . . . .	42
3.5.4	Case 4 . . . . .	44
3.5.5	Case 5 . . . . .	47
3.5.6	Case 6 . . . . .	47
3.6	Compositional Control Statements . . . . .	49
3.6.1	Points-to Analysis without <b>break</b> and <b>continue</b> . . . .	51
3.6.2	Points-to Analysis with <b>break</b> and <b>continue</b> . . . . .	56
3.7	Summary . . . . .	58
<b>4</b>	<b>Interprocedural Points-to Analysis for Scalar Variables</b>	<b>61</b>
4.1	Invocation Graph . . . . .	61
4.2	Interprocedural Analysis for Non-recursive Function Calls . . . . .	66
4.2.1	Relationship Between Invocation Graph and Interprocedural Analysis . . . . .	69
4.2.2	Map and Unmap Processes . . . . .	70
4.2.3	Return Statement . . . . .	93
4.3	Interprocedural Analysis for Recursive Function Calls . . . . .	96

<b>5</b>	<b>Aggregate Structures</b>	<b>102</b>
5.1	Abstract Stack Locations for Arrays and Structures . . . . .	102
5.1.1	Arrays . . . . .	102
5.1.2	Non-recursive Structures . . . . .	103
5.1.3	Recursive Structures . . . . .	101
5.1.4	Union Type . . . . .	107
5.2	Handling Complex Structure References . . . . .	107
5.3	Points-to Analysis for Aggregate Structures . . . . .	109
5.3.1	Interprocedural Points-to Analysis for Recursive Data Structures	115
<b>6</b>	<b>Handling other Pointer Features in C</b>	<b>119</b>
6.1	Pointer Arithmetic . . . . .	119
6.2	Type Casting . . . . .	121
6.3	Dynamic Data Structures . . . . .	123
<b>7</b>	<b>Implementation Details and Limitations</b>	<b>125</b>
7.1	Abstract Stack Data Structure . . . . .	125
7.2	Some Special Features of Our Analysis . . . . .	128
7.3	Some Implementation Details of Map and Unmap Processes . . . . .	130
7.4	Library Functions . . . . .	131
<b>8</b>	<b>Experimental Results and Practical Uses of Points-to Analysis</b>	<b>132</b>
8.1	Experimental Results . . . . .	132
8.2	Practical Applications of Points-to Analysis . . . . .	139
8.2.1	Replacement of Indirectly-referenced Variables . . . . .	139
8.2.2	Dependence Analysis for ALPHA . . . . .	140
8.2.3	Function Pointer Parameters . . . . .	142
8.2.4	Generalized Constant Propagation . . . . .	144
8.2.5	Practical Array Dependence Analysis . . . . .	147

8.2.6	Reaching Definition Analysis . . . . .	149
8.2.7	Live Variable Analysis . . . . .	150
8.2.8	Loop Unrolling . . . . .	150
<b>9</b>	<b>Related Work</b>	<b>152</b>
9.1	Historical Background on Alias Analysis . . . . .	152
9.2	Comparison . . . . .	155
9.2.1	Alias Representation . . . . .	157
9.2.2	Must Aliases . . . . .	160
9.2.3	Interprocedural Analysis . . . . .	161
9.2.4	Dynamic Allocation and k-limiting . . . . .	165
9.2.5	Type Casting . . . . .	166
<b>10</b>	<b>Conclusions and Future Work</b>	<b>168</b>
10.1	Summary and Conclusions . . . . .	168
10.2	Future Work . . . . .	169
<b>A</b>	<b>The SIMPLE Grammar</b>	<b>171</b>
<b>B</b>	<b>The Interprocedure Algorithms</b>	<b>175</b>
<b>C</b>	<b>Rules for the Basic Cases</b>	<b>188</b>

# List of Figures

1.1	An example of C program. . . . .	2
1.2	An example of the use of alias in building call-graph. . . . .	6
1.3	A general view of McCAT. . . . .	8
2.1	An example of pointer to array in the C-language. . . . .	15
2.2	Different cases of tree_nodes with some examples of each case. . . . .	17
2.3	An example for the representation of arrays in SIMPLE. . . . .	19
2.4	An example for the representation of structures in SIMPLE. . . . .	20
2.5	An example to show the limitation in breaking down complicated data structures. . . . .	21
2.6	SIMPLE grammar for expressions. . . . .	23
2.7	List of the 15 basic statements. Variables <b>x</b> and <b>y</b> denote varname. Variables <b>a</b> , <b>b</b> , and <b>c</b> denote val. Variables <b>p</b> and <b>q</b> denote ID. . . . .	25
2.8	The SIMPLE AST for IF_STMT. . . . .	26
2.9	The SIMPLE AST for FOR_STMT. . . . .	26
2.10	The SIMPLE AST for WHILE_STMT and DO_STMT. . . . .	28
2.11	An example of <b>switch</b> statement in SIMPLE. . . . .	28
2.12	An example for the connection between the statements. . . . .	29
2.13	An example for the connection between statements and variables. . . . .	29
2.14	An example for the representation of scope in a <b>while</b> statement. . . . .	30
2.15	An example for the connection between functions and global variables. . . . .	31
3.1	A motivating example behind our approach to intraprocedural points-to analysis. . . . .	31

3.2	An example of a definitely-points-to relationship. . . . .	35
3.3	An example of a possibly-points-to relationship. . . . .	35
3.4	General algorithm for intraprocedural analysis. . . . .	38
3.5	An example where no pointer is involved in the assignment statement. . . . .	39
3.6	Algorithm of basic statements for scalar variables. . . . .	40
3.7	An example of rule 1 in the basic statements. . . . .	42
3.8	An example of rule 2 in the basic statements. . . . .	43
3.9	A more complicated example of rule 2 in the basic statements. . . . .	44
3.10	An example of rule 3 in the basic statements. . . . .	45
3.11	An example of rule 4 in the basic statements. . . . .	46
3.12	An example of rule 5 in the basic statements. . . . .	48
3.13	An example of rule 5 that contains possibly points-to relationship in the basic statements. . . . .	49
3.14	An example of rule 6 in the basic statements. . . . .	50
3.15	Algorithm of the compositional control statements. . . . .	52
3.16	Algorithm of the <b>if</b> statement for points-to analysis. . . . .	52
3.17	Algorithm of the <b>while</b> statement for points-to analysis. . . . .	53
3.18	Algorithm of the <b>do-while</b> statement for points-to analysis. . . . .	55
3.19	The algorithm of <b>for</b> statement for points-to analysis. . . . .	55
3.20	An example of <b>if</b> statement with a <b>break</b> statement. . . . .	57
3.21	Algorithm of the <b>while</b> statement in the presence of <b>break</b> and <b>continue</b> statements. . . . .	59
3.22	Algorithm of the <b>do-while</b> statement in the presence of <b>break</b> and <b>continue</b> statements. . . . .	59
3.23	Algorithm of the <b>for</b> statement in the presence of <b>break</b> and <b>continue</b> statements. . . . .	60
3.24	Algorithm of the <b>switch</b> statement in the presence of <b>break</b> and <b>continue</b> statements. . . . .	60
4.1	An example for the construction of the invocation graph. . . . .	63

4.2	An example for the construction of the recursive invocation graph. . .	63
4.3	An example of the construction of the recursive invocation graph using our method. . . . .	64
4.4	An example for construction of the invocation graph in the presence of mutual recursion. . . . .	65
4.5	Relationship between program and invocation graph. . . . .	66
4.6	A motivating example for interprocedural analysis. . . . .	67
4.7	The memory representation without the interprocedural analysis based on a conservative assumption instead. . . . .	68
4.8	Relationship between program and invocation graph (the numbers writ- ten on the edges show the order of the process). . . . .	70
4.9	The map and unmap process. . . . .	71
4.10	A simple example of map process. . . . .	80
4.11	An example of map process where invisible variables are used. . . . .	81
4.12	An example of unmap process. . . . .	83
4.13	An example of unmap process in the presence of invisible variables. . .	85
4.14	An example of our accurate map process when an invisible variable stands for a structure. . . . .	87
4.15	An example of general map process when an invisible variable stands for more than one variable. . . . .	88
4.16	An example of our accurate map process when two invisible variables stand for more than one variable. . . . .	90
4.17	An example for the third point of the map process. . . . .	91
4.18	The algorithm for <b>return</b> statement. . . . .	94
4.19	An example of <b>return</b> statement. . . . .	95
4.20	An example of the interprocedural points-to analysis with recursive function call. . . . .	101
5.1	An example of structure representation in the abstract stack. . . . .	103
5.2	An example of nested structure representation in the abstract stack. .	104
5.3	An example of the abstract stack representation for invisible variables.	105



5.4	The incomplete abstract stack for the structure <code>foo</code> . . . . .	105
5.5	The complete abstract stack for the structure <code>foo</code> . . . . .	106
5.6	An example of a recursive data structure. . . . .	106
5.7	The overall view of points-to analysis. . . . .	110
5.8	An example of interprocedural points-to analysis in the presence of recursive data structures. . . . .	115
5.9	The points-to information of function <code>main</code> before the map process. . .	116
5.10	The points-to information of function <code>f</code> after the map process. . . . .	117
5.11	The points-to information of function <code>f</code> before the unmap process. . .	117
5.12	The points-to information of function <code>main</code> after the unmap process. .	118
6.1	An overview of the abstract heap representation. . . . .	123
6.2	An example of dynamic data allocation. . . . .	124
7.1	The data structure of the abstract stack. . . . .	127
7.2	The data structure of map information ( <code>map-info</code> ). . . . .	130
8.1	A simple example of dependence analysis. . . . .	141
8.2	An example of extending the invocation graph and points-to analysis in the presence of function pointer. . . . .	143
9.1	A general division of alias analysis. . . . .	153
9.2	An example of ICFG. . . . .	156
9.3	A simple C program. . . . .	157
9.4	A comparison example of alias representation and points-to representation. . . . .	158
9.5	An example of points-to representation. . . . .	159
9.6	An example of intraprocedural analysis for Landi and Ryder's method. .	162
9.7	An example of the usage of the map and unmap information saved in invocation graph. . . . .	165

# List of Tables

3.1	Definitions and notations used in the thesis. . . . .	37
3.2	All possible Basic Statements. . . . .	38
3.3	Basic statements to be studied in this chapter, when lhs (or rhs) is of pointer type. . . . .	39
5.1	All possible cases of basic statements which affect the points-to information when basic statement is of pointer type. . . . .	110
8.1	Characteristics of benchmark files. . . . .	131
8.2	Statistics for the number of indirect reference access in a benchmark. The first line in each case is related to cases like $*x$ and $(+x).y.z$ . The second line in each case is related to $x[i][j]$ when $x$ is a pointer to array. The third line is the sum of the first two lines. . . . .	136
8.3	General statistics of the points-to analysis. The first line in each case contains the statistics of the definitely-points-to information while the second line contains the statistics of the possibly-points-to information. . . . .	138
8.4	The statistic results for indirectly accessed variables. . . . .	140
8.5	Read and Write Sets for statement $s$ relative to point-to information $P/I$ . . . . .	141
8.6	Statistics of general constant propagation in the absence (first line) and presence (second line) of points-to analysis. . . . .	147

# Chapter 1

## Introduction

New and innovative compiling techniques as well as novel architecture design philosophies are required to exploit the ample hardware resources provided by the recent advances in VLSI technology and architecture design. To provide both high performance and cost effectiveness, it is essential that compilation techniques and architecture models are developed together, so that the effects of one on the other can be studied.

Optimizing and/or parallelizing compilers play important role in the design of high performance systems. An optimizing compiler requires detailed and accurate data flow analysis in order to perform code-improving transformations and to generate efficient code. Data-flow analysis is the process of collecting information such as the definitions and uses of variables in a program. At the core of accurate data flow analysis, lie effective dependence and alias analysis mechanisms. The ability to accurately disambiguate memory references is critical in determining precise data dependence between variables used and defined in a program.

An important thrust in architectural improvements in modern architectures involves increasing parallelism. The availability of parallel hardware is of no use without the appropriate software to exploit parallelism. In response to this requirement, great strides have been made in compiler technology to automatically detect parallelism and perform transformations to enhance parallelism and minimize dependence in the code. A major part of such dependency minimization and parallelism enhancement is based on using **alias** information.

## 1.1 Alias Analysis

An *alias* occurs at some program point when two or more names exist for the same memory location. The aliases of a particular name at a program point are all other names that refer to the same memory location at that point. *Alias analysis* is a technique that collects the information required to determine which names are aliased at each program point.

Traditionally, aliases have been represented as sets of alias pairs. Figure 1.1 demonstrates this, using some simple examples. At program point *p1* in Figure 1.1(a), *a* and *b* refers to the same memory location and therefore *\*a* and *b* are aliased. This aliasing is usually represented by the pair  $\langle *a, b \rangle$ . Figure 1.1(b) illustrates another example of aliasing where at program point *p2*, *\*a*, *\*c* and *b* are sharing the same memory location. This results in the following alias pairs:  $\{ \langle *a, b \rangle, \langle *c, b \rangle, \langle a, c \rangle \}$ .

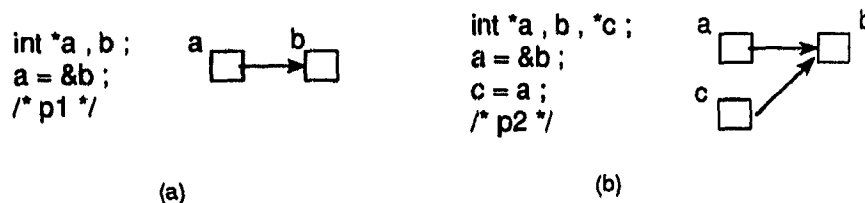


Figure 1.1: An example of C program.

The appropriate alias analysis techniques to detect the aliases that occur in Fortran, due to call-by-reference parameters have been studied and developed [Bar78, Ban79, Myc81, Coo85, CK89]. Unlike the problem of aliasing due only to call-by-reference, where aliases are **only** created by the association of actual arguments with formal parameters due to procedure call invocations, the aliasing problem in C is much more complex. In C, one must consider multi-level reference types such as `int*` and `int**`, and one must support the `&` and `*` operators that allow new aliases to be created at any point in a program (and not just at procedure call time) in addition to call-by-reference parameter passing (call by reference in C is achieved by passing the address of the parameter). The problem of alias analysis in C can be separated into the following three sub-problems:

- (1) analysis for call-by-reference parameters,
- (2) analysis for stack-allocated, non-recursive data structures, and
- (3) analysis for heap-allocated, recursive data structures.

In this thesis, we present a novel technique for precise analysis of the first two subproblems. Alias analysis for heap-allocated objects is more complicated and has been the subject of some interesting studies in the past [JM81, JM82, LH88, HN90, CWZ90, HHN92a]. The analysis presented in this thesis has been designed to complement the path matrix analysis [Hen90, HN90].

## 1.2 Applications of Alias Analysis

The presence of pointers and reference parameters makes data-flow analysis more complicated. Without accurate alias information, one needs to make the worst case conservative assumptions regarding pointers – any pointer variable can refer to any other variable in the program. Some of the typical analyses that will benefit from alias analysis are: live variable analysis, data dependence analysis, common subexpression elimination, copy propagation, array range checking, constant propagation, dead code elimination, induction-variable elimination, reaching definitions, use-definition chains, and so on.

In the following subsections, we present some examples which benefit from accurate alias analysis.

### 1.2.1 Constant Propagation

Constant propagation is a process of collecting constants at each program point, and propagating the constants along control flow paths. As an example, consider the following program:

		Constants without alias info.	Constants with alias info.
int	*a, b, c, d ;		
c = 3 ;	/* stmt 1 */	{c=3}	{c=3}
a = &c ;	/* stmt 2 */	{c=3}	{c=3}
read(d) ;	/* stmt 3 */	{c=3, d=?}	{c=3, d=?}
b = *a ;	/* stmt 4 */	{c=3, d=?, b=?}	{c=3, d=?, b=3}

At statement 1, the variable *c* gets the value of 3. At statement 2, *\*a* and *c* become aliased. At statement 3, the variable *d* gets an arbitrary value. Now let us look at statement 4 more carefully. Without the alias information, the worst case assumption

says that *\*a* can refer to any of the variables *b*, *c*, or *d*. However, in the case that we have access to the alias information, by using the fact that *\*a* is identical to *c*, we can say that *b* is equal to the constant 3. This allows the optimizer to replace *\*a* with the constant 3.

### 1.2.2 Reaching Definitions

A *definition* of a variable *x* is a statement that assigns, or may assign, a value to *x*. A definition *d* is said to *reach* a point *p* if there is a path from the point immediately following *d* to *p*. Consider the following example:

		Reaching Def. without alias info.	Reaching Def. with alias info.
int *a, b, c ;			
d1: c = 3 ;	/* stmt 1 */	{d1}	{d1}
d2: a = &c ;	/* stmt 2 */	{d1, d2}	{d1, d2}
d3: *a = 5 ;	/* stmt 3 */	{d1, d2, d3}	{d2, d3}
d4: b = c ;	/* stmt 4 */	{d1, d2, d3, d4}	{d2, d3, d4}

At statement 2, we get the alias pair  $\langle *a, c \rangle$ , and the definition of *c* at statement 1 reaches this point. If we do not have the alias information at statement 1, we should assume that the reaching definition of *c* can be either statement 1 or statement 3 (because *\*a* at statement 3 could be either *c* or *d*). But, if we know that *\*a* is aliased to *c*, we can definitely say that only statement 3 reaches statement 4. Furthermore, using Constant Propagation at statement 4, we obtain: *b* = 5.

### 1.2.3 Replacement of Aliased Variables

If two variables are aliased through all of the execution paths of the program, they can be replaced by one another. This replacement is a useful tool in most of the analyses. In the following, we will briefly explain the effect of such replacement on **live-variable analysis**.

In live-variable analysis, we want to know for a given variable *x* and program point *p*, whether the value of *x* could be used in some execution path starting at *p*. One important use of live-variable analysis is in register allocation. The live-variable information allows the register allocator to efficiently use the registers for storing the variables. Consider the following example:

<code>int *a, b, c, d ;</code>		<b>Live-variables</b>
		<b>without alias replacement</b>
<code>b = 2;</code>	<code>/* stmt 1 */</code>	<code>{}</code>
<code>d = 1 ;</code>	<code>/* stmt 2 */</code>	<code>{b}</code>
<code>c = 4 ;</code>	<code>/* stmt 3 */</code>	<code>{d, b}</code>
<code>printf ("%d", d, b);</code>	<code>/* stmt 4 */</code>	<code>{d, c, b}</code>
<code>a = &amp;c ;</code>	<code>/* stmt 5 */</code>	<code>{d, c, b}</code>
<code>printf ("%d ", *a);</code>	<code>/* stmt 6 */</code>	<code>{}</code>

Live-variable is a backward analysis in the sense that the analysis starts from the last statement of the program and continued to the first statement. Due to this fact, we start from statement 6 of the given example. Without having the alias information, the worst case assumption at statement 6 says that `*a` can be any of the variables `b`, `c`, and `d`. Consequently, at statement 5, all of these variables are live and only `a` gets killed. The same set of variables are live at statement 3 while variable `c` gets killed. Considering these facts, variables `b` and `d` are live after statement 3. Now let us consider the same example after the replacement of the alias information. This information results in the replacement of `*a` by `c` in statement 6. The same program after this replacement is shown below:

<code>int *a, b, c, d ;</code>		<b>Live-variables</b>
		<b>with alias replacement</b>
<code>b = 2;</code>	<code>/* stmt 1 */</code>	<code>{}</code>
<code>d = 1 ;</code>	<code>/* stmt 2 */</code>	<code>{b}</code>
<code>c = 4 ;</code>	<code>/* stmt 3 */</code>	<code>{d, b}</code>
<code>printf ("%d", d, b);</code>	<code>/* stmt 4 */</code>	<code>{d, c, b}</code>
<code>a = &amp;c ;</code>	<code>/* stmt 5 */</code>	<code>{c}</code>
<code>printf ("%d ", c);</code>	<code>/* stmt 6 */</code>	<code>{}</code>

Since `c` is the unique variable used at statement 6, `c` is the only live variable at statement 5. Thus, using alias replacements, the live ranges of variables `d` and `b` are considerably reduced.

#### 1.2.4 Alias Information in Building the Call-graph

The alias information has a special use for function pointers. Instead of a direct call to a function, the address of the function can be assigned to a variable of type function pointer which will be called later. Without having the alias information for

function pointer variables, it is impossible to know which function is being called at a particular call-site. Due to this fact, the construction of the complete call-graph is not accurate in presence of function pointers. In order to complete the call-graph, the worst case assumption has to be taken which in this case says that each call to a function pointer is equivalent to calling all the possible functions in the program. This leads to a very inaccurate result, whereas by using alias analysis a far more precise result will be obtained.

```
f() {
...
}
g() {
...
}
h() {
...
}
main() {
int (*func_ptr)();

func_ptr = f; /* stmt 1 */
func_ptr(); /* stmt 2 */
}
```

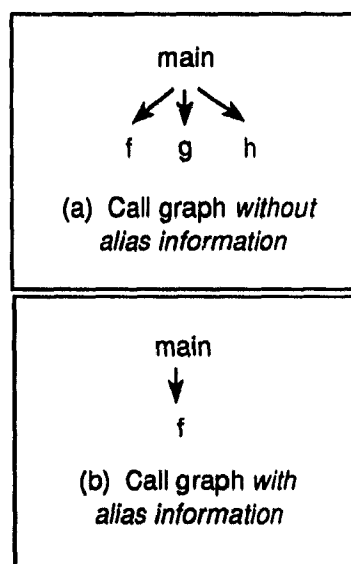


Figure 1.2: An example of the use of alias in building call-graph.

Consider the example shown in Figure 1.2. At statement 1, `*func_ptr` and `f` become aliased. At statement 2, without the alias information, one must assume that the call to `func_ptr` can be a call to any of the functions `f`, `g`, and `h`. However, with the alias information in hand, we can restrict this to the set of all functions aliased to the function pointer (in the given example, only `f` is aliased to the function pointer `func_ptr`).

### 1.3 McGill Compiler Architecture Testbed (McCAT)

In order to study quantitatively the effect of various compilation techniques on sophisticated architectures, it is necessary to develop a complete compiler-architecture testbed. Our alias analysis method has been developed hand-in-hand with the development of the McCAT [Sri92, HDE<sup>+</sup>92] compiler. Figure 1.3 gives a general view of



McCAT system.

The first component of such a testbed is a compiler which supports both high level and low-level compiler transformations which translate high-level programs to low level programs suitable for a variety of architecture simulators and real machines. For example a precise alias analysis is more effective at the higher-level representation, while instruction scheduling needs to be done on a lower-level representation. On the other hand, the information collected in the high-level intermediate representation should be passed to the lower level, to be used by low-level analysis. The second component consists of architecture simulation tools that process the output of the compiler to produce a variety of performance results. McCAT was designed and developed with the consideration of the above objectives.

The McCAT compiler compiles C programs. C is chosen as the source language, because of: (i) its wide range of usage, (ii) having a variety of programming languages features, and (iii) being powerful because of the flexibility of its usage.

To avoid the redundant work of building an efficient front-end, the front end of GNU-C compiler has been used and modified. The source code of GNU-C compiler is freely available.

The McCAT compiler first translates the C program into FIRST which is a high level representation of the program. Next, a series of tree transformations are performed on FIRST to create SIMPLE which is a simplified Abstract Syntax Tree (AST). SIMPLE forms the intermediate tree representation for analyses and high level optimizing transformations.

A more detailed description of McCAT will be presented in Chapter 2. The readers are referred to [Sri92, HDE<sup>+</sup>92] for a complete discussion.

## 1.4 Thesis Contributions

This thesis reports on a new, general, and accurate alias processing technique that estimates the alias information at each point of a given program. This analysis is performed on the SIMPLE representation of C programs.

Our approach differs from other approaches in that we do not compute the alias pairs explicitly. We have chosen an approximation that computes the relationships between abstract stack locations. The central idea is that since we are dealing with data-structures that are stack-allocated, we can model the real stack with an abstraction of the stack. Using these abstract stack location, we estimate which stack

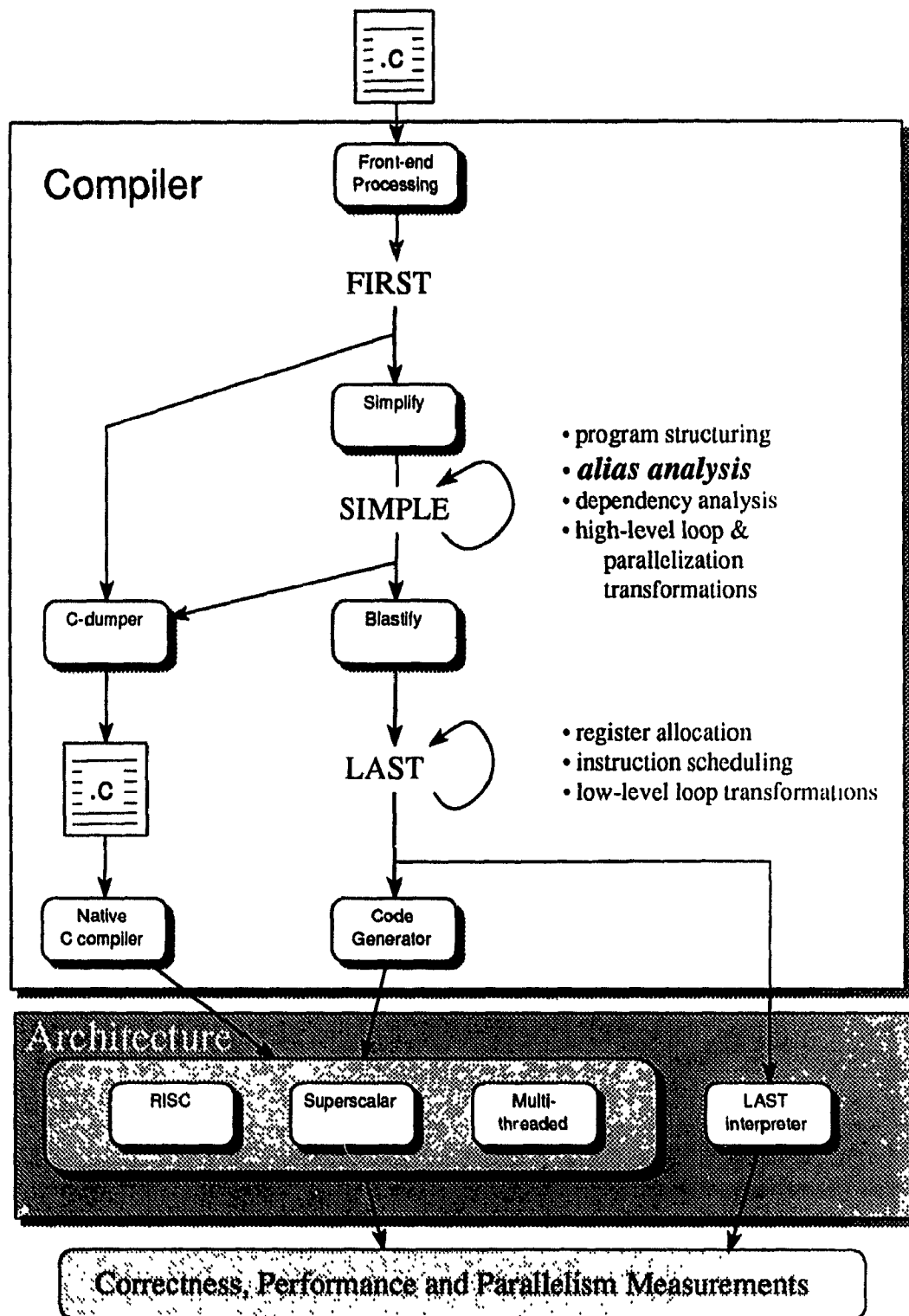


Figure 1.3: A general view of McCAT.

locations point-to other stack locations. Thus, we call our analysis, points to analysis rather than alias analysis. This is a direct and simple approach that gives accurate results with which we can perform a straight-forward dependence analysis based on the *reads* and *writes* to the abstract stack locations. Furthermore, this information about relationships is sufficient to compute the conventional alias information

In summary, the main contributions of this thesis are:

- Design of an accurate interprocedural points-to analysis technique for stack allocated data-structures (including records and arrays) which forms the basis for the design criteria of the intermediate representation SIMPLE.
- Development and implementation of the analysis on the compositional control statements, like **if**, **while**, **switch**, **for**, **repeat**, **break**, and **continue**
- Design and implementation of a general framework to perform interprocedural points-to analysis in the presence of both recursive and non-recursive procedure calls.
- Presentation of experimental results indicating the effectiveness of the analysis
- Discussion of how the points-to analysis has been used to improve many other analyses in the McCAT compiler.

It should be mentioned that since we are focusing on structured and structurable programs, any program using **gotos** in an unrestricted manner must be first converted into an equivalent structured program [Ero93].

## 1.5 Thesis Organization

Chapter 2 discusses the required background and the McCAT testbed. Chapter 3 provides detailed information about the intraprocedural points-to analysis on scalar variables. Chapter 4 is focused on the design of invocation graph and the details of interprocedural points-to analysis (including recursive function calls). Chapter 5 discusses aggregate structures (record and array types). Chapter 6 discusses other features of the C-language like dynamic data allocation and type casting. Chapter 7 is devoted to some implementation details. Chapter 8 is concerned with the practical aspects of our analysis where we give numerical results using standard benchmarks and explain other analyses which are using our result. Chapter 9 presents a review of related work. Finally, Chapter 10 contains some concluding remarks and directions for the future work.

In most parts of the thesis, we assume that the reader has some understanding of the C-language.

## Chapter 2

# Background

In this chapter, we provide the relevant background material required for the understanding of the following chapters. More specifically, we focus on the SIMPLE intermediate representation. SIMPLE is designed to effectively support various analyses and transformations, in particular alias analysis. This chapter is a summary of the work done by Bhama Sridharan. Readers are referred to [Sri92, HS92, HDE<sup>+</sup>92] for a complete description.

In order to separate the front-end processing from other analyses and optimizations, the McCAT compiler is designed to support a family of intermediate representation, namely: FIRST, SIMPLE, and LAST (refer to Figure 1.3). These three representations can be summarized as follows:

1. **FIRST**: a high-level Abstract Syntax Tree (AST) representation of the original program. FIRST retains program and data declarations as written by the programmer.
2. **SIMPLE**: an intermediate AST representation based on FIRST. In SIMPLE, complex program statements and expressions are presented in a simplified form. SIMPLE is designed for high-level compiler transformations, analyses and optimizations.
3. **LAST**: a low-level AST representation. LAST is designed for register allocation, instruction scheduling, code generation and low-level transformations [Don93].

Since the analysis reported in this thesis is performed at the SIMPLE level, we focus on the SIMPLE representation in this chapter. We first give an introduction to the FIRST AST in Section 2.1 and then we explain the SIMPLE AST in Section 2.2.

## 2.1 An Introduction to FIRST

FIRST is built to separate the front-end processing (e.g. scanning, parsing, and type checking) from high-level analysis, optimization and code generation.

The intermediate representation of the original GNU C compiler [Sta90] is based on the Register Transfer Language (RTL). In the GNU C compiler, the RTL representation is produced one statement at a time. For each statement, the parser builds an abstract syntax tree, converts it to RTL, and then frees the corresponding abstract syntax tree. In this manner, the RTL intermediate code for an entire function is generated. After optimization of the RTL, assembly code for a program is generated, one function at a time. For several reasons which are explained in the next section, FIRST, SIMPLE, and LAST have been designed to replace RTL in the GNU C compiler.

In Section 2.1.1, the motivation for building FIRST is explained. In Section 2.1.2, the modifications that have been done to the GNU C compiler in order to build FIRST are given.

### 2.1.1 Motivations for Building FIRST

- As mentioned before, the GNU C compiler generates the RTL for an entire function. Then, several intraprocedural optimizations are performed on RTL to produce the target code for that function. After generating code for a function or a top-level declaration, all storage used by the function definition is freed unless the function is "inlined". As a result, interprocedural optimization can not be performed by the original GNU C. Furthermore, as the intermediate code for the entire program is unavailable, high-level interprocedural analysis and detailed alias analysis techniques can not be applied.
- High-level compiler optimizations such as loop transformations and array optimizations are extremely difficult to perform on the RTL code. This is due to the fact that the identity of loop structures and array references are completely lost at this low level. For example, it is difficult to perform the array dependence analysis because the array references are broken down to lower-level statements. Similarly, as loops are transformed into blocks with goto's and labels, high-level loop transformations are difficult to perform. Retaining the identity of loop structures and array references enables us to perform a number of high-level loop transformations which are otherwise far more difficult to perform.

### 2.1.2 Modifications to GNU C to Build FIRST

In the original GNU C compiler the syntax tree is created up to the expression level. Once the statement has been parsed, the tree nodes representing these expressions are freed. Since we are interested in building the syntax tree for the entire program, the parser was modified to retain the tree nodes. The parser is further modified to continue building the AST for the complete program. The parser now constructs the FIRST tree for the entire program.

In RTL, different kinds of loops are not differentiated from each other. New nodes to construct different kinds of loop structures, such as: while-loop, for-loop, and do-loop constructs, are added to FIRST.

An additional field, the TREE\_INFO field, has been incorporated in the basic tree\_node structure to store the data-flow analysis information.

## 2.2 SIMPLE Intermediate Representation

The SIMPLE intermediate representation was specifically designed to handle high level data-flow analyses and transformations, and in particular alias analysis.

In the rest of this section, we give the important characteristics of SIMPLE (Section 2.2.1), the transformation from FIRST to SIMPLE (Section 2.2.2), and a general overview of the SIMPLE AST (Section 2.2.3).

### 2.2.1 Important Characteristics of SIMPLE

**Compositional Representation:** The SIMPLE representation is a compositional representation of the program where the control flow is explicit. For example, it is possible to analyze a while loop by analyzing only its components, namely, the conditional expression and the body. This gives the opportunity for alias analysis to take a compositional approach in the implementation. This kind of compositional representation has three advantages: (i) the flow of control is structured and is explicit in the program representation, (ii) structured analyses tools supporting such techniques can be used to analyze all the control-flow constructs, and (iii) it is simple to find and transform groups of nested loops. It should be noted that in addition to ordinary compositional constructs such as conditionals and loops, the compositional approach directly support the commonly used **break** and **continue** statements for loops, and the **return** statement

for procedures and functions. However, unrestricted use of `goto` is not compositional and cannot be supported directly. Any program with unstructured control flow must be converted into an equivalent program with structured control flow [WO75, Bak77, Amm92]. The McCAT compiler overcomes the problem associated with `goto` statements by transforming unstructured programs to structured program through a process called *restructuring* [Ero93].

**Explicit Array and Structure References:** The identity of array and structure references is retained, i.e., the array and structure references are not broken down into a series of lower-level statements that perform address computations. This is required to make full use of high-level information such as array dimension, array size, pointer types, and recursive structure types. Without this information, alias analysis could not collect the alias information related to arrays and structures. This information is essential in array dependence analysis [Jus93].

**Types and Typecasting:** The exact type information and type casting is also retained. Often alias analysis takes advantage of type information to provide more accurate results. For example, if there are no type casts available, it can be inferred that a variable of one type cannot be aliased to a variable of another type using type information. A more advanced example is the use of recursive types for dynamically-allocated pointer structures [HN89, Deu92].

**Pervasive Data-flow Information:** It is possible to transmit important data-flow information collected at a higher-level intermediate representation to a lower-level, and thus improve the effectiveness of the low-level transformations. For example, alias analysis information collected at a high-level can be used to perform better dependence analysis, and therefore better instruction scheduling, at a lower-level.

**Simple to Analyze:** The intermediate representation is simple enough so that it could be analyzed in a straightforward manner. To simplify accurate analysis of such a program, one needs to break down complex structures and statements into simpler cases. In SIMPLE, there are a small number of basic statements all at a level at which structured analyses rules can be easily specified, i.e. at a level most suitable to perform alias analysis. Further, one is able to represent any complicated C statement or expression as a sequence of these basic statements. Similarly, the conditional parts of while-loops, for-loops, do-loops and if-statements are simple expressions. Any complicated expression is simplified when represented in the intermediate form. This results in a reduced number of possible cases and allows the alias analysis to define a general rule for each case. Furthermore, in C, side-effects can occur in many places where one expects an



expression. In this simpler form, statements that can have side-effects are clearly separated. This assure the generality of rules that we provide for alias analysis.

**Clear Semantics:** SIMPLE has clear and obvious semantics. One part of this process is clarifying some of the implicit meanings in C programs. Consider the example shown in Figure 2.1. The statement `b = a` means "assign the address

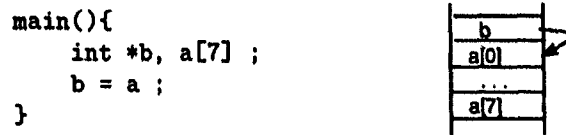


Figure 2.1: An example of pointer to array in the C-language.

of the first item of `a` to `b`", and not "assign the value of `a` to `b`" as one would expect for a scalar assignment. These two interpretations of the same statement can make a difference in alias analysis. For example, in the first case `*a` would be alias to `b`, while in the second case, the alias information would not change. These implicit semantic rules in C are made explicit in SIMPLE.

**Interprocedural Analysis:** SIMPLE retains all the information about the complete program, so that interprocedural data-flow and alias analysis can be performed. This is particularly important when we have non-scientific code composed of many small and possibly recursive procedures.

### 2.2.2 Transformation of FIRST to SIMPLE

The following are some of the transformations that have been done to transform FIRST to SIMPLE. The detailed explanation is given in Section 2.2.3.

- Complex variable names are split up whenever it is possible.
- Loops, switches and conditionals are transformed into SIMPLE format.
- Assignment statements and expression statements are broken down into 3-address code format.

A complete grammar for SIMPLE is given in Appendix A. In the next section, a brief overview of SIMPLE is given.

### 2.2.3 An Overview of SIMPLE

SIMPLE is basically a simplified form of FIRST. It is built of a `tree_node` structure. `Tree_nodes` are varied according to the data type and expression. They are built of the following two major parts:

1. The first part is common to all the `tree_nodes`. Following is the list of the corresponding major fields:
  - `Tree_uid` : contains a unique identification integer for the corresponding `tree_node`.
  - `Tree_type`: points to the type of the `tree_node`.
  - `Tree_code`: contains an enumerated type integer which gives the name of the `tree_node`.
  - `Tree_info`: points to an array of pointers which holds data-flow information.
2. The second part of `tree_nodes` is dependent on the corresponding `tree_code`. For example, a node with `tree_code` `MODIFY_EXPR` (modify expression) has two fields to hold the left and right hand sides of an assignment, while `IF_STMT` (if statement) has three nodes to hold condition expression, then-body, and else-body.

`Tree_nodes` are classified into seven basic types. Different cases together with some examples for each case are shown in Figure 2.2. In the rest of this chapter, we explain these different basic cases.

#### **Constant Nodes**

These nodes represent constants in C-language. They have the common suffix `_CST`. Following are the examples given in Figure 2.2:

- `INTEGER_CST`: its value can be any integer constant (e.g. 7).
- `REAL_CST`: its value can be any float number (e.g. 5.7).
- `STRING_CST`: its value can be any string constant (e.g. "abc").

#### **Type Nodes**

As mentioned before, `tree_type` is one of the fields in the common part of each node. This field points to a node of type "type node" which contains the type of the corresponding node. Type nodes have the common suffix `_TYPE`. Some of the different kinds of type nodes are listed in the following:

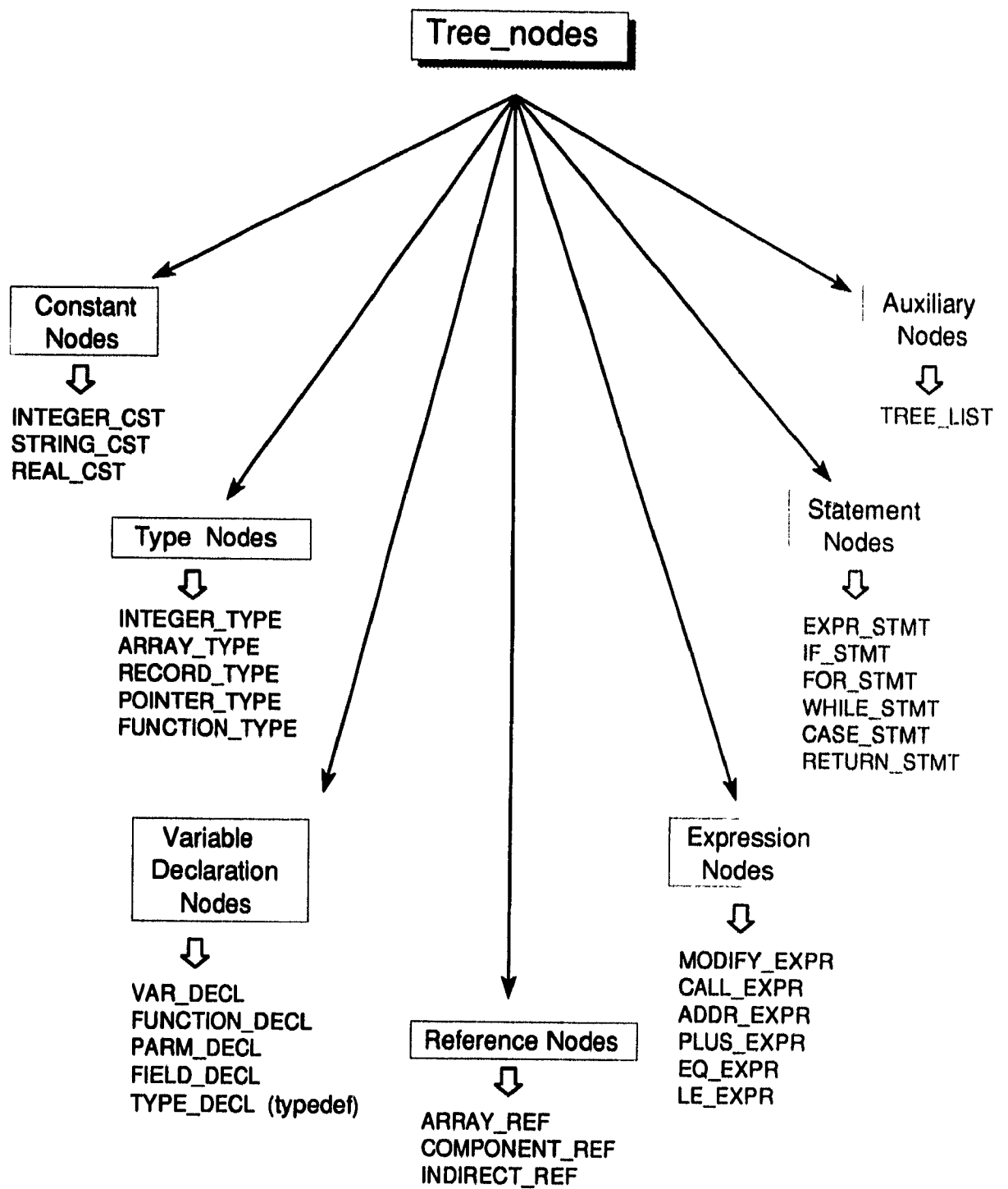


Figure 2.2: Different cases of tree\_nodes with some examples of each case.

- **INTEGER\_TYPE**: used for integer basic type. **VOID\_TYPE**, **REAL\_TYPE**, and **CHAR\_TYPE** are other examples of basic type.
- **ARRAY\_TYPE**: used for specifying the array type. Among other fields, this node contains the dimension of the array. The **tree\_type** field of this node points to the type of the array which can be any valid type.
- **RECORD\_TYPE**: used as the type of a structure variable. It has a pointer to the fields of the corresponding structure. Each of the fields is built of a **FIELD\_DECL** node. This will be explained shortly.
- **POINTER\_TYPE**: used for the pointer type. The **tree\_type** field of this node is another "type node". For example, if variable **a** is of type **int\***, the type of **a** would be **POINTER\_TYPE** of type **INTEGER\_TYPE**.
- **FUNCTION\_TYPE**: used for functions. The **tree\_type** field of this node is another "type node" which contains the type of the corresponding function. For example, if a function is of type **void**, the type of the related **FUNCTION\_TYPE** would be **VOID\_TYPE**.

### **Variable Declaration Nodes**

These types of nodes which have the common suffix **\_DECL**, represent variables declaration. All of these nodes have a field "name" which contains the name of the corresponding variable. These nodes also contain some other information which are beyond our discussion. The **tree\_type** field (a field of the common part) contains the type of corresponding variable. Following are some examples of the declaration nodes:

- **VAR\_DECL**: used for any global or local variable.
- **FUNCTION\_DECL**: used for the function declaration. One of its fields is a pointer to the list of formal parameters.
- **PARAM\_DECL**: used for formal parameters of a function declaration.
- **FIELD\_DECL**: used for the fields of a structure.
- **TYPE\_DECL**: used to represent a type defined by the user, using the **typedef** command.

### **Reference Nodes**

The more common cases of these nodes are listed in the following. They have the common suffix **\_REF**. Readers are referred to the subsequent part for some examples.

- **ARRAY\_REF**: used for representing arrays (e.g. **a[i][5]**).

- COMPONENT\_REF: used for representing structures (e.g. a.b.c).
- INDIRECT\_REF: used for representing the indirect access to a variable (e.g. \*a).

Variables are either scalar or aggregate (array and structure). Since scalar variables are straight forward (they are of either VAR\_DECL or PARM\_DECL tree node type), we explain the aggregate variables. We first show how variable references are represented and then we show how the complicated cases are broken down into simpler ones.

The example presented in Figure 2.3 is devised to illustrate array references in SIMPLE. Assignment statement 1 (S1) is equivalent to  $t = \&a[0]$ . This statement

```
main(){
    int    a[10], *t, p, q ;
    char *str ;
    int    b[5][7] ;

    t = a ;      /* S1 */
    p = t[2] ;   /* S2 */
    q = a[1] ;   /* S3 */
    str = "abc" ; /* S4 */
    b[2][i] = 6 ; /* S5 */
}
```

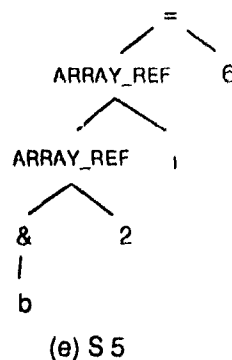
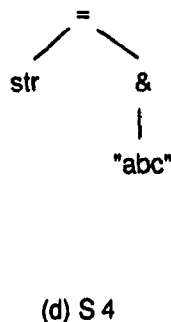
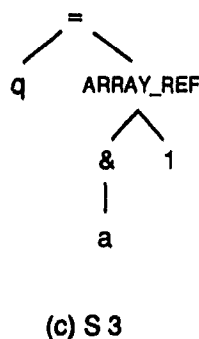
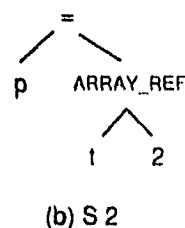
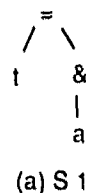


Figure 2.3: An example for the representation of arrays in SIMPLE.

implicitly assigns the address of `a` to `t`. Since one of the goals in the design of SIMPLE is to have clear semantics, the address operator is made explicit in the SIMPLE AST.

This is represented in Figure 2.3(a). As it is shown in statements 2 and 3, the syntax of accessing an array through a pointer to an array ( $t[2]$ ) and the actual array ( $a[1]$ ) look exactly the same, but, semantically they are different. The address of  $t[2]$  is computed by increasing the contents of  $t$  by 2, while the address of  $a[1]$  is computed by increasing the address of  $a$  by 1. SIMPLE clarifies this difference by adding an address operator before the array reference. This is shown in Figures 2.3(b) and (c). At statement 4, a string constant "abc" is assigned to variable  $str$ . Since a string is actually treated as an array, it will also have an address operator as shown in Figure 2.3(d). Finally, Figure 2.3(e) shows the representation of a two-dimensional array.

In SIMPLE, different fields of a structure are connected through a COMPONENT\_REF node. An example of the SIMPLE tree for variable  $d.c.a$  is shown in Figure 2.4.

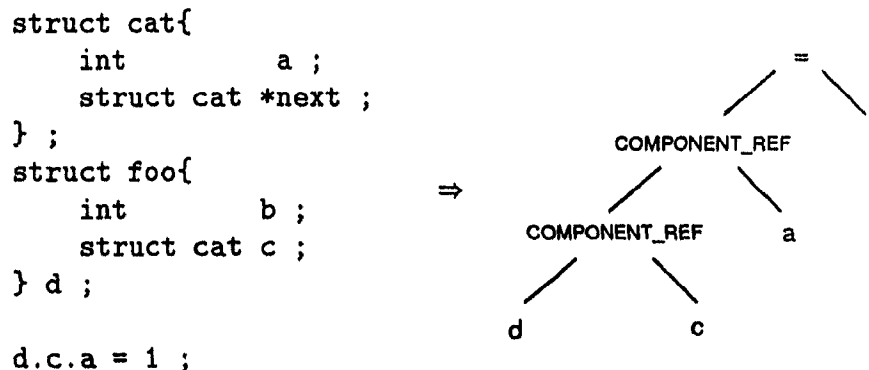


Figure 2.4: An example for the representation of structures in SIMPLE.

Now that the representation of arrays and structures is clarified, we show how the complicated data structures, which are a combination of arrays and structures (e.g.  $a.b[4].c$ ), are broken down into simpler cases. This is done as long as the meaning of the variable reference is not lost. For example:  $a.b.c$ ,  $a$ ,  $a[5][i]$ ,  $(\&a).b$ ,  $(\&a).b.c$  are some basic cases which can not be broken down.

The example in Figure 2.5 is devised to show why cases like  $(\&a).b$  can not be further broken down. The replacement of statement 1 by statement 2 and 3 does not give the same result. At statement 1, variable  $(\&a).b$  modifies the location  $x.b$  while at statement 3,  $temp0.b$  is the variable which is modified. Since these two cases are not identical,  $(\&a).b$  or  $(\&a).b.c$  can not be broken down.

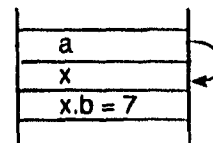
```

struct foo{
    int b ;
} x, *a ;

main()
{
    x.b = 5 ;
    a = &x ;
    (*a).b = 7 ; /* S1 */
}

```

(a) The original C program.



(b) The stack representation of the original program.

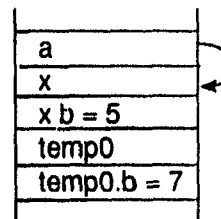
```

struct foo{
    int b ;
} x, *a ;

main()
{
    x.b = 5 ;
    a = &x ;
    {
        struct foo temp0 ;
        temp0 = *a ; /* S2 */
        temp0.b = 7 ; /* S3 */
    }
}

```

(c) The modified C program.



(d) The stack representation of the modified program

Figure 2.5: An example to show the limitation in breaking down complicated data structures.

In the following, we represent some simple examples of valid conversions in SIMPLE (the left sample is a C language statement and the right one is the same statement as represented in SIMPLE).

Example 1:

<code>x = a[5].b.c</code>	$\Rightarrow$	<code>temp0 = &amp;a[5]</code>
		<code>x = (*temp0).b.c</code>

Example 2:

<code>x = a-&gt;b-&gt;c</code>	$\Rightarrow$	<code>temp0 = (*a).b</code>
		<code>x = (*temp0).c</code>

Example 3:

<code>x = a.b[i]</code>	$\Rightarrow$	<code>temp0 = &amp;a.b</code>
		<code>x = temp0[5]</code>

Example 4:

		<code>temp0 = &amp;a.b</code>
		<code>temp1 = &amp;temp0[i]</code>
<code>x = a.b[i].c.d[2][j].e</code>	$\Rightarrow$	<code>temp2 = &amp;(*temp1).c.d</code>
		<code>temp3 = &amp;temp2[2][j]</code>
		<code>x = (*temp3).e</code>

### Expression Nodes

Referring to the SIMPLE grammar shown in Figure 2.6, an expression is built of `rhs` when `rhs` is either `unary_expr` or `binary_expr`. We first consider `unary_expr` and then `binary_expr`. Among `unary_expr`, we are particularly interested in the `*`, `&` operators, and `call_expr` because of their effect in alias analysis. Some examples are given in the following to clarify the grammar for `unary_expr`.

Example 5 (on `*` operator):

<code>a = **b</code>	$\Rightarrow$	<code>temp0 = *b</code>
		<code>a = *temp0</code>

Example 6 (on `*` operator):

<code>a = *(b.c)</code>	$\Rightarrow$	<code>temp0 = b.c</code>
		<code>a = *temp0</code>

The address operator is also one of the cases that can not always be simplified (because the meaning of the expression might change). Therefore, a complex variable name may appear after '`&`' operator. As an example, `&(a.b)` or `&(*a.b)` must remain unchanged.

Example 7 (on `&` operator):

<code>a = &amp;(b-&gt;c-&gt;d)</code>	$\Rightarrow$	<code>temp0 = (*b).c</code>
		<code>a = &amp;((*temp0).d)</code>



<pre> expr : modify_expr         rhs  modify_expr : varname '=' rhs                '*' ID '=' rhs  rhs : unary_expr        binary_expr  unary_expr : simp_expr               '*' ID               '&amp;' varname               call_expr               unop val               '(' cast ')' varname /* 'cast' stands for valid C typecasts */  simp_expr : varname   CONST  call_expr : ID '(' arglist ')' </pre>	<pre> arglist : arglist ',' val   val  unop : '+'   '-'   '!'   '~'  binary_expr : val binop val  binop : relop           '-'   '+'   '/'   '*'           '%'   '&amp;'   ' '   '&lt;&lt;'           '&gt;&gt;'   '^'  relop : '&lt;'   '&lt;='   '&gt;'   '&gt;='           '=='   '!='  val : ID   CONST  varname : arrayref /*ARRAY_REF*/            compref /*COMPONENT_REF*/            ID </pre>
---	--

Figure 2.6: SIMPLE grammar for expressions.

The actual parameters of a function call (call expression) are simplified to simple variables (ID) or constants. Some examples of CALL\_EXPR are given in the following

Example 8 (on function call):

<pre>f( 3, &amp;a, *b)</pre>	$\Rightarrow$	<pre>temp0 = &amp;a temp1 = *b f( 3, temp0, temp1)</pre>
------------------------------	---------------	--

Example 9 (on function call):

<pre>f( a.b, c[7], "abc")</pre>	$\Rightarrow$	<pre>temp0 = a.b temp1 = c[7] temp2 = "abc" f( temp0, temp1, temp2)</pre>
---------------------------------	---------------	---

Note that since string constant "abc" is considered as an array, it is replaced by a temporary variable.

As it is shown in the grammar represented in Figure 2.6, the binary expressions (binary\_expr) are in 3-address code format. This is shown in the following example

Example 10 (on 3-address code format):

$a = (*b) + c / d$	$\Rightarrow$	$temp0 = *b$
		$temp1 = c/d$
		$a = temp0 + temp1$

One of the objectives of SIMPLE is to produce a clear definition for each expression or statement. Therefore, SIMPLE transforms the conditional, compound and logical expressions into a series of simpler statements. Following are some examples to show this transformation.

Example 11:

$(a > b) ? a : b = c$	$\Rightarrow$	$if ( a > b )$
		$a = c$
		$else$
		$b = c$

Example 12:

$a = b = c$	$\Rightarrow$	$b = c$
		$a = b$

Example 13:

$a = b \ \&\& \ c + d$	$\Rightarrow$	$temp0 = (b \neq 0);$
		$if (temp0)$
		$temp0 = (c \neq 0);$
		$a = temp0 + d ;$

### Statement Nodes

In the following, we study different types of statements in SIMPLE. These are basically divided into two major groups:

- **Basic statements:** are the expressions followed by ';' (e.g. modify expressions and function calls).
- **Compositional control statement:** are all the conditional statements (e.g. `if` statement), and loop statements (e.g. `while` statement).

### Basic Statements:

Different types of basic statements in a C-program can be replaced by one or more of the 15 basic statements given in Figure 2.7. The node related to basic statements in SIMPLE is `EXPR_STMT`.

1. `x = a binop b`    where `binop` is any binary operation
2. `*p = a binop b`
3. `x = unop a`    where `unop` is any unary operation
4. `*p = unop a`
5. `x = y`
6. `*p = y`
7. `x = f(args)`    where `args` is a possibly empty list of arguments
8. `*p = f(args)`
9. `x = (cast)b`    where `cast` is any typecast
10. `*p = (cast)b`
11. `x = &y`
12. `*p = &y`
13. `x = *q`
14. `*p = *q`
15. `f(args)`

Figure 2.7: List of the 15 basic statements. Variables `x` and `y` denote varname. Variables `a`, `b`, and `c` denote val. Variables `p` and `q` denote ID.

### Compositional Control Statements:

In SIMPLE, compositional control statements consist of: the `if` statement (`IF_STMT`), `for` loop (`FOR_STMT`), `while` loop (`WHILE_STMT`), `do` loop (`DO_STMT`), `switch` (`SWITCH_STMT`), `case` (`CASE_STMT`), `default` (`DEFAULT_STMT`), `break` (`BREAK_STMT`), `continue` (`CONTINUE_STMT`), and `return` (`RETURN_STMT`) statements. The flexibility of C-language allows the programmers to write the compositional control statements in a complicated manner. SIMPLE changes their syntax to a restricted format. In the following, we explain each of the above cases:

**if statement:** The grammar of `if` statement is as follows:

```
stmt:   IF '(' condexpr ')' stmt
        | IF '(' condexpr ')' stmt ELSE stmt
```

The graph representation of `if` statement in SIMPLE AST is shown in Figure 2.8 (the else-body will be NULL if the else part does not exist). Clearly the 'condexpr' is already being simplified. Following is an example of `if` statement:

Example 14:

```
if ( a = b = c + d, d > a )
{ ... }

      ⇒      b = c + d
              a = b
              if ( d > a )
              { ... }
```

**for loop statement:** The grammar of a `for` loop is as follows:

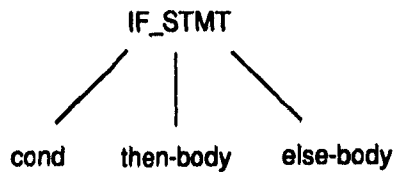


Figure 2.8: The SIMPLE AST for IF\_STMT.

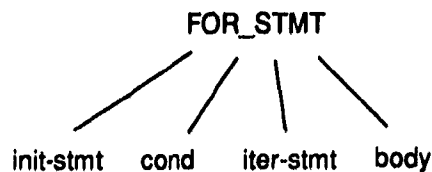


Figure 2.9: The SIMPLE AST for FOR\_STMT.

FOR '(' expr ';' condexpr ';' expr ') stmt

The graph representation of **for** statement in SIMPLE AST is shown in Figure 2.9. Only one initial statement and iteration statement is allowed. If more than one is used, it is moved outside of the loop. This is shown in the following example:

Example 15:

<pre> for (i=1,j=1 ; j&lt;10 ; i++,j++) {     stmts ; } </pre>	$\Rightarrow$	<pre> i = 1 for (j=1 ; j&lt;10 ; j=j+1) {     stmts ;     i = i + 1 ; } </pre>
--	---------------	--

If there is more than one iteration statement, in order to get the correct answer, the iteration statement(s) (in this case  $i=i+1$ ) should be moved to the end of the for loop.

The conditional expression should be also in 3-address code format. Following example is devised to show this fact.

Example 16:

<pre> for (i=1,j=1 ; i+j&lt;k ; i++,j++) {     stmts ; } </pre>	$\Rightarrow$	<pre> i = 1 j = 1 for (temp0=i+j ; temp0&lt;k ;     temp0=i+j) {     stmts ;     i = i + 1 ;     j = j + 1 ; } </pre>
---	---------------	---

For all loop constructs, the **continue** statement indicates that flow of control moves to the beginning of the loop. Therefore, the newly added statement(s) to the end of the loop will not be executed. To solve this problem, the newly added statement(s) should be also added before each **continue** statement. An example of this case is given in the following:

Example 17:

<pre> for (i=1 ; i&lt;10 ; j++,i++) {     if (cond)         continue     stmts ; } </pre>	$\Rightarrow$	<pre> for (i=1 ; i&lt;10 ; i=i+1) {     if (cond)     {         j = j + 1 ;         continue     }     stmts ;     j = j + 1 ; } </pre>
---	---------------	---

The **break** statement moves the flow of control to the end of the loop. In this case, the iteration statement(s) is not executed and, therefore, nothing should be done for the **break** statement.

**while and do-while statements:** The grammar of **while** and **do** loop statements is as follows:

```

stmt: WHILE '(' condexpr ')' stmt
stmt: DO stmt WHILE '(' condexpr ')'

```

The graph representation of **while** and **do-while** statements are shown in Figure 2.10. As **while** and **do-while** statements are similar to **for** statement, we do not give any further examples for them.



Figure 2.10: The SIMPLE AST for WHILE\_STMT and DO\_STMT.

**switch and case statements:** These statements can be more complicated than other ones. This happens because the body of different **case** statements can be partially shared. This results in the following two problems: (i) the statement sequence may be labeled with more than one case label, (ii) the flow of control into and out of the **switch** statement may not be regular (for example, one may have a case label in the middle of statement sequence). In SIMPLE, each statement sequence begins with one more case labels and ends with either a **break**, **continue**, or **return** statement. Readers are referred to Appendix A for the corresponding grammar. An example of simplifying a **switch** statement is given in example 18 of Figure 2.11.

Example 18:

<pre> switch(a) {     case 1:     case 2:         stmt1 ;     case 3:         stmt2 ;     default:         stmt3 ;         break ;     case 4:         stmt4 ; } </pre>	<p>⇒</p>	<pre> switch(a) {     case 1:     case 2:         stmt1 ;         stmt2 ;         break ;     case 3:         stmt2 ;         break ;     case 4:         stmt4 ;         break ;     default:         stmt3 ;         break ; } </pre>
---	----------	---

Figure 2.11: An example of **switch** statement in SIMPLE.

## Auxiliary Nodes

Among the auxiliary nodes, we explain the **TREE\_LIST**. This node is used to create lists of different **SIMPLE** statements. An example is shown in Figure 2.12. Statements are linked through **TREE\_CHAIN** field while the **TREE\_VALUE** field points to each statement.

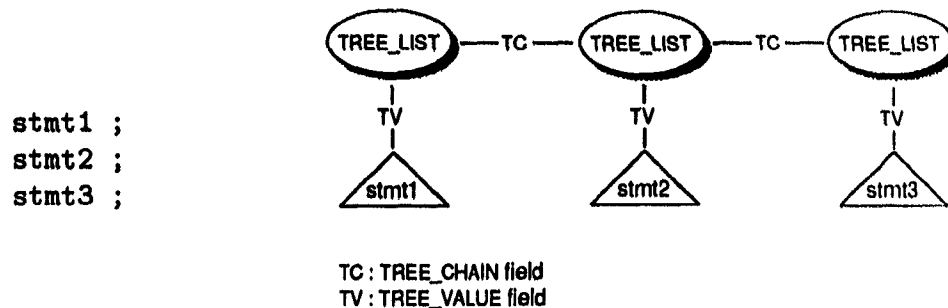


Figure 2.12: An example for the connection between the statements.

A new scope is defined by another **TREE\_LIST** node. If it has some variables, the **TREE\_PURPOSE** field points to the list of variables in that scope. Otherwise, this field would be **NULL**. **TREE\_VALUE** field points to the list of statements in the scope. Figure 2.13 shows this fact (the newly added parts are shown in bold face).

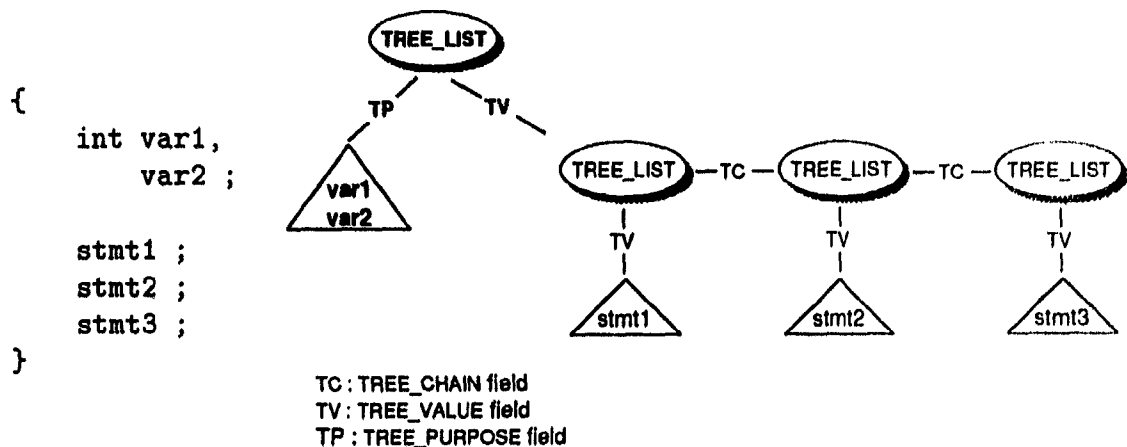


Figure 2.13: An example for the connection between statements and variables.

This construction is done recursively for all statements and functions. It includes the scope appearing in the body of loops and conditions. Consider the example shown in Figure 2.14. The body of the **while** statement is defined in a new scope. Therefore,

a new **TREE\_LIST** node is defined for this scope (shown in bold face).

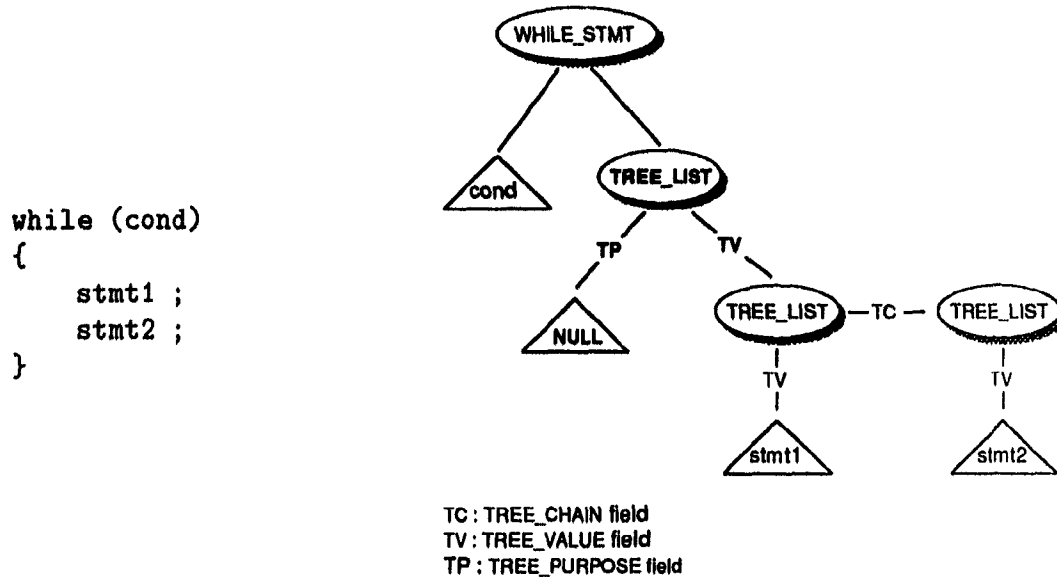


Figure 2.14: An example for the representation of scope in a **while** statement.

The next step is to provide a connection between functions and global variables. A tree node called **AST\_DECL\_NODE** is used for this purpose. Each **AST\_DECL\_NODE** is related to a function or global variable, its **AST\_DECL\_PTR** field points to the function declaration or global variable declaration, its **AST\_DECL\_BODY** points to the body of the function in case of function, or to **NULL** in case of global variable, and its **TREE\_CHAIN** node points to the next **AST\_DECL\_NODE**. The example shown in Figure 2.15 clarifies this connection. By having the head of **AST\_DECL\_NODE**, one has access to the whole **SIMPLE** tree. The head of the tree in the example of Figure 2.15 is called 'decl\_list'.



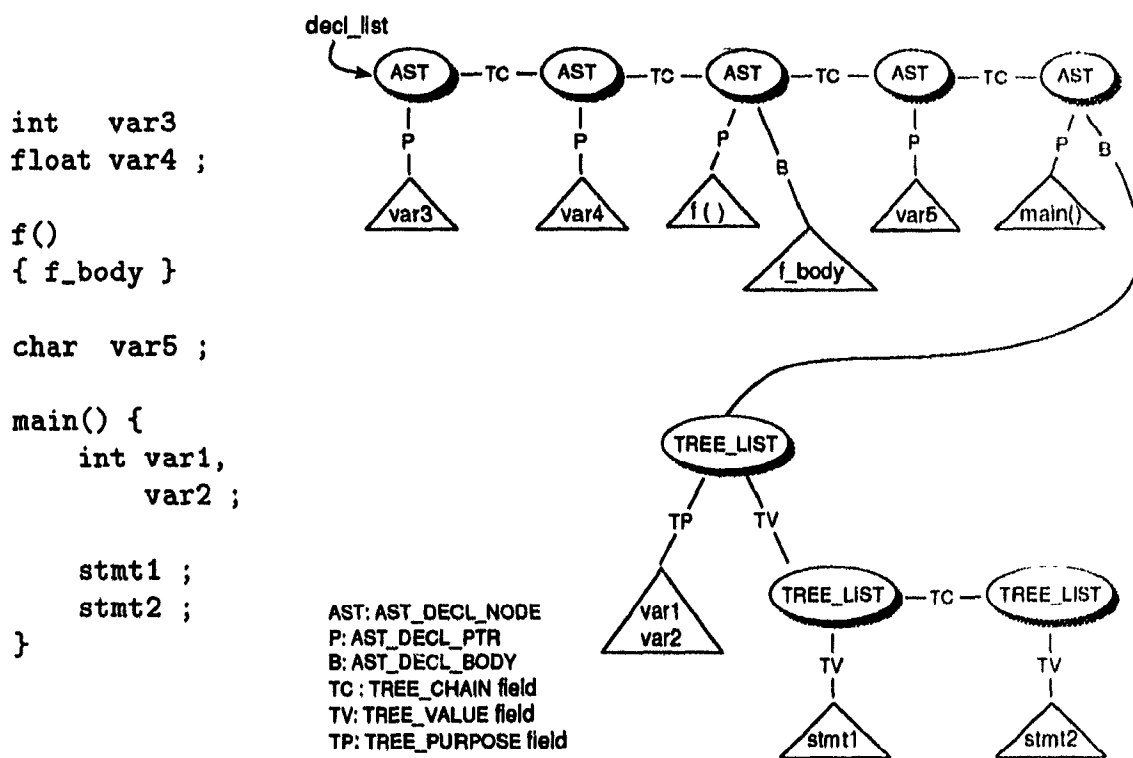


Figure 2.15: An example for the connection between functions and global variables.

## Chapter 3

# Intraprocedural Points-to Analysis for Scalar Variables

In this chapter, we present a detailed explanation of our method for collecting the intraprocedural points-to information for stack-allocated data structures in the C language. This concerns the points-to analyses for the programs where no function calls and aggregate structures (e.g. arrays and structures) are present. This presentation forms the basis for extending the analysis to interprocedural analysis (Chapter 4) and handling aggregate structures (Chapter 5).

This chapter is structured as follows. In Section 3.1, we present a simple example to explain our motivation behind the proposed approach. This example is intended to prepare the reader for the main approach which will be explained in Section 3.2. The relationship between our approach and alias information is explained in Section 3.3. In Section 3.4, we discuss our notations together with the main algorithm. The details of basic statements and compositional control statements are explained in Sections 3.5 and 3.6, respectively. Finally, a summary of the chapter is given in Section 3.7.

### 3.1 Motivating Example

Figure 3.1(a) shows a simple C program. As explained in Chapter 2, a C-program is represented in the SIMPLE AST format where all the analyses are performed. After simplification, we obtain the version shown in Figure 3.1(b). Note that the statement `**x = 5` in Figure 3.1(a) is changed into an equivalent form composed of statements 4 and 5 in Figure 3.1(b).

Figure 3.1 exemplifies our abstraction of intraprocedural analysis at each statement. Notice that we use abstract stack location and memory representation to illustrate the value of each variable. After processing statement 1, variable  $z$  gets the value of constant 1 (Figure 3.1). After processing statement 2, location  $x$  points to location  $y$  (Figure 3.1(d)). After processing statement 3, by following the link in abstract stack or memory representation, we can see that  $*x$  is equivalent to  $y$ . Using this fact, we conclude that the location  $y$  in the abstract stack points to the location  $z$  (Figure 3.1(e)). After processing statement 4,  $temp0$  gets the same value as  $x$  which is equivalent to  $y$ . This means  $temp0$  points to the same location as  $y$  is pointing to, namely  $z$  (Figure 3.1(f)). Finally, after processing statement 5, by following the abstract stack, we see that  $*temp0$  is the same as location  $z$ . Using this fact,  $z$  gets the value of constant 5 (Figure 3.1(g)).

This example shows how we follow the abstract stack location to collect the information about the points-to relationships between the variables.

In the next section, we explain our approach in more detail.

## 3.2 Our Approach

From the discussion in the previous section, we see that our approach uses points to relation as the abstraction to describe the relationship between locations in the abstract stack. It is based on a structured analysis which approximates the relationships between abstract stack locations. At each program point, we collect the points-to information abstracting relationship between each pair of the abstract stack locations, say  $x$  and  $y$ . For such pair, one of the following relationships hold:

1. If the contents of the stack location  $x$  definitely points to the stack location  $y$  at a given program point, we say that  $x$  **definitely-points-to**  $y$ . This is denoted as  $(x,y,D)$ . Figure 3.2 shows an example.
2. If the contents of the stack location  $x$  possibly points to the stack location  $y$  at a given program point, we say that  $x$  **possibly-points-to**  $y$ . This is denoted as  $(x,y,P)$ . Figure 3.3 shows an example.
3. The contents of stack location  $x$  does not point to stack location  $y$ .

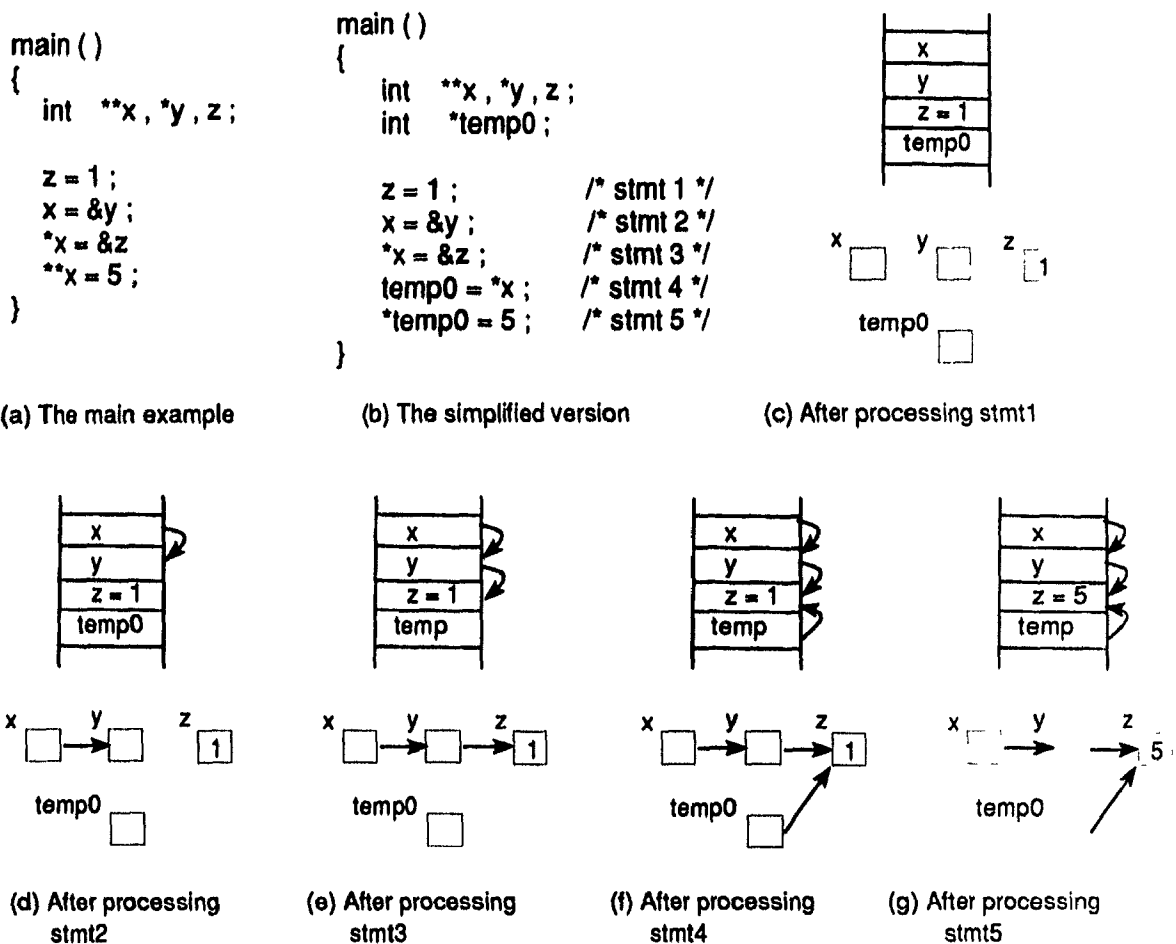


Figure 3.1: A motivating example behind our approach to intraprocedural points-to analysis.

### 3.3 Rules to Convert *Points-to* Information to *Alias* information

As explained briefly in the introduction, traditionally alias pairs were used as an abstraction to determine the relationships between memory locations. In this section, we show how to transform points-to relationships to alias pair relationships. The rules for converting points-to relationships to alias pair relationships are:

1. A relationship of the form  $(x, y, rel)$ , where  $rel$  is either  $D$  or  $P$ , is considered to be equivalent to  $\langle *x, y \rangle$ . For example, in the program:

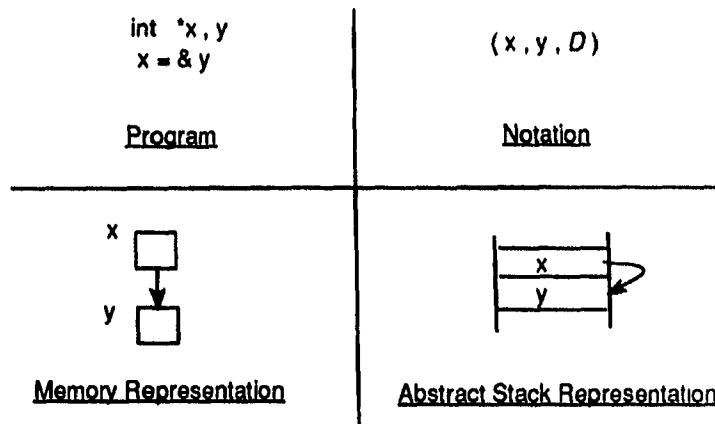


Figure 3.2: An example of a definitely-points-to relationship.

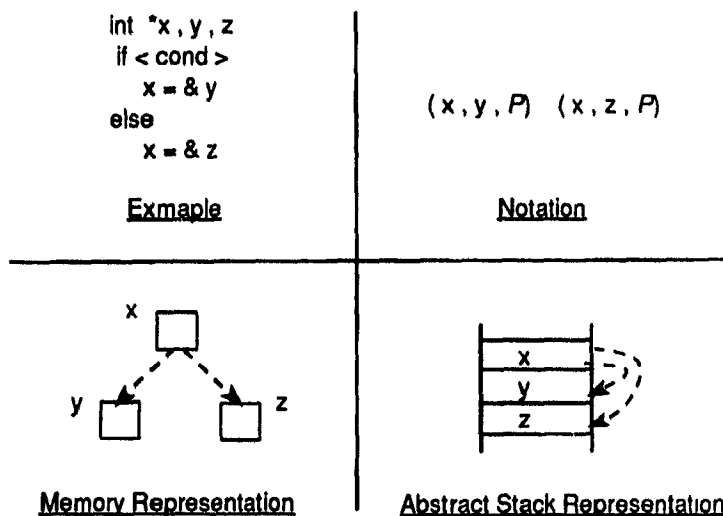


Figure 3.3: An example of a possibly-points-to relationship.

```
int *a, b;
a = &b ;
```

we say that *a* points-to *b*, or equivalently, *\*a* is aliased to *b*.

2. For each relationship of the form  $\langle x, y \rangle$ , apply the '\*' as long as the type of variable allows. For example in program:

```
int ***a, **b;
a = &b ;
```

by applying the first rule, we obtain  $\langle *a, b \rangle$ , and by applying the second rule we obtain:  $\langle **a, *b \rangle$  and  $\langle ***a, **b \rangle$ .

3. Apply the *commutative rule*<sup>1</sup> and the *transitive closure*<sup>2</sup> on the result of each alias pair obtained so far. This has to be done recursively until no new pairs can be generated. For example in  $(x, y, rel)$  and  $(z, y, rel)$ , using the first rule we can obtain:  $\langle *x, y \rangle$ ,  $\langle *z, y \rangle$ . Then, by applying the commutative rule on the second pair, we get:  $\langle *x, y \rangle$ ,  $\langle y, *z \rangle$ . Finally, using the transitive closure results in  $\langle *x, *z \rangle$ .

Note that the commutative rule does not give a new pair. However, it provides a different way of looking at the pairs which will eventually help us to apply the transitive closure.

Here is a more complicated example which uses all these three rules together:

```
int **d, *a, *b, c
d = &a ;
a = &c ;
b = &c ;
```

Our result:  $(d, a, D) (a, c, D) (b, c, D)$

Applying rule(1):  $\langle *d, a \rangle \langle *a, c \rangle \langle *b, c \rangle$

Applying rule(2):  $\langle *d, a \rangle \langle ***d, *a \rangle \langle *a, c \rangle \langle *b, c \rangle$

Applying rule(3):  $\langle *d, a \rangle \langle ***d, *a \rangle \langle *a, c \rangle \langle *b, c \rangle \langle ***d, c \rangle$   
 $\langle *a, *b \rangle \langle ***d, *b \rangle$

We conclude that, in general, the alias information can be easily obtained from our more concisely represented points-to relationships.

In each points-to relationship  $(x, y, rel)$ , the type of  $x$  is always a pointer to the type of  $y$ <sup>3</sup>. As a result of this, the definitely-points-to and possibly-points-to relationships are neither commutative nor transitive.

### 3.3.1 Justification of the Proposed Approach

Our reasons behind using the proposed approach are as follows:

<sup>1</sup>  $a \text{ op } b \Rightarrow b \text{ op } a$ .

<sup>2</sup>  $a \text{ op } b \wedge b \text{ op } c \Rightarrow a \text{ op } c$ .

<sup>3</sup> An exception to this rule is the special case of *type casting* which will be discussed in Chapter 5.

1. It is straight-forward and easy to understand because it is similar in nature to the way in which variables are actually stored in the stack.
2. It gives accurate results with which we can perform an efficient dependence analysis using the reads and writes to the abstract stack locations.
3. It provides sufficient information about the alias pairs in a compact way

### 3.4 Notations and Algorithm

The definitions and notations which will be used throughout this thesis are summarized in Table 3.1.

*	:	indirect reference in the C-language, e.g. <code>*x</code>									
&	:	address operator in the C-language, e.g. <code>&amp;x</code>									
rhs	:	right-hand side of an assignment									
lhs	:	left-hand side of an assignment									
$\forall$	:	for all									
$\wedge$	:	and									
base-type	:	int   float   char   void   <i>etc.</i>									
ptr-type	:	ptr-type *   base-type *									
rel1 $\bowtie$ rel2	:	merge two relationships using the following table									
<table> <tr> <td><math>\bowtie</math></td><td>D</td><td>P</td></tr> <tr> <td>D</td><td>D</td><td>P</td></tr> <tr> <td>P</td><td>P</td><td>P</td></tr> </table>			$\bowtie$	D	P	D	D	P	P	P	P
$\bowtie$	D	P									
D	D	P									
P	P	P									

Table 3.1: Definitions and notations used in the thesis.

In general, each statement is either a basic statement or a compositional control statement. In the intraprocedural analysis, the definitely-points-to relationships are generated/resolved by the basic statements. The data flow merge forced by compositional control statements like loop and conditionals, due to unknown flow of control causes the appearance of possibly-points-to relationships.

Figure 3.4 gives an overall view of the algorithm for the intra-procedural points to analysis. The algorithm for basic statements (`basic_points_to`) is discussed in Section 3.5 (Figure 3.6). The algorithm for compositional control statements (`control_points_to`) is discussed in Section 3.6 (Figure 3.15).

```

/* Given a basic statement and input information, return the output
 * information (where input.in is the points_to relationships) */
points_to (S,input) =
  if basic_statement(S) then
    /* S is basic statement */
    return(basic_points_to(S,input));
  else if control_statement(S) then
    /* S is a compositional control statement */
    return(control_points_to(S,input));
  else
    return(input) ;

```

Figure 3.4: General algorithm for intraprocedural analysis.

### 3.5 Basic Statements

The simplifying process reduces the many different cases that one may encounter in a typical program (refer to Figure 2.7) to the eight types of basic statements listed in Table 3.2. Case 2, 6 and 7 are discussed in this chapter, case 4 and 8 are discussed in Chapter 4, and case 1 and 3 are discussed in Chapter 5. Note that case 5 does not introduce any points-to information and consequently is not of any concern to us.

(1) x = a binop b *x = a binop b	(2) x = y *x = y	(3) x = cast y *x = cast y	(4) f(args)
(5) x = unop a *x = unop a	(6) x = *y *x = *y	(7) x = &y *x = &y	(8) x = f(args) *x = f(args)

Table 3.2: All possible Basic Statements.

The type of the variables plays an important role in our approach to points-to analysis. A basic statement will be studied only if it is involved in an assignment with a pointer type variable. Let us consider the example illustrated in Figure 3.5.

Since *\*x* and *\*y*, in statement *\*x = \*y*, are not of pointer type, this statement does not make any difference in the points-to information. Therefore, it is not of concern to us. Now consider the example:



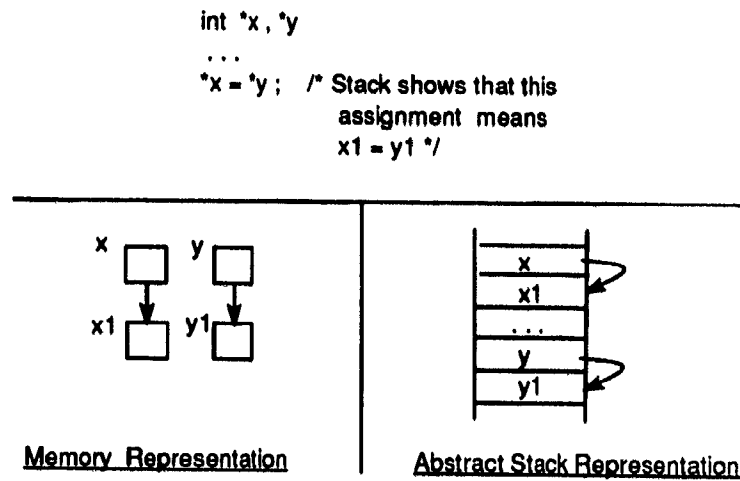


Figure 3.5: An example where no pointer is involved in the assignment statement.

```

int *x, **y ;
...
x = *y ;

```

In this example, since variables `x` and `*y` are of pointer type, after the execution of statement `x = *y`, the points-to information of `x` will be affected (`x` points to the same location as `*y` is pointing to). Therefore, this statement should be considered in the points-to analysis.

We have classified all the cases studied in this chapter according to the basic statements and the type information. These are summarized in Table 3.3.

lhs \ rhs	&y	y	*y
x	1	2	3
*x	4	5	6

Table 3.3: Basic statements to be studied in this chapter, when lhs (or rhs) is of pointer type.

The algorithm for the basic statements of scalar variables is given in Figure 3.6 (we will discuss arrays and structures in Chapter 5). Each entry in the Table 3.3 has a corresponding rule in the function *basic\_points\_to*. In each case, we compute: (i) the

```

/* Given a basic statement and input information, return the output
 * information (input.in is the points_to relationships) */
basic_points_to (S,input) =
  if ( ! pointer_type(S))
    return( input ) ;
  else case S of
    /* rule 1 */
    < x = &y > :
      { kill = { (x,x1,rel) | (x,x1,rel) ∈ input };
        gen = { (x,y,D) };
        return( gen ∪ (input - kill) );
      }
    /* rule 2 */
    < x = y > :
      { kill = { (x,x1,rel) | (x,x1,rel) ∈ input };
        gen = { (x,y1,rel) | (y,y1,rel) ∈ input };
        return( gen ∪ (input - kill) );
      }
    /* rule 3 */
    < x = *y > :
      { kill = { (x,x1,rel) | (x,x1,rel) ∈ input };
        gen = { (x,y2,rel1 ⋈ rel2) | (y,y1,rel1), (y1,y2,rel2) ∈ input };
        return( gen ∪ (input - kill) );
      }
    /* rule 4 */
    < *x = &y > :
      { kill = { (x1,x2,rel) | (x,x1,D), (x1,x2,rel) ∈ input };
        change = (input - { (x1,x2,D) | (x,x1,P), (x1,x2,D) ∈ input })
                  ∪ { (x1,x2,P) | (x,x1,P), (x1,x2,D) ∈ input };
        gen = { (x1,y,rel) | (x,x1,rel) ∈ input };
        return( gen ∪ (change - kill) );
      }
    /* rule 5 */
    < *x = y > :
      { kill = { (x1,x2,rel) | (x,x1,D), (x1,x2,rel) ∈ input };
        change = (input - { (x1,x2,D) | (x,x1,P), (x1,x2,D) ∈ input })
                  ∪ { (x1,x2,P) | (x,x1,P), (x1,x2,D) ∈ input };
        gen = { (x1,y1,rel1 ⋈ rel2) | (x,x1,rel1), (y,y1,rel2) ∈ input };
        return( gen ∪ (change - kill) );
      }
    /* rule 6 */
    < *x = *y > :
      { kill = { (x1,x2,rel) | (x,x1,D), (x1,x2,rel) ∈ input };
        change = (input - { (x1,x2,D) | (x,x1,P), (x1,x2,D) ∈ input })
                  ∪ { (x1,x2,P) | (x,x1,P), (x1,x2,D) ∈ input };
        gen = { (x1,y2,rel1 ⋈ rel2 ⋈ rel3) | (x,x1,rel1), (y,y1,rel2), (y1,y2,rel3) ∈ input };
        return( gen ∪ (change - kill) );
      }
    /* any other simple statement */
    < _ > : return(input);
  }

```

Figure 3.6: Algorithm of basic statements for scalar variables.

points-to information which gets killed, (ii) the changes in the points-to information where some definitely-points-to relationship becomes possibly-points-to, and, (iii) the points-to information which is generated. Then, we return *input* (the original set of points-to relationships) while updating the information by removing the killed information, incorporating the changes, and adding the newly created information. In the next sections, we examine each case in detail. To facilitate the understanding, we use a simple pointer type, *int\** or *int\*\**.

### 3.5.1 Case 1

This is the case denoted as *rule1* in Figure 3.6.

```
int *x, y ;
...
x = &y ;
```

The rules applied in this case are as follows:

```
kill = { (x,x1,rel) | (x,x1,rel) ∈ input }
gen = { (x,y,D) }
return( gen ∪ (input - kill) )
```

Referring to Figure 3.7, we see that after processing the statement *x = &y*, *x* should point to the location *y*. This means that all the relations in which *x* points to a variable should be removed from the points-to information (*kill* set). Then, *x* definitely-points-to *y* is the unique element of the newly generated set (*gen* set). Finally, by removing the *kill* set and adding the *gen* set to *input* set (the original set of information), we obtain the new set of information.

### 3.5.2 Case 2

This case is denoted as *rule2* in Figure 3.6.

```
int *x, *y ;
...
x = y ;
```

The rules applied in this case are as follows:

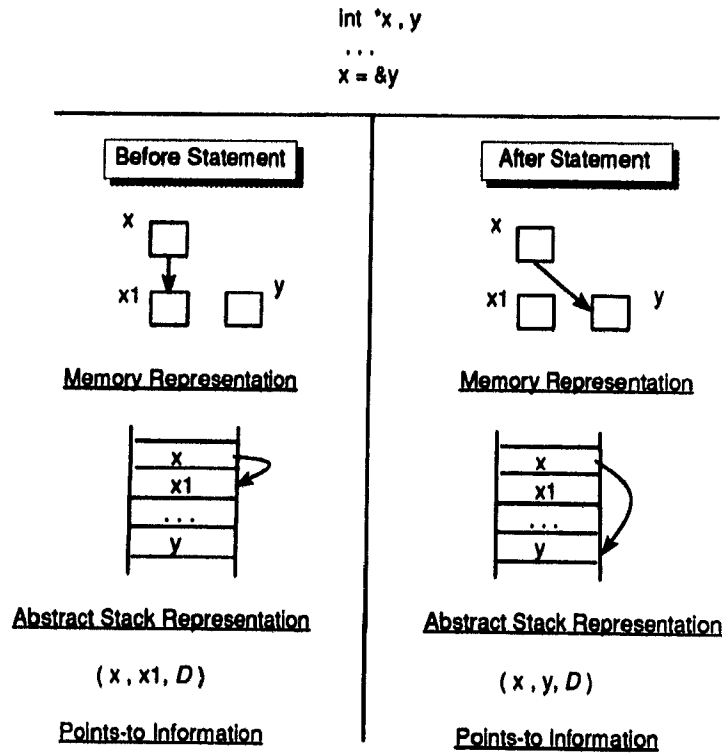


Figure 3.7: An example of rule 1 in the basic statements.

$kill = \{ (x, x1, rel) \mid (x, x1, rel) \in input \}$   
 $gen = \{ (x, y1, rel) \mid (y, y1, rel) \in input \}$   
**return**(  $gen \cup (input - kill)$  )

After processing the statement  $x = y$ ,  $x$  gets changed. This is shown in Figure 3.8. Therefore, we should remove all the relations in which  $x$  points to a variable  $x1$  from the points-to information (*kill* set). Then make  $x$  point to the same location as  $y$  is pointing to (*gen* set). The new set of information can be obtained by removing the *kill* set from the *input* set and adding the *gen* set.

A more complicated, self explanatory example is given in Figure 3.9

### 3.5.3 Case 3

This case is denoted as *rule3* in Figure 3.6.

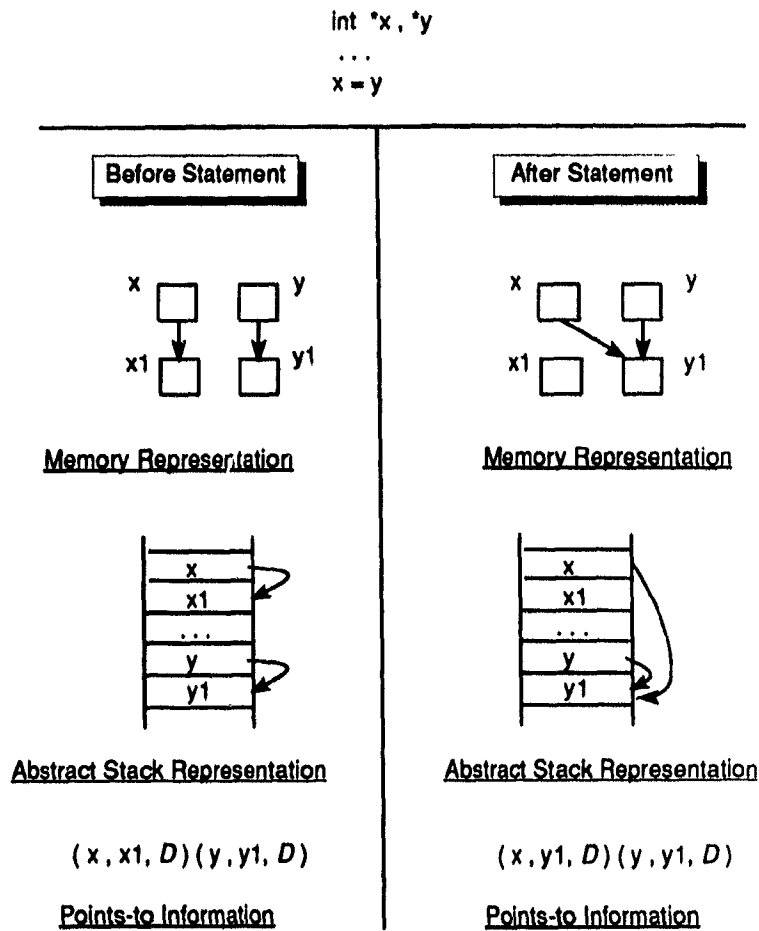


Figure 3.8: An example of rule 2 in the basic statements.

```

int *x, **y;
...
x = *y ;

```

The rules applied in this case are as follows:

```

kill = { (x,x1,rel) | (x,x1,rel) ∈ input }
gen = { (x,y2,rel1 ⋈ rel2) | (y,y1,rel1), (y1,y2,rel2) ∈ input }
return( gen ∪ (input - kill) )

```

Referring to Figure 3.10, after processing the statement `x = *y`, `x` should point to the same location as `*y` is pointing to. This means that all the relations in which `x` points to a variable, should be removed from the points-to information (*kill* set) and `x`

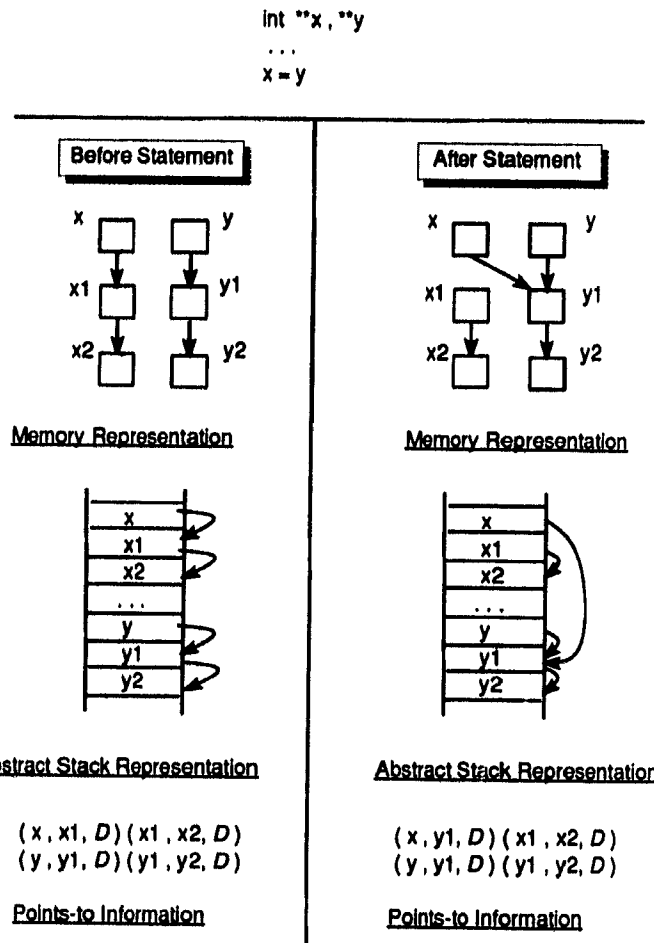


Figure 3.9: A more complicated example of rule 2 in the basic statements.

should point to the same location as  $*y$  is pointing to. If  $y$  definitely-points-to location  $y1$ , and  $y1$  definitely-points-to location  $y2$ , we can say that  $x$  definitely-points-to  $y2$  (because it is guaranteed that  $*y$  points to  $y2$ ). If one of these relationships is possibly-points-to, then  $x$  possibly-points-to  $y2$  (because it is not guaranteed that  $*y$  points to  $y2$ ). The new set of information can be obtained by removing the *kill* set from the original set of information and adding the *gen* set.

#### 3.5.4 Case 4

This case is denoted as *rule4* in Figure 3.6.

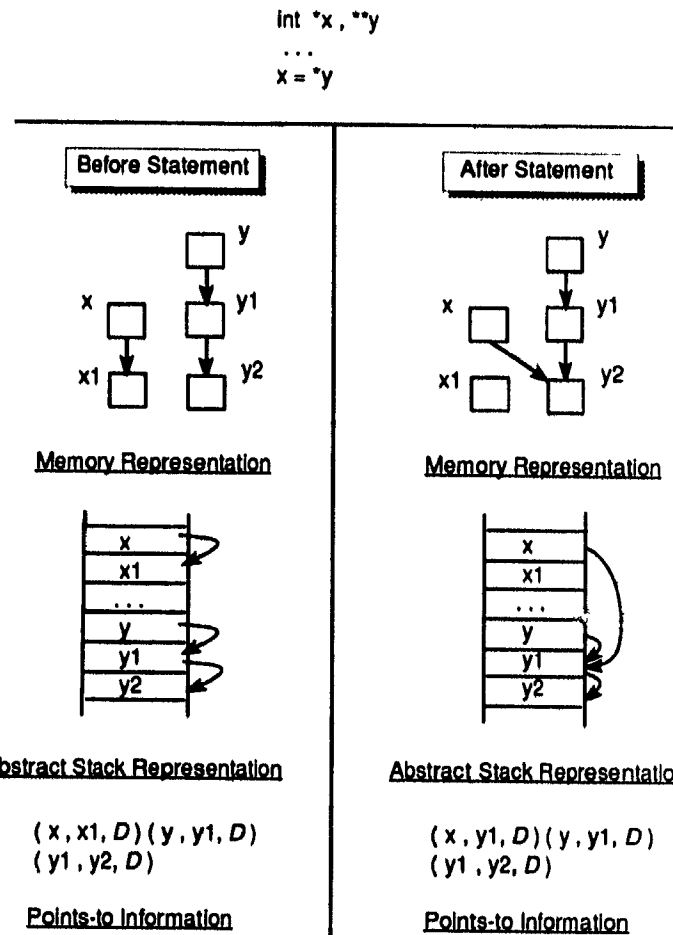


Figure 3.10: An example of rule 3 in the basic statements.

```

int **x, y ;
...
*x = &y ;

```

The rules applied in this case are as follows:

```

kill = { (x1,x2,rel) | (x,x1,D), (x1,x2,rel) ∈ input }
gen = { (x1,y,rel) | (x,x1,rel) ∈ input }
changed_input = (input - { (x1,x2,D) | (x,x1,P), (x1,x2,D) ∈ input })
                ∪ { (x1,x2,P) | (x,x1,P), (x1,x2,D) ∈ input }
return( gen ∪ (changed_input - kill) )

```

Referring to Figure 3.11, after processing the statement  $*x = \&y$ , if  $x$  definitely-points-to  $x1$ , the modification of location  $x1$  is guaranteed and we remove the relations in which  $x1$  points to a variable from points-to information (*kill set*). If  $x$  possibly-points-to  $x1$ , changing of  $x1$  is not guaranteed. In this case, we change all the definitely-points-to relationship of  $x1$  into possibly-points-to (*changed\_input set*). If  $x$  definitely-points-to  $x1$ , then  $x1$  definitely-points-to  $y$ . But, if  $x$  possibly-points-to  $x1$ , then  $x1$  possibly-points-to  $y$ . So, the way that  $x1$  points to  $y$  depends on the way that  $x$  points to  $x1$  (*gen set*). Finally, by removing the *kill set* from the *changed\_input set* and adding the *gen set* to it, we obtain the final information.

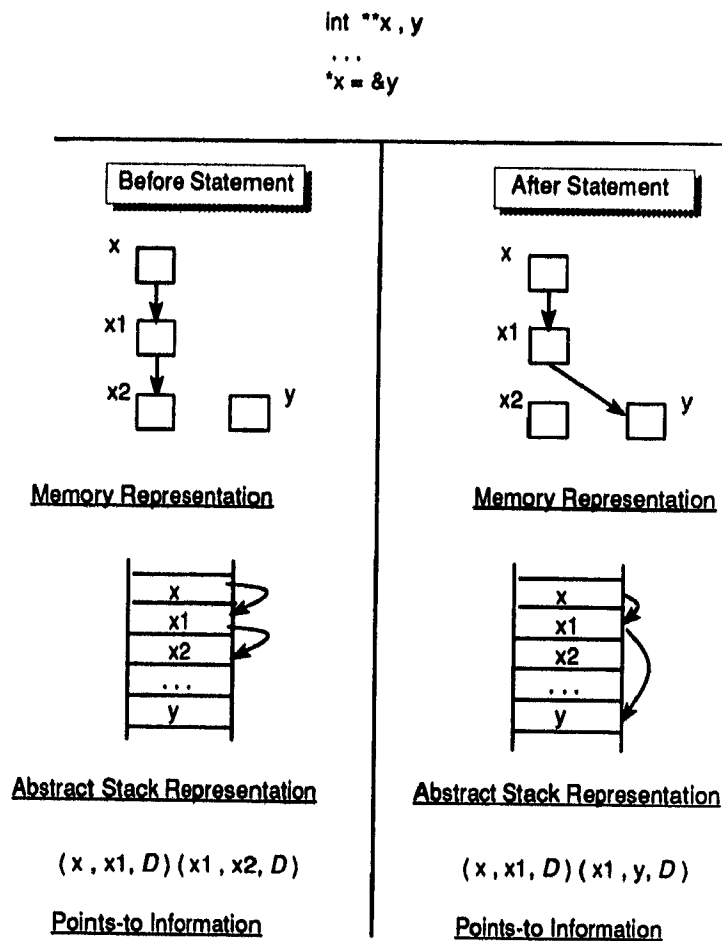


Figure 3.11: An example of rule 4 in the basic statements.



### 3.5.5 Case 5

This case is denoted as *rule5* in Figure 3.6.

```
int **x, *y ;  
...  
*x = y ;
```

The rules applied in this case are as follows:

```
kill = { (x1,x2,rel) | (x,x1,D), (x1,x2,rel) ∈ input }  
gen = { (x1,y1,rel1 ⋈ rel2) | (x,x1,rel1), (y,y1,rel2) ∈ input }  
changed_input = (input - { (x1,x2,D) | (x,x1,P), (x1,x2,D) ∈ input })  
                ∪ { (x1,x2,P) | (x,x1,P), (x1,x2,D) ∈ input }  
return( gen ∪ (changed_input - kill) )
```

Figure 3.12 and Figure 3.13 help in understanding the analysis. After processing the statement  $*x = y$ , the *kill* set and the *changed\_input* set are computed in the same way as in case 4. If  $x$  has a points-to relationship *rel1* with  $x1$  and  $y$  has a points-to relationship *rel2* with  $y1$ , then  $x1$  will have a points-to relationship of the form  $rel1 \bowtie rel2$  with  $y1$ . This relationship says that  $x1$  definitely-points-to  $y1$  if *rel1* and *rel2* are of definitely-points-to type, otherwise,  $x1$  possibly-points-to  $y1$ . Finally, by removing the *kill* set from the *changed\_input* set and adding the *gen* set to it we obtain the final information.

### 3.5.6 Case 6

This case is denoted as *rule6* in Figure 3.6.

```
int **x, **y ;  
...  
*x = *y ;
```

The following rules are applied in this case:

```
kill = { (x1,x2,rel) | (x,x1,D), (x1,x2,rel) ∈ input }  
gen = { (x1,y2,rel1 ⋈ rel2 ⋈ rel3) | (x,x1,rel1), (y,y1,rel2), (y1,y2,rel3)  
      ∈ input }
```

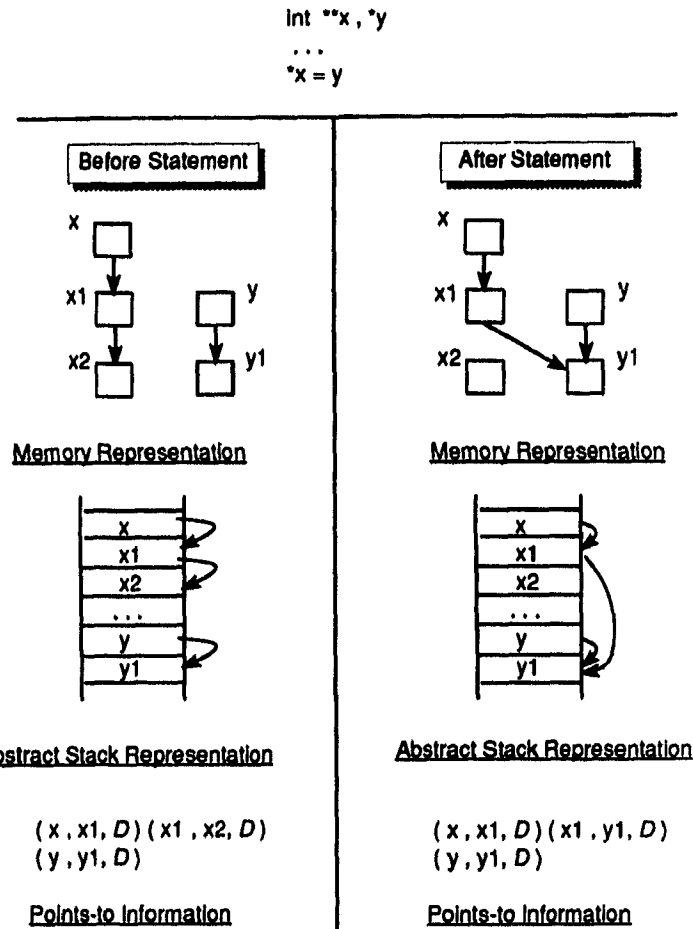


Figure 3.12: An example of rule 5 in the basic statements.

```

changed_input = (input - { (x1,x2,D) | (x,x1,P), (x1,x2,D) ∈ input })
                ∪ { (x1,x2,P) | (x,x1,P), (x1,x2,D) ∈ input }
return( gen ∪ (changed_input - kill) )

```

Referring to Figure 3.14, after processing the statement  $*x=*y$ , the *kill* set and the *changed\_input* set are computed in the same way as in case 4. If  $x$  has a points-to relationship  $rel1$  with  $x1$ ,  $y$  has a points-to relationship  $rel2$  with  $y1$ , and  $y1$  has a points-to relationship  $rel3$  with  $y2$ , then  $x1$  will have a points-to relationship of the form  $\{rel1 \bowtie rel2 \bowtie rel3\}$  with  $y2$ . This relationship says that  $x1$  definitely-points-to  $y2$  if  $rel1, rel2$  and  $rel3$  are all of definitely-points-to type. otherwise,  $x1$  possibly-points-to  $y2$ . Finally, by removing the *kill* set from the *changed\_input* set, and adding the *gen* set to it, we obtain the final information.

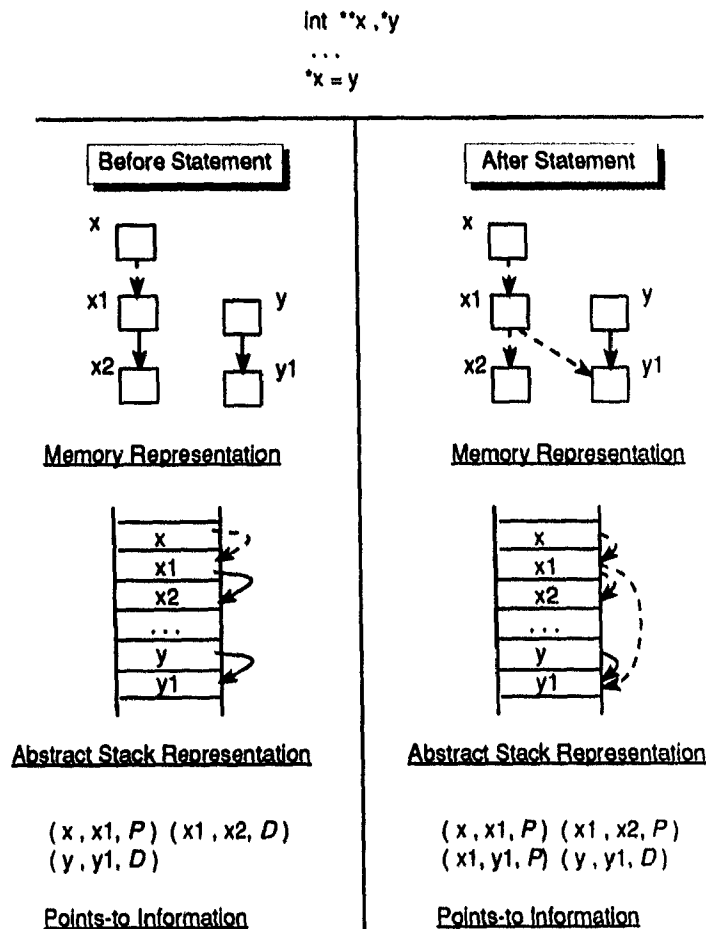


Figure 3.13: An example of rule 5 that contains possibly points-to relationship in the basic statements.

### 3.6 Compositional Control Statements

In the previous section, we presented our analysis method for simple statements. In this section, we extend our method to programs that include control statements. SIMPLE supports the following control statements: **if**, **for**, **while**, **do-while**, **switch**, **continue**, and **break** (note that **return** statement appears in interprocedural analysis, therefore it is explained in Chapter 4). To simplify the explanation, we first consider compositional control statements without the presence of **break** and **continue** statements. Then, the more general case containing **break** and **continue** statements is studied. To support control statements such as conditionals and loops.

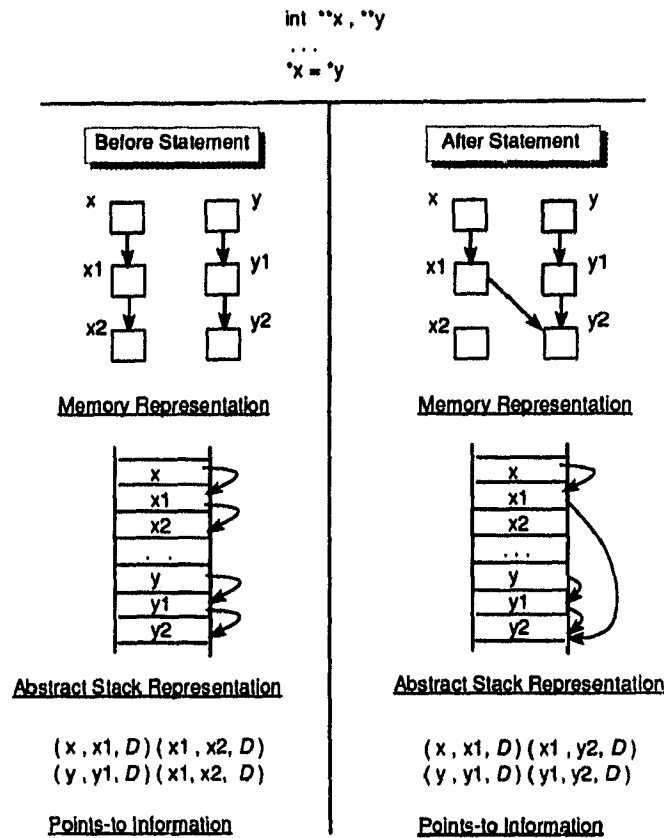


Figure 3.14: An example of rule 6 in the basic statements.

we need to define the following two concepts: (i) merge between sets, and (ii) fixed-point computation.

The concept of merge between definitely-points-to and possibly-points-to relationships was defined in Table 3.1. Here we extend this definition to the case of merging two sets of relationships. This is denoted as **Merge(S1, S2)** where S1 and S2 are the sets of the relationships to be merged. The definition is as follows:

$$\text{Definite-set} = \{(x, y, D) \mid (x, y, D) \in (S1 \cap S2)\}$$

$$\text{Possible-set} = \{(x, y, P) \mid (x, y, \text{rel}) \in (S1 \cup S2) \wedge (x, y, D) \notin \text{Definite-set}\}$$

$$\text{Merge}(S1, S2) = \text{Definite-set} \cup \text{Possible-set}$$

where 'rel' is either D or P. Here is an example:

$$S1 = \{(a, a1, D), (b, b1, D), (c, c1, D)\}$$

$$S2 = \{(a, a1, D), (b, b1, P)\}$$

Definite-set = { (a,a1,D) }

Possible-set = { (b,b1,P), (c,c1,P) }

Merge(S1, S2) = { (a,a1,D), (b,b1,P), (c,c1,P) }

Note that the relationship (a,a1,D) is in the Definite-set because 'a' and 'a1' have a definitely-points-to relationship in both S1 and S2. The relationship (b,b1,P) is in the Possible-set because 'b' and 'b1' have definitely-points-to relationship in S1 but they have possibly-points-to relationship in S2. The relationship (c,c1,P) is in the Possible-set because the relationship between 'c' and 'c1' is in S1 and not in S2.

The concept of **fixed-point** computation is defined in conjunction with the analysis of loops and recursion. We say that a fixed-point is reached when analysis of the loop-body in two successive iterations does not result in any new information. An example of fixed-point will be given later in our discussion concerning while loop.

Figure 3.15 gives an outline of the algorithm for the analysis of the control statements. Each of the function calls will be discussed in detail in the rest of this section. Note that in SIMPLE, the expressions in conditionals and loops are side effect free, so we ignore them during the analysis. For the same reason, the expression of the switch statement is also ignored.

### 3.6.1 Points-to Analysis without break and continue

#### if statement:

For the if statement, we propagate the input to both the then-body and the else-body. After processing both parts, we merge the information to get the new input set. In the absence of the else-body, the output of else-body is the same as the input to the if statement. Therefore, the output of the then-body is merged with the input to the if statement. The corresponding algorithm (denoted as 'process if' in Figure 3.15), together with its graph representation, is given in Figure 3.16. Note that the parameter **in** is a structure that contains the points-to information and other fields which are useful during the analysis (e.g. the fields related to break and continue that is explained in Section 3.6.2).

#### while statement:

In general, the body of a while-loop, depending on the while-loop condition, is executed for  $n = 0, 1, 2, \dots$  times. As we are not sure about the exact number of the iterations, we have to approximate the output of our analysis. The first approximation corresponds to the initial stage before starting the iterations. In this case, the output

```

/* Given a basic statement and input information, return the output
 * information (input.in is the points_to relationships) */
control_points_to (S,input) =
  case S of
    < IF(cond,s_then,s_else) > :
      return(process_if(cond,s_then,s_else,input));

    < SWITCH(expr,caselist) > :
      return(process_switch(expr,caselist,input));

    < WHILE(cond,s_body) > :
      return(process_while(cond,s_body,input));

    < DO(s_body,cond) > :
      return(process_do(s_body,cond,input));

    < FOR(s_init,cond,s_iter,s_body) > :
      return(process_for(s_init,cond,s_iter,s_body, input));

    < BREAK > :
      return(process_break(input)) ;

    < CONTINUE > :
      return(process_continue(input)) ;

    < _ > :
      return(input) ;

```

Figure 3.15: Algorithm of the compositional control statements.

```

process_if(cond, then_body, else_body, in) {
  out1 = points_to(then_body, in) ;
  out2 = points_to(else_body, in) ;
  return(merge_info(out1, out2)) ;
}

merge_info(in1, in2) {
  out = in1 ;
  /* merge the points-to information
   * field of the two given input */
  out.in = Merge(in1.in, in2.in) ;
  return(out) ;
}

```

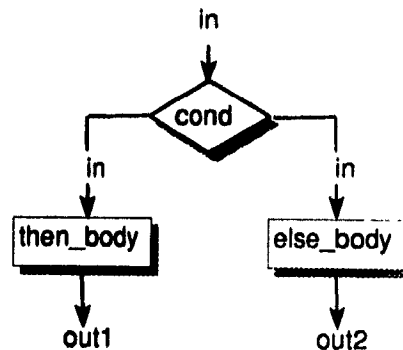


Figure 3.16: Algorithm of the if statement for points-to analysis.

is the same as the input. The second approximation is obtained by merging the following sets of information:

- the previous approximation,
- the output obtained from processing the while-body with the previous approximation as the input.

This process is repeated until a fixed-point is reached. The corresponding algorithm (denoted as 'process\_while' in Figure 3.15), together with its graph representation, is shown in Figure 3.17.

```
process_while(cond, body, in) {
{
    last_in = in ;
    out = points_to(body, in) ;
    in = merge_info(in, out) ;
} while (last_in != in) ;

result = in ;
return(result) ;
}
```

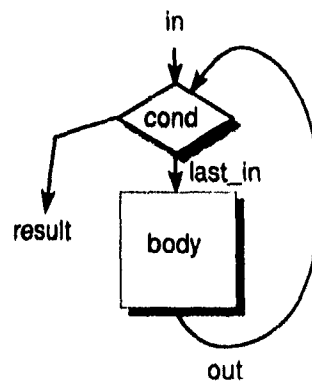


Figure 3.17: Algorithm of the **while** statement for points-to analysis.

Following example demonstrates the process of finding the fixed-point of a while-loop. The information represented for each statement, is after processing the statement. The following shows the fixed-point calculation for a while-loop:

int *a, *b, c, d ;		First approximation
		-----
b = &c ;	/* s1 */	{(b,c,D)}
while (cond){	/* s2 */	{(b,c,D)}
a = b ;	/* s3 */	{(b,c,D), (a,c,D)}
b = &d ;	/* s4 */	{(b,d,D), (a,c,D)}
}	/* s5 */	{(b,c,P), (b,d,P), (a,c,P)}

#### Second approximation

```

/* s1 */ {(b,c,D)}
/* s2 */ {(b,c,P), (b,d,P), (a,c,P)}
/* s3 */ {(b,c,P), (b,d,P), (a,c,P), (a,d,P)}
/* s4 */ {(b,d,D), (a,c,P), (a,d,P)}
/* s5 */ {(b,c,P), (b,d,P), (a,c,P), (a,d,P)}

```

#### Third approximation

```

/* s1 */ {(b,c,D)}
/* s2 */ {(b,c,P), (b,d,P), (a,c,P), (a,d,P)}
/* s3 */ {(b,c,P), (b,d,P), (a,c,P), (a,d,P)}
/* s4 */ {(b,d,D), (a,c,P), (a,d,P)}
/* s5 */ {(b,c,P), (b,d,P), (a,c,P), (a,d,P)}

```

The input to the loop at each iteration is the set represented at statement s2 and the corresponding output is the set represented at statement s5. We observe that after the third iteration the information does not change anymore (fixed-point of the loop). Therefore, the final points-to information is the information obtained at this stage.

#### do-while statement

The **do-while** statement is similar to the while-loop with the difference that the body of the **do-while** is processed for  $n = 1, 2, \dots$  times. This means that the body will be processed at least once. Therefore, the first approximation to the do-loop is what we have after analyzing the loop-body once. The corresponding algorithm (denoted as 'process.do' in Figure 3.15), together with its graph representation, is given in Figure 3.18.



```

process_do(cond, body, in) {
    out = points_to(body, in) ;
    in = out ;
    {
        last_in = in ;
        out = points_to(body, in) ;
        in = merge_info(in, out) ;
    } while (last_in != in) ;

    result = in ;
    return(result) ;
}

```

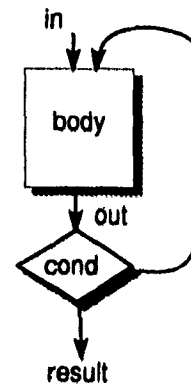


Figure 3.18: Algorithm of the **do-while** statement for points-to analysis.

#### for statement:

The **for** statement is also similar to the **while** statement. The difference is that the initial statement has to be processed before processing the body and the iteration statement has to be processed after processing the body. The corresponding algorithm (denoted as 'process\_for' in Figure 3.15), together with its graph representation, is given in Figure 3.19. If it can be determined at compile time that the for loop will iterate at least once, then a more accurate answer might be obtained by processing the body once before computing the fixed point.

```

process_for(init_stmt, cond, iter_stmt, body, in) {
    in = points_to(init_stmt, in) ;
    {
        last_in = in ;
        out1 = points_to(body, in) ;
        out2 = points_to(iter_stmt, out1) ;
        in = merge_info(in, out2) ;
    } while (last_in != in) ;

    result = in ;
    return(result) ;
}

```

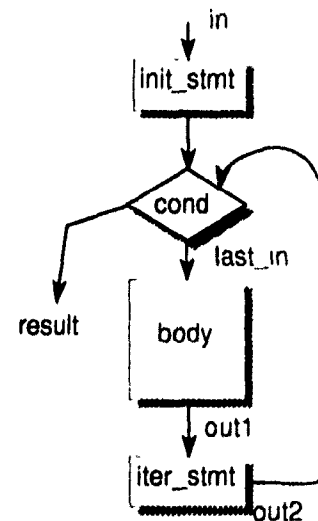


Figure 3.19: The algorithm of **for** statement for points-to analysis.

### 3.6.2 Points-to Analysis with break and continue

In this section, we present our analysis for programs with **break** and **continue** statements. First, let us recall the semantics of **break** and **continue**. Execution of a **break** statement terminates the execution of the closest **while**, **do-while**, **for** or **switch** statement. Then, the program control is immediately transferred to the point just after the body of the corresponding statement.

Execution of a **continue** statement terminates the execution of the body of the closest **while**, **do-while**, or **for** statement. Then, the program control is immediately transferred to the beginning of the body and the execution continues from that point with a re-evaluation of the loop condition. In the case of the **for-loop**, the iteration expression is also re-evaluated.

Note that the **continue** statement has no interaction with the **switch** statement. The **continue** statement within the body of a **switch** statement actually belongs to the closest **while**, **do-while**, or **for** loop.

In our analysis, we handle **break** and **continue** statements as follows: We define two structures, called the **break-list** and **continue-list**, to collect the points-to information at **break** points and at **continue** points encountered during processing the **while**, **do-while**, **for**, and **switch** statements. Whenever we encounter a **break/continue** statement, we save the corresponding points-to information in **break-list/continue-list** and pass **BOTTOM** as the output where **BOTTOM** denotes no information. Propagating **BOTTOM** corresponds to taking paths in the program that would never occur in any execution (dead-code). Any statement with the input **BOTTOM**, produces **BOTTOM** as the output. The merge rule for **BOTTOM** is as follows:

$$\text{Merge}(S1, \text{BOTTOM}) = \text{Merge}(\text{BOTTOM}, S1) = S1$$

The intuition behind this rule is that a path resulting in **BOTTOM** corresponds to an impossible execution path and, consequently, can be ignored in the merge. This fact is represented in Figure 3.20. The input to the **if** is called **in**, and the result of **then** statements (**S1** and **S2**) is represented as **out1** and the result of **else** statement (**S3**) is represented as **out2**. After processing the **break** statement, **BOTTOM** is passed as the result. This path is not a valid path, therefore only the path coming from **S3** can reach to this point. That is why the input to **S4** can be only **out2**. In other words, that is why the merge of **out2** and **BOTTOM** is **out2**.

In the following, we explain how to use the information gathered in the **break-list**

```

main() {
    ...
    while(cond1) {
        if (cond2)
        {
            S1 ;
            S2 ;
            break ;
        }
        else{
            S3 ;
        }
        S4 ;
    }
    ...
}

```

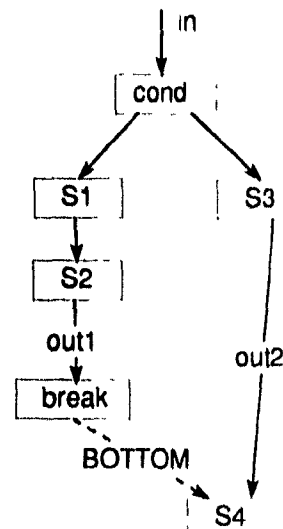


Figure 3.20: An example of if statement with a break statement.

and in the **continue-list** in our analysis. Let us first consider the **continue-list**. Recall that a **continue** statement takes the program control back to the beginning of the corresponding loop. This means that we should merge the following three sets to get a new approximation (each of these sets can be a new input to the loop):

- the previous approximation (the first approximation for **while** statement is the input set, for **for** statement is the result after processing the initial statement, and for **do-while** is the result after the first process to its body.).
- the output obtained from processing of the loop-body with the previous approximation as the input.
- all sets of information stored in the **continue-list**. Note that each of these sets is a potential input to the loop and corresponds to a path in the body that was terminated by a **continue** statement.

This process is repeated until a fixed-point is reached.

Unlike the **continue-list**, the **break-list** does not participate in the fixed-point calculation. In the case of the **break-list**, we merge the final approximation after reaching a fixed-point (for loops) with each set of the information contained in the **break-list**. The reason behind this approach is that each set of information in the **break-list** is a potential output of the statement.

The corresponding algorithms for **break** and **continue** (denoted as 'process\_break' and 'process\_continue' in Figure 3.15) are:

```

process_break(in) {
    /* break_lst field contain the merge of all the information
       * reaching to the break statements related to a loop */
    in.break_lst = merge_info(in.break_lst, in) ;
    return(BOTTOM) ;
}

process_cont(in) {
    /* cont_lst field contain the merge of all the information
       * reaching to the continue statements related to a loop */
    in.cont_lst = merge_info(in.cont_lst, in) ;
    return(BOTTOM) ;
}

```

Figures 3.21 to 3.24 illustrate the algorithm together with a graph representation of the **while**, **do-while**, **for**, and **switch** statements containing **break** and **continue**.

### 3.7 Summary

In this chapter, we discussed the intraprocedural points-to analysis for scalar variables. The analysis rules for basic statements and compositional control statements were described in a structured and compositional way. We also described a systematic strategy to handle the analysis of control statements with **break** and **continue** features embedded. The algorithm devised for points-to analysis can be used as a model to design structured algorithms for other flow analyses. In the next chapter, the interprocedural aspects of calculating points-to information are discussed.

```

process_while(cond, body, in) {
{
    last_in = in ;
    out = points_to(body, in) ;
    /* get the continue list */
    cont_lst = in.cont_lst ;
    out1 = merge_info(out, cont_lst) ;
    in = merge_info(in, out1) ;
} while (last_in != in) ;

break_lst = in.break_lst ;
result = merge_info(in, break_lst) ;
return(result) ;
}

```

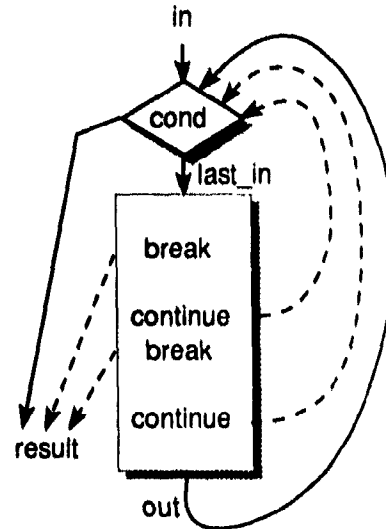


Figure 3.21: Algorithm of the **while** statement in the presence of **break** and **continue** statements.

```

process_do(cond, body, in) {
/* the body is processed at least once */
out = points_to(body, in) ;
/* get the continue list */
cont_lst = in.cont_lst ;
in = merge_info(out, cont_lst) ;
{
    last_in = in ;
    out = points_to(body, in) ;
    /* get the continue list */
    cont_lst = in.cont_lst ;
    out1 = merge_info(out, cont_lst) ;
    in = merge_info(in, out1) ;
} while (last_in != in) ;

break_lst = in.break_lst ;
result = merge_info(in, break_lst) ;
return(result) ;
}

```

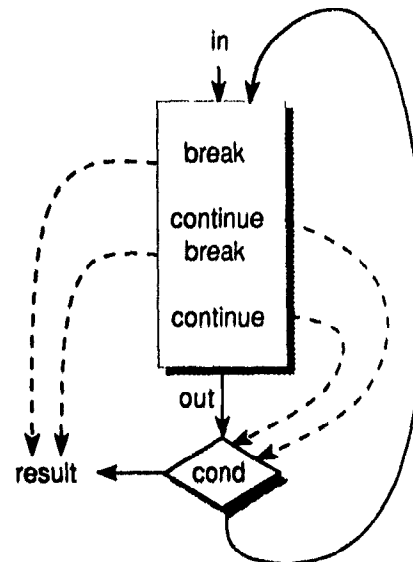


Figure 3.22: Algorithm of the **do-while** statement in the presence of **break** and **continue** statements.

```

process_for(init_stmt, cond, iter_stmt, body, in) {
    in = points_to(init_stmt, in) ;
    {
        last_in = in ;
        out1 = points_to(body, in) ;
        /* get the continue list */
        cont_lst = in.cont_lst ;
        out2 = merge_info(out1, cont_lst) ;
        out3 = points_to(iter_stmt, out2) ;
        in = merge_info(in, out3) ;
    } while (last_in != in) ;

    break_lst = in.break_lst ;
    result = merge_info(in, break_lst) ;
    return(result) ;
}

```

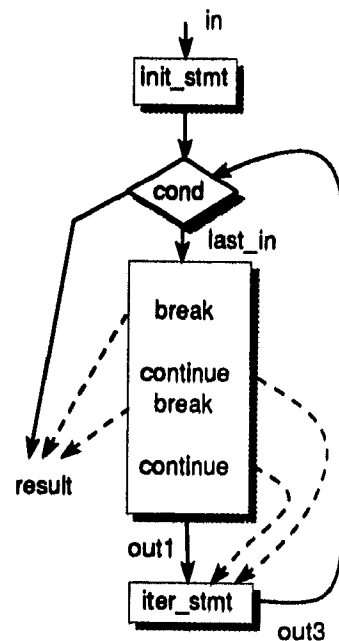


Figure 3.23: Algorithm of the **for** statement in the presence of **break** and **continue** statements.

```

process_switch(expr, caselist, in) {
    for each case in caselist do
        tmp = points_to(case_body, in) ;

        break_lst = in.break_lst ;
        result = break_lst ;
        return(result) ;
}

```

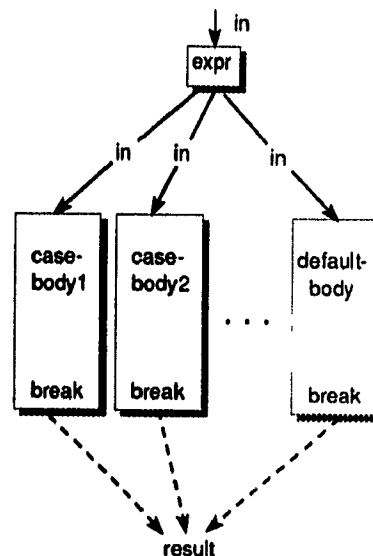


Figure 3.24: Algorithm of the **switch** statement in the presence of **break** and **continue** statements.

## Chapter 4

# Interprocedural Points-to Analysis for Scalar Variables

This chapter is concerned with the interprocedural abstract stack analysis which is the process of collecting the points-to information in the presence of function calls. This is more complicated than the intraprocedural abstract stack analysis discussed in the previous chapter because it takes care of: (i) the renaming process between formal parameters and actual parameters of pointer-type, (ii) the effect of function calls on the points-to information involving global variables, (iii) the points-to information for variables of pointer-type which are returned by a function, and (iv) the effect of recursive function calls.

To perform the interprocedural analysis, we make use of a special representation for the interprocedural structure, namely the *invocation graph*. We first explain the construction of the invocation graph in Section 4.1. Then, we discuss the interprocedural analysis. Since the methods for recursive function calls are more complicated, we first consider the interprocedural analysis without the presence of recursive function calls in Section 4.2. Then, the recursive function calls are included in Section 4.3.

### 4.1 Invocation Graph

In this section, we explain the construction of the invocation graph in the presence of both recursive and non-recursive function calls. The construction of the invocation graph is independent of points-to analysis and can be used by any other analysis. Before presenting the details, let us first explain the following terminology: If a function *f* calls another function *g*, *f* is called the *caller* and *g* is called the *callee*.

An invocation graph captures the activation order and structure of function calls in a program. Our invocation graph is made up of: (i) a set of nodes representing function call instances, and, (ii) a set of edges specifying the callee-caller relationships between functions.

The following are three major reasons for using an invocation graph:

1. Our invocation graph is specially useful in the presence of recursive function calls. This is discussed in Section 4.3.
2. By using an invocation graph one can save the analysis information of each call separately. This can be used for the optimization purposes. Readers are referred to the section "future work" in Chapter 10 for further information on this subject.
3. The invocation graph can be used in conjunction with function-pointers. Readers are referred to the Section 8.2.3 for further information on this subject.

In the case of non-recursive function calls, the invocation graph is in the form of an ordinary tree while for recursive function calls it is a tree with some implicit back-edges. We first explain the non-recursive invocation graph and then extend it to include recursive calls.

The nodes in a non-recursive invocation graph are called *ordinary nodes* and the edges are called *calling arcs*. To build a non-recursive invocation graph, for each function, we collect the list of all the functions who gets called by that function. Then, we recursively construct the invocation graph starting from the `main` by linking the caller and the callee functions. Consider the example shown in Figure 4.1. Figure 4.1(b) shows the list of functions called by each function. For example, function `main` calls three functions, `f()`, `g()`, and `h()`. The construction of the invocation graph can be understood by following the steps shown in Figure 4.1(c).

Next, we construct an invocation graph for a program with a recursive function call using the same steps as explained before. Consider the example given in Figure 4.2. Using the previous method, we obtain an infinite call to function `f`. This is shown in Figure 4.2(c).

To solve this problem, the number of function calls has to be approximated. This is achieved by cutting down the infinite number of the calls to the function `f()` into two calls: (i) the first appearance of the function call in the invocation graph, namely *recursive node* denoted as `f-R`, and (ii) the second appearance of the function call in the invocation graph, namely *approximate node* denoted as `f-A`. To set up the correspondence between an approximate node and the corresponding recursive node,



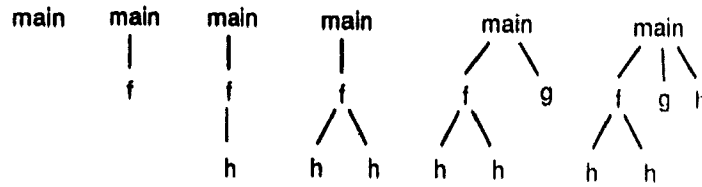
```

main ( )
{
    f ( ) ;
    g ( ) ;
    h ( ) ;
}
f ( )
{
    h ( ) ;
    h ( ) ;
}
g ( )
{
    h ( ) ;
}

```

Func	Func-list
main ( )	f -> g -> h
f ( )	h -> h
g ( )	
h ( )	

(b) 'Func' is the function name and 'Func-list' is the list of functions who get called by 'Func'



(a) C program

(c) The steps of generation of invocatin graph from left to right

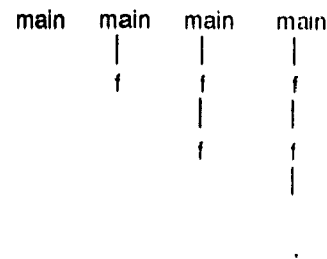
Figure 4.1: An example for the construction of the invocation graph.

```

main ( )
{
    f ( ) ;
}
f ( )
{
    if ( ... )
        f ( ) ;
}

```

Func	Func-list
main ( )	f
f ( )	f



(a) C program

(b) 'Func' is the function name and 'Func-list' is the list of functions who get called by 'Func'

(c) The steps of generation of invocation graph from left to right

Figure 4.2: An example for the construction of the recursive invocation graph.

a link is established from the approximate node to the recursive node in the invocation graph. Figure 4.3 shows the extended invocation graph for the example of Figure 4.2.

Now that the idea is clear, we give the formal definition and a precise algorithm. A call is *recursive* if it has already occurred once in the call-chain from the root. In the presence of a recursive function call **f**, the following steps are taken:

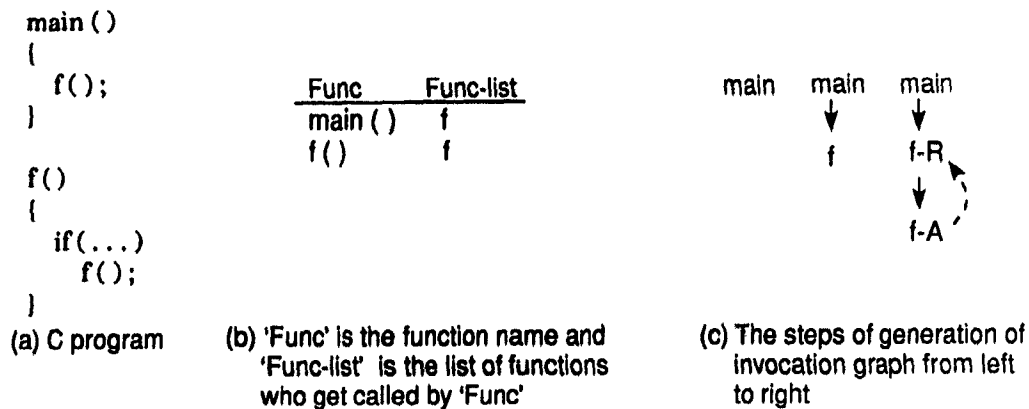


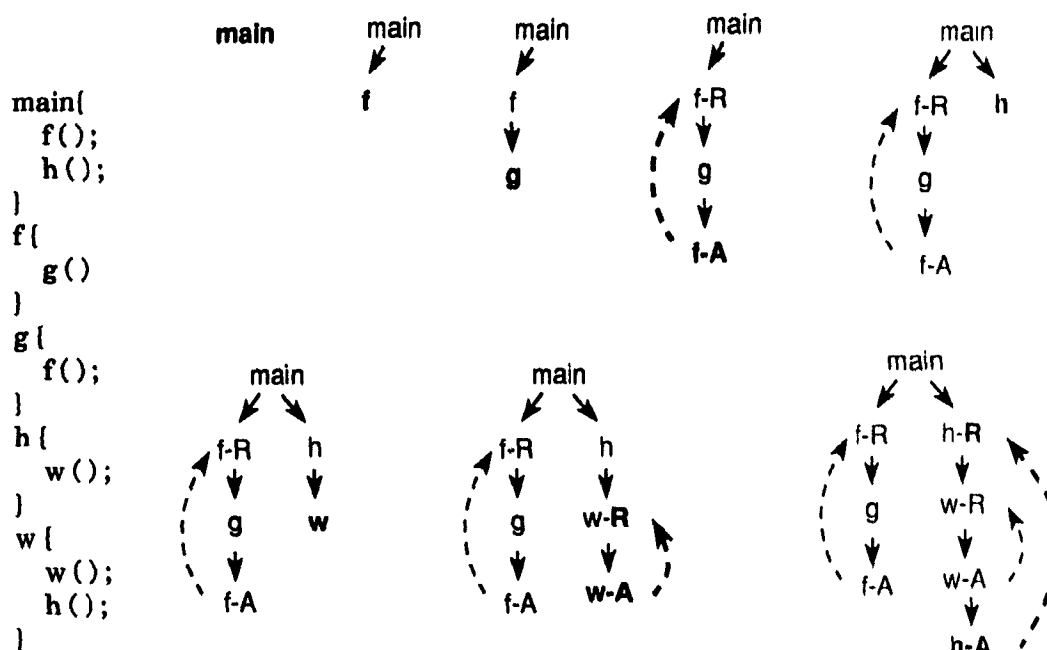
Figure 4.3: An example of the construction of the recursive invocation graph using our method.

1. Change the first occurrence of **f** from ordinary to recursive node **f-R**.
2. Create an approximate node for the recursive call **f**, denoted as **f-A**, and connect it to the current place (the caller's node in the invocation graph) by a calling arc.
3. Connect the approximate node **f-A** and its corresponding recursive node **f-R** by an *approximate arc*.

In the following, we explain the construction of the invocation graph in the presence of mutual recursion, using the example shown in Figure 4.4(a). Figure 4.4(b) shows all the steps of this construction. The new modifications at each step are shown using bold face notation. Since function **main** calls **f**, we create two ordinary nodes for **main** and **f** and connect them by a calling arc. When **f** calls **g**, we create another ordinary node for **g** and connect it to **f** by a calling arc. Now **g** calls **f** recursively. Since **f** has already occurred once in the call-chain from the root, we do the following: (i) change the first occurrence of **f** to a recursive node denoted as **f-R**, (ii) create an approximate node for **f** denoted as **f-A** and connect **g** and **f-A** by a calling arc, and (iii) connect the approximate node **f-A** and its corresponding recursive node **f-R** by an approximate arc. Approximate arcs are shown by dotted lines in Figure 4.4. Similarly, functions **h** and **w** call themselves recursively. So, we convert the first occurrence of **h** and **w** to **h-R** and **w-R**, create new nodes **h-A** and **w-A**, and connect them by approximate arcs to **h-R** and **w-R**, respectively.

The depth-first algorithm for building invocation graph is given in Appendix B.

Each node in the invocation graph (except **main**) corresponds to a call-site in the



(a) The C program (b) All the steps while building the invocation graph for the given program

Figure 4.4: An example for construction of the invocation graph in the presence of mutual recursion.

program and each function call in the program corresponds to a unique path in the invocation graph. This fact is shown in the example of Figure 4.5. This example pose the question that how the same function call `g()` (in the body of `f()`) points to two different nodes in the invocation graph. In order to maintain a one to one relationship between the paths of invocation graph and the call sites, one should keep track of (i) the invocation graph node related to corresponding caller (for example the `f` nodes from the invocation graph) and (ii) the corresponding call number (for example function `g` is the first function call in the body of function `f`, therefore the call number in one).

In the subsequent section, we explain how the invocation graph is used in interprocedural analysis.

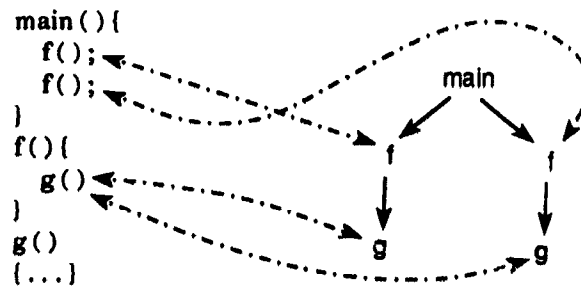


Figure 4.5: Relationship between program and invocation graph.

## 4.2 Interprocedural Analysis for Non-recursive Function Calls

In this section, we discuss interprocedural analysis in the presence of non-recursive function calls. Interprocedural analysis captures points-to information across functions affected through both function parameters and global variables.

We first give a motivating example to demonstrate the need for the interprocedural analysis. Figure 4.6 represents a program that indirectly swaps two variables. The memory representation of `main` before calling function `indirect_swap`, the memory representation of `indirect_swap` before it is processed, and the memory representation of `main` after the call to `indirect_swap` is processed, are shown in Figure 4.6(a), (b), and (c), respectively. The variable `y1` which points to the location `z1` before the call to the function `indirect_swap` in `main`, points to the location `z2` after this call is completed. This is due to the fact that the memory location `y1` is accessible through the pointer dereference `*a` (although `y1` is not in the scope of the function `indirect_swap`). The points-to relationship for the variable `y2` changes in a similar way.

Two important observations can be made here:

1. A function call can considerably affect the points-to relationships holding in the caller.
2. The program variables, whose points-to information can be modified by a function call, need not lie in the scope of the callee.

The above two possibilities make interprocedural analysis necessary for collecting accurate points-to information.

```

int *m ;
main()
{
    int **x1, *y1, z1 ;
    int **x2, *y2, z2 ;

    x1 = &y1 ;
    y1 = &z1 ;

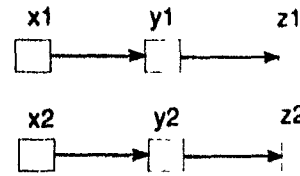
    x2 = &y2 ;
    y2 = &z2 ;

    indirect_swap(x1,x2) ;
}

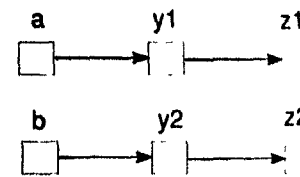
void indirect_swap(a, b)
int **a, **b ;
{
    int *temp ;

    temp = *a ;
    *a = *b ;
    *b = temp ;
}

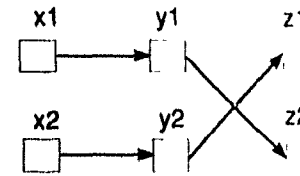
```



(a) The memory representation of main(), before function call indirect\_swap().



(b) The memory representation of indirect\_swap(), before the execution of the function body



(c) The memory representation of main(), after returning from the call to indirect\_swap().

Figure 4.6: A motivating example for interprocedural analysis.

### A conservative approach:

In the absence of interprocedural analysis, safe conservative assumptions need to be made about the effects of a function call on the points-to information in the caller. The minimum requirements for any such approximation is to assume that the following types of variables can point to any variable in the scope of the callee:

1. Pointer variables which are global in scope.
2. Pointer variables whose address is passed as a parameter to the callee.
3. Pointer variables accessible through one or more levels of dereference of any variable of the above two types.

As type casting is permitted in C, we have to assume that these pointer variables can point to any variable, irrespective of its type, in the scope of the callee.

Using these assumptions, the memory representation after the call to the function `indirect_swap` is estimated as shown in Figure 4.7. The relationships from/to `m`

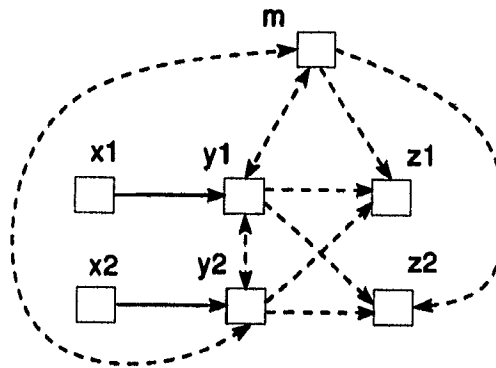


Figure 4.7: The memory representation without the interprocedural analysis based on a conservative assumption instead.

are caused by the fact that `m` is a global variable and therefore can be effected by any function call. The possibly-points-to relationship between the pointer variables of the same level, e.g.  $(y1, y2, P)$ , can be caused by type casting. Since the address of variables `x1` and `x2` are not accessible in function `indirect_swap`, their points-to information does not change.

We conclude that the conservative approach introduces substantial imprecision in the points-to information. This imprecision accumulates with frequently occurring function calls in C-programs. Thus, the need for interprocedural analysis is evident.

#### Our approach using the invocation graph:

We have already discussed our representation for the invocation graph which captures the interprocedural structure of the program. The invocation graph is a basic requirement for interprocedural analysis as it makes both the interprocedural program paths and the nature of function calls, explicit. The usage of invocation graph in our analysis is explained in Section 4.2.1.

For interprocedural points-to analysis, we need to propagate points-to information from a call-site to the entry of the corresponding function. As this propagation progresses, we modify the points-to information to take care of the following:

1. The renaming of variables caused by parameter passing.

2. The possibility of accessing a variable inside the callee which is not in the scope of callee (as shown in the indirect swap example).

Next, the body of the function is analyzed using this points-to information. The information computed during this analysis is returned to the call site. Due to the above-mentioned factors, this information has to be modified again. The map and unmap processes, explained in Section 4.2.2, are designed to perform proper propagation of points-to information from the caller to the callee and vice versa.

After simplification, the occurrence of a function call is limited to one of the following cases when each argument,  $\text{arg}_i$ , is a scalar variable or a constant:

1.  $f(\text{arg}_1, \text{arg}_2, \dots, \text{arg}_n)$
2.  $a = f(\text{arg}_1, \text{arg}_2, \dots, \text{arg}_n)$  and `return(var)`

In the following sections, we discuss the first case in detail. The second case, which is obtained by adding some slight modifications to the first case, will be discussed in Section 4.2.3.

#### 4.2.1 Relationship Between Invocation Graph and Interprocedural Analysis

As mentioned before, our interprocedural analysis follows the sequence of function calls. This means that for each function call, analysis proceeds from the call site to the analysis of the body of the callee and then returns to the call-site.

The relationship between our analysis with invocation graph is explained by studying the example given in Figure 4.8. Analysis of the program starts from the main function. For each function, the intraprocedural analysis is done until we reach a function call (in this case  $f()$ ). Then, the analysis proceeds via the invocation graph. If the corresponding invocation graph node is an ordinary node (other kind of nodes will be discussed in Section 4.3), the body of the callee ( $f$ ) is analyzed and the result is returned through the invocation graph to the call-site upon completion. After that, the analysis continues in the body of the caller (`main`) until reaching another call ( $f()$ ). Analysis continues in this fashion until all statements of caller (`main`) have been analyzed. The movement to/from the invocation graph is shown as the labels of the links between the invocation graph and the program in Figure 4.8

Using this method, we can be sure of the following points:

1. The points-to information coming from different call-sites will not be used at the same time to collect new points-to information.

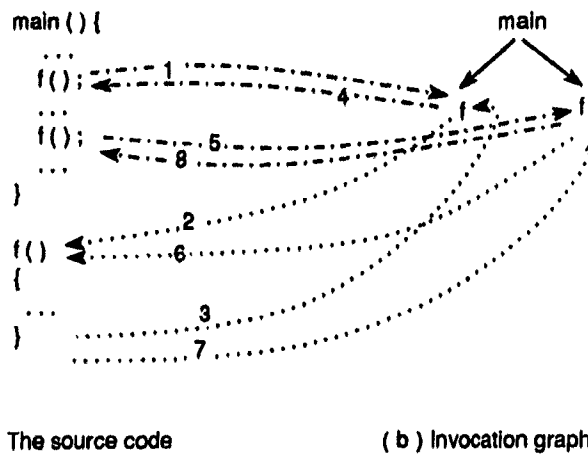


Figure 4.8: Relationship between program and invocation graph (the numbers written on the edges show the order of the process).

2. The points-to information always returns to the appropriate unique call-site.

#### 4.2.2 Map and Unmap Processes

We characterize the interprocedural analysis by the following steps:

- Map process: is a process that compares all the actual and formal parameters and sets all the possible points-to information for formal parameters. It does the same thing for the global variables.
- Function process: is the intraprocedural analysis for the body of the callee. The input to this process is the abstract stack as created by the map process.
- Unmap process: is a process which takes the resulting abstract stack from the function process and unmaps the variable names - changing the names in the callee back to the names in the caller.

Figure 4.9 shows a graph representation of the map and unmap processes.

The interprocedural points-to algorithm is represented in the following:



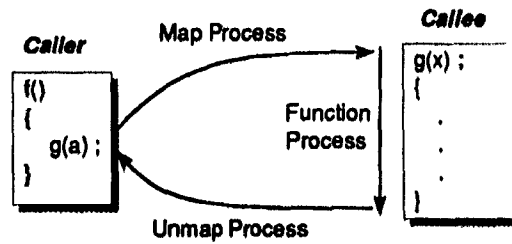


Figure 4.9: The map and unmap process.

```

/* Given a basic statement and input information, return the output
 * information (input.in is the points_to relationships) */
points_to (S,input) =
  if basic_statement(S) then
    /* S is basic statement */
    return(basic_points_to(S,input));
  else if control_statement(S) then
    /* S is a compositional control statement */
    return(control_points_to(S,input));
  else if func_call(S) then
    /* S is a call expression statement */
    return(func_points_to(S,input));
  else
    return(input) ;

```

The algorithm for 'basic\_points\_to' and 'control\_points\_to' was given in Chapter 3. The algorithm for 'func\_points\_to' in the absence of recursive function calls is given in the following (the algorithms for 'map-process' and 'unmap-process' which are used here will be explained later in this chapter). The algorithm of 'func\_points\_to' in the presence of recursive function calls is given in Section 4.3.

```

/*
* Function name : func_points_to
* Purpose : find out the points-to information after
* processing a non recursive function call, using
* the input information to the call.
* Parameters : call_expr_node -- the call node in caller
* in_data -- the input information to
* 'call_expr_node' before it is processed.
* Note that the field 'in' represents the
* points-to relationships.
* Return : out_data -- the output information of caller
* after processing function call
*/
func_points_to( call_expr_node, in_data)
{
    /* get the related invocation graph node to the callee by using the
    * invocation graph node of caller ('in_data.cur_ig_node') and
    * related call number of callee ('call_expr_node.call_expr_num').
    * For example, if 'f(a)' is the third function call in the body of 'g',
    * the related invocation graph node to callee 'f(a)' is the third child
    * of the related invocation graph node of caller 'g'. */
    ig_node = get_related_ig_node( in_data.cur_ig_node,
                                   call_expr_node.call_expr_num) ;

    /* get the function node of the callee from the 'call_expr_node' */
    func_node = get_func_node( call_expr_node) ;

    /* get the list of arguments (actual parameters) */
    arg_lst = get_arg_lst( call_expr_node) ;

    /* 'map_process' maps the points-to information ('in_data.in'), returns
    * the mapped points-to information ('func_in_data.in') and the map
    * information ('map_info') */
    [map_info, func_in_data.in] = map_process( func_node, arg_lst, in_data.in) ;

    /* save the map information and the entry points-to relationships in the
    * invocation graph node related to the function call. */
    save_in_info( ig_node, map_info, func_in_data.in) ;

    /* get the body of the callee */
    func_body = get_func_body( func_node) ;

    /* if the invocation graph node is an ordinary node (non-recursive), analyze
    * the body of callee and return output information in 'func_out_data' */
    if (is_ordinary_node(ig_node))
        func_out_data = points_to( func_body, func_in_data) ;
}

```

```

/* save the result of the points-to analysis of callee in the related
 * invocation graph node */
save_out_info( ig_node, func_out_data.in) ;

/* the unmap process gets the points-to information of caller
 * and callee and returns the new set of points-to information
 * for caller ('.in' is the field related to points-to info).*/
out_data.in = unmap_process( in_data, func_out_data.in, map_info) ,

return( out_data) ;
}

```

As mentioned before, the following two cases complicate the map and the unmap processes.

1. The renaming process between the actual and formal parameters. This is explained in the next two sections.
2. The accessibility of the variables which are not in the scope of the callee (through indirect reference). To solve this problem, we introduce the concept of *Invisible Variables*<sup>1</sup>. These are names used for the variables that are not in the scope of a function but are accessible through indirect reference. The invisible variables of **x** with type **int\*\*** are **1\_x** and **2\_x** with types **int\*** and **int**, respectively.

The following example explains the use of invisible variables.

```

main(){
    int *a, b, *c, d ; /* stmt 1 */
    a = &b ;           /* stmt 2 */
    f(a) ;             /* stmt 3 */
    c = &d ;           /* stmt 4 */
    f(c) ;             /* stmt 5 */
}                      /* stmt 6 */
f(x)                  /* stmt 7 */
int *x ;              /* stmt 8 */
{ }

```

After processing statement 2, we get (a,b,D). As the result of the call to function **f** at statement 3, **x** gets a copy of the contents of **a**. This means that **x** should point to

<sup>1</sup> A similar notation of non-visible variables is given in [LR92].

the same location as  $a$  is pointing to, namely  $b$ . Even though  $b$  is not in the scope of function  $f$ , the memory location represented by  $b$  is still accessible in the function  $f$  through the indirect reference  $*x$ . Since we deal with locations of abstract stack, we need a name corresponding to the location  $b$ . There are two solutions to this problem:

- (i) Having a location in the callee for each possible access to a variable even if the variable is not in the scope of the callee. For the above example, we should have abstract stack locations for variables  $b$  and  $d$  because they will be used by the calls to  $f$  at statements 3 and 5.
- (ii) Introducing invisible variables and using them instead of the variables which are not available in the scope of callee. In the above example,  $x$  has only one invisible variable namely  $1_x$  with type `int`. At the call in statement 3, the invisible variable  $1_x$  stands for the local variable  $b$  and leads to the relationship  $(x, 1_x, D)$  as the result of the map process. At the call in statement 5, the invisible variable  $1_x$  stands for the local variable  $d$  and resolves the relationship  $(x, 1_x, D)$  as the result of the map process.

The following are the advantages of using the invisible variables instead of the local variables of the caller in the abstract stack:

1. Adding the local variables to the abstract stack location of the callee increases the size of the abstract stack. In the above example, without using invisible variables, we would need two locations for  $b$  and  $d$  in the abstract stack of  $f$ . Using the invisible variable  $1_x$  is sufficient to handle both of these cases (in the first call  $1_x$  stands for  $b$  and in the second call it stands for  $d$ ). This results in a more compact representation for the points-to information.
2. The points-to information of each function call can be reused by keeping track of the input-output sets of points-to information of each function. If an input set has already occurred and an output is computed for it, in the second appearance of the same input, we can use the already computed output. The input-output information of each call is saved in its related invocation graph node. In the above example, the input to the function  $f$  in both cases is the same. Consequently, there is no need to process the second call to  $f$ , if the output of the first call is already available. On the other hand, as we have two different locations for  $b$  and  $d$  in the abstract stack for  $f$ , it becomes necessary to process both function calls to  $f$ . Although this could have been possible if a function is called by the same parameters (e.g. if the second call to  $f$  was  $f(a)$ ), but it can be done more often while using invisible variables.

3. Adding the local variables can result in a problem in the case of recursion. This happens because the number of iterations in a recursive function is not known. In this case, we select a finite number of invisible variables using the type of arguments and global variables.

To make sure that each accessible variable has a location in the abstract stack, we generate names for all the possible locations that each indirection of a formal parameter or global variable may need to access.

A set of relationships between local variables in the caller and invisible variables in the callee is kept for the unmap purposes. This set is called `map_info`. In the presence of possibly-points-to relationship, an invisible variable may stand for more than one variable. The details will be explained in subsequent sections.

In the following sections, we describe: (i) a general algorithm for map process, (ii) a general algorithm for unmap process, and (iii) our accurate algorithm for the map and unmap processes.

### Map Process

The idea of the map process was explained in the previous section. In general, one can devise different algorithms for the map process. In this section, we present a general, and straight-forward algorithm for the map process. Later, this is extended to a more accurate method.

Consider the case that the function `f` is called as `f( arg1, arg2, ..., argn )` and is defined as `f( param1, param2, ..., paramn )`. As in the C-language, parameters are always passed by value, to get the equivalent abstract stack links for each formal parameter `parami`, we consider the parameters as:

```
param1 = arg1 ;  
param2 = arg2 ;  
...  
paramn = argn ;
```

The rule determining the points-to information is similar to the case for basic statements which was described in Chapter 3. This rule specifies that `parami` should point to the same location as `argi` is pointing to. In the present case, we have two additional features for this rule which are as follows:

- (i) For multi-level pointers, the rule extends the points-to relationship to each of the corresponding lower level of dereference that is of pointer type. For example, if `argi` is of type `int**`, the rule should also apply to `*parami = *argi`.

- (ii) The invisible variables are used instead of the variables which are not in the scope of callee.

Since a global variable might point to a local variable of caller which is an invisible variable for callee, the map process should be applied to global variables as well. This is achieved by applying the assignment `vari = vari` to all the global variable `vari` in the same way as was explained for the function parameter. In this assignment, the global variable `vari` appearing in the lhs (left hand side) is related to the scope of the callee and the global variable `vari` appearing in rhs (right hand side) is related to the scope of the caller (each function has its own abstract stack).

In the case that an actual parameter or a global variable points to a location which is not in the scope of callee, we use the related invisible variable and set its related location in a set called `map_info` (this set is later used by the unmap process). If the location is already assigned to an invisible variable, we just use that invisible variable. The reason behind not redefining an invisible variable is to avoid a situation where one real stack location is used for two or more different abstract stack locations.

The general algorithm for the map process is as follows:

```

/*
 * Function name : map_process
 * Purpose : finding the points-to information of callee,
 *           and the corresponding map information using
 *           the points-to information of caller.
 * Parameters : func_node -- the function declaration node of callee
 *             arg_lst -- argument list
 *             caller_in -- the points-to information of caller
 * Return : map_info -- a set of relationship between invisible
 *                    variables of callee and local variables
 *                    of caller.
 *           callee_in -- the points-to information entering to callee
 */
map_process( func_node, arg_lst, caller_in)
{
    /* initialization */
    map_info = { } ;
    callee_in = { } ;
    param_lst = get_param_lst( func_node) ;

    /* doing the map process for each parameter */
    for each 'param_i' in 'param_lst' and 'arg_i' in 'arg_lst' do
        /* 'callee_in' and 'map_info' will be updated. The argument
         * corresponding to 1 is the depth of the pointer type. For
         * the first call, depth is 1. */
        [map_info, callee_in] =
        map_func( param_i, arg_i, callee_in, caller_in, 1, map_info) .

    /* doing the map process for each global variable */
    for each global variable 'var_i' do
        /* 'callee_in' and 'map_info' will be updated. The argument
         * corresponding to 1 is the depth of the pointer type. For
         * the first call, depth is 1. */
        [map_info, callee_in] =
        map_func( var_i, var_i, callee_in, caller_in, 1, map_info) .

    return( [map_info, callee_in] ) ;
}

```

```

/*
* Function name : map_func
* Purpose : assigning the points-to information of caller_var
*           to callee_var (considering the invisible
*           variables). This function is recursively called
*           to assign the points-to information of all the
*           indirectly accessible variables through caller_var.
* Parameters : callee_var -- the variable in callee
*              caller_var -- the related variable in caller
*              callee_in -- points-to information of callee
*              caller_in -- points-to information of caller
*              depth -- the depth of current invisible variable
*                      (for the first call, it is 1).
*              map_info -- the set of map information
* Return : map_info -- the updated set of map information
*          callee_in -- the updated set of points-to information
*                  of callee
*/
map_func( callee_var, caller_var, callee_in, caller_in, depth, map_info)
{
    /* if 'callee_var' is not of pointer-type, there is no need of
    * points-to analysis (no extra points-to relationship can be
    * generated) */
    if !(is_pointer_type(callee_var))
        return([map_info, callee_in]) ;

    for each 'x' such that a relationship '(caller_var, x, rel)' exists do
    /* 'rel' can be either definitely-points-to or possibly-points-to */
    {
        if (is_in_callee_scope(x))
        {
            /* add the same type of relationship to 'callee_in' */
            callee_in = callee_in  $\cup$  {(callee_var, x, rel)} ;

            /* recursively check all the relationship of the variables
            * which can be derived from 'caller_var'. In this case,
            * it is all the variables that 'x' points to.
            * As 'x' is one level deeper, depth is increased by one */
            [map_info, callee_in] =
            map_func( x, x, callee_in, caller_in, depth+1) ;
        }
    }
}

```



```

else{ /* variable 'x' is not in the scope of callee */

    /* look at 'map_info' set to check if an invisible variable
     * is already assigned to the variable 'x' */
    if (exist_invisible_for( x, map_info))
        /* if an invisible variable is already assigned to variable
         * 'x', get that variable and use the same name */
        x_invisible = get_invisible_var( x, map_info) ;

    else{ /* an invisible variable does not exist for 'x'*/
        /* get an invisible variable for 'x' using 'depth'
         * and 'callee_var'. If the name of variable 'callee_var'
         * is 'data' and 'depth' is 1, the invisible variable
         * would be '1_data', for 'depth' 2, the invisible variable
         * would be '2_data', and so on. */
        x_invisible = define_invisible( callee_var, depth) ;

        /* set the equivalency of 'x_invisible' and 'x' in the
         * 'map_info'. This information is used by the unmap process/
        add_map_info( x_invisible, x, map_info) ;
    }

    /* add the relationship 'rel' between 'callee_var' and the
     * invisible variable 'x_invisible' in 'callee_in'. */
    callee_in = callee_in  $\cup$  {(callee_var, x_invisible, rel)} ;

    /* recursively check all the relationship of the variables
     * which can be derived from 'caller_var'. In this case,
     * it is all the variables that 'x' points to.
     * As 'x' is one level deeper, depth is increased by one */
    [map_info, callee_in] =
    map_func( x_invisible, x, callee_in, caller_in, depth+1) .
}
}
return([map_info, callee_in]) ;
}

```

In the following, we further explain the general map process using some examples which are devised to give a clear idea of the concept.

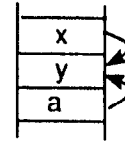
Consider the example given in Figure 4.10. At statement 1, before the process of function  $f$ , we get  $\{(a, y, D), (x, y, D)\}$  (refer to Figure 4.10(a)). We first apply the map process on the parameters. To do this, we add the points-to relationship resulting from  $m = a$  to the set of the points-to information of callee. This means that  $m$  should point to the same location as  $a$  is pointing to. As the result, we obtain

```

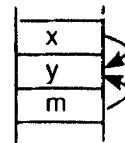
int *x, y ;
main()
{
    int *a ;
    a = &y ;
    x = &y ;
    f(a) ;    /* stmt1 */
}

f(m)
int *m ;      /* stmt2 */
{
    ...
}

```



(a) Abstract stack of main ( ) at statement 1



(b) Abstract stack of f ( ) at statement 2

Figure 4.10: A simple example of map process.

the relationship  $(m, y, D)$ . Next, we perform the map process on global variables. This results in the relationship  $(x, y, D)$  for callee. The final result of map process is shown in Figure 4.10(b).

Figure 4.11 shows another example that makes use of the invisible variables. The main program under consideration and its simplified version (obtained by passing through the simplifier program) are shown in parts (a) and (b) of this figure. As already mentioned in Chapter 2, an address operand can never appear as a parameter in the SIMPLE AST. This fact is shown in statement 1 of Figure 4.11(b). In this case, the address operand will be moved out by defining a new variable (in this case temp0) and using it as a replacement. At statement 1, before the process of function  $f$ , we get  $\{(x, c, D), (a, b, D), (temp0, b, D)\}$ . The corresponding abstract stack representation is shown in Figure 4.11(c). At this point, the map process is performed on function parameters and global variables. We first apply the map process to the two function parameters,  $m$  and  $n$ . This is described in the following:

1. We should add the points-to information resolved from  $m = a$  to the set of the points-to information of callee ( $f$ ). This means that  $m$  should point to the same location as  $a$  is pointing to. This results in the relationship  $(m, b, D)$ . Since variable  $b$  is not in the scope of function  $f$ , we do not want to use the name  $b$  in the analysis of  $f$ . Rather, at this point, we use the invisible variable  $1\_m$  instead of  $b$ . We also note in the `map_info` set that `1_m` and  $b$  are equivalent. As the final result of this step, we get the relationship  $(m, 1\_m, D)$ . Since `1_m` is not of

```

int **x ;
main()
{
    int *a, b,
        *c, d ;

    a = &b ;
    x = &c ;
    c = &d ;
    f(a, &b) ;
}

f(m, n)
int *m, *n ;
{
}

int **x ;
main()
{
    int *a, b,
        *c, d ;
    int *temp0 ;

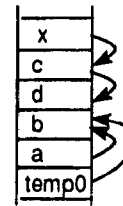
    a = &b ;
    x = &c ;
    c = &d ;
    temp0 = &b
    f(a, temp0) ;/*stmt 1*/
}

f(m, n)
int *m, *n ;    /*stmt 2*/
{
}

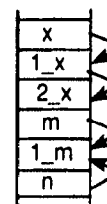
```

(a) A C-program

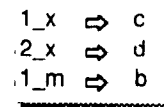
(b) The simplified version



(c) Abstract stack of `main()` at statement 1



(d) Abstract stack of `f()` at statement 2



(e) The map information (map\_info)

Figure 4.11: An example of map process where invisible variables are used

pointer-type, this step is completed.

2. We should add the same points-to information resolved from `n = temp0`. This means that `n` should point to the same location as `temp0` is pointing to. This results in the relationship  $(n, b, D)$ . Once again, we want to express this relationship in terms of the names in the scope of function `f`. However, in this case, as it is not the first occurrence of `b`, the invisible variable `1_m` already stands for variable `b`. Therefore, we can use `1_m` instead of `b` (there is no need to define another variable). As the final result, we get the relationship  $(n, 1_m, D)$ . Since `1_m` is not of pointer-type, this step is completed.

Now, we apply the map process to the global variable `x` resulting in  $(x, c, D)$ . In this case, `c` is not in the scope of `f`, and, consequently, it is the first time that `c` is used in

the map process. Due to this reason, we use another invisible variable  $1\_x$  and add it to the set of `map_info`. This results in  $(x, 1\_x, D)$ . Since  $1\_x$  is of pointer type (`int *`) the map process continues to a lower level. Because  $1\_x$  is equivalent to  $c$ ,  $1\_x$  points to the same location as  $c$  is pointing to. The variable  $c$  definitely points to  $d$  but  $d$  is not in the scope of function  $f$  and also is not already assigned to an invisible variable. In this case, another invisible variable has to be used, namely  $2\_x$ . The equivalence of  $2\_x$  and  $d$  is added to `map_info`. This resolves the relationship  $(1\_x, 2\_x, D)$ . As  $2\_x$  is not of pointer-type, the process is completed at this point. Figure 4.11(d) shows the abstract stack after the completion of map process and Figure 4.11(e) shows the corresponding `map_info` set.

### Unmap Process

In this section, we explain the general unmap process and the use of invisible variables. In the next section, we extend the unmap process to a more accurate method.

As mentioned before, when the map process is completed, the body of the callee will be processed (intraprocedural analysis). After that, the unmap process returns all the changes to the caller. In other words, the unmap process updates the abstract stack of caller according to the changes occurring in the process of callee. The map information (`map_info`) collected from map process is used for this updating purpose. The relationship between local variables of caller and invisible variables of callee are kept in the `map_info` set.

Assume that `caller_input` is the set of points-to information of caller  $f$  just before the function call to  $g$ , and `callee_output` is the set of points-to information of callee  $g$  after the process of  $g$  is completed. We want to apply the unmap process to the two sets `caller_input` and `callee_output` and thereby compute the caller output set which is the points-to information of  $f$  after the process of callee  $g$  is completed. Following is the general algorithm of unmap process:

```

unmap_process( caller_input, callee_output, map_info)
{
    kill = {(x, y, rel) : (x, y, rel) ∈ caller_input ∧
              (x_map, y_map, rel) ∉ callee_output}

    gen = {(x, y, rel) : (x_map, y_map, rel) ∈ callee_output}

    caller_output = gen ∪ (caller_input - kill)

    return(caller_output)
}

```

where  $x/y$  is in the scope of caller ( $f$ ) and  $x\_map/y\_map$  is its related variable in the callee ( $g$ ). If  $x/y$  is in the scope of the callee ( $g$ ),  $x\_map/y\_map$  is the same as  $x/y$ . Otherwise, it can be found from the `map_info` set. Basically, the above procedure removes all the changed information from points-to set of caller and adds the new information to it. The following are some examples to clarify the corresponding algorithm.

Consider the example given in Figure 4.12. As it is shown in Figure 4.12(a), there

```

int *x, y ;
main()
{
    int *a ;
    f() ; /* stmt 1 */
    a = x ; /* stmt 3 */
} /* stmt 4 */
f()
{
    x = &y ;
} /* stmt 2 */

```

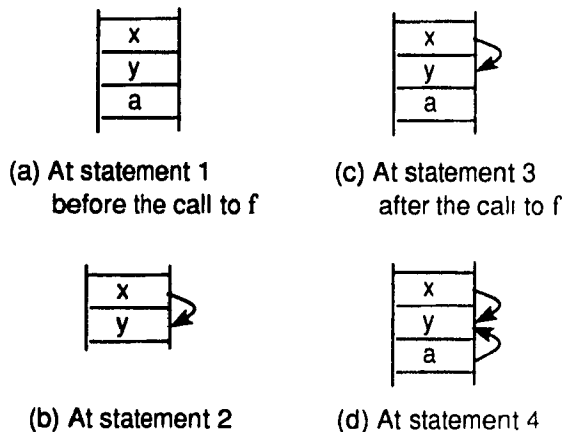


Figure 4.12: An example of unmap process.

is no points-to information that holds before the call to function  $f$  at statement 1. Due to this fact, the map process does not pass any information to the function  $f$

At statement 2, we get  $(x,y,D)$ . The corresponding stack representation is shown in Figure 4.12(b). Now is the time for the unmap process which is applied through computing the sets mentioned in the above algorithm. The involved sets are as follows:

```

caller_input = { }
callee_output = {(x,y,D)}
kill = { }
gen = {(x,y,D)}
caller_output = {(x,y,D)}

```

Note that since  $x$  and  $y$  in the set `caller_input` are in the scope of `main` then mapped variables do not change. As a result, `gen` set holds the relationship  $(x,y,D)$ . The result of unmap process is shown in Figure 4.12(c). Finally, after the execution of statement 3, we get  $(a,y,D)$  (since  $x$  definitely-points-to  $y$ ,  $a$  also definitely points to  $y$ ). This is shown in Figure 4.12(d). This example shows that the unmap process is essential in the case that no invisible variable is available. The example in Figure 4.13 shows the importance of the invisible variables in the unmap process.

The points-to information available before the function call `f` at statement 1 is shown in Figure 4.13(a). The result of the map process of function `f` is shown in Figure 4.13(b). The abstract stack representation at statement 3 after processing the body of `f`, is shown in Figure 4.13(c). The map information saved in `map_info` is shown in Figure 4.13(e). The following are the involved sets computed for the unmap process:

```

caller_input = {(a,b,D), (b,c,D)}
callee_output = {(m,l_m,D), (l_m,x,D), (n,x,D)}
kill = {(b,c,D)}
gen = {(b,x,D)}
caller_output = {(a,b,D), (b,x,D)}

```

The 'kill' set is  $\{(b,c,D)\}$  because  $(b,c,D)$  is in the caller but its related points-to relationship which is  $(l_m,2_m,D)$  is not in the callee. The 'gen' set is  $\{(b,x,D)\}$  because  $(l_m, x, D)$  is in callee but its related points-to information in caller which is  $(b, x, D)$ , is not available. Note that the relation  $(m,l_m,D)$  is in callee but since  $m$  is not in the scope of `main` and it is not in the `map_info` set, it has no related relationship in the caller. Also,  $(n,x,D)$  is in the callee but, since  $n$  is local to the callee, it has no related relationship in the caller. The final result is shown in Figure 4.13(d).

```

int x ;
main()
{
    int **a, *b, c ;
    a = &b ;
    b = &c ;
    f(a) ;    /* stmt 1 */
}             /* stmt 4 */
f(m)
int **m ;    /* stmt 2 */
{
    int *n ;
    n = &x ;
    *m = &x ;
}             /* stmt 3 */

```

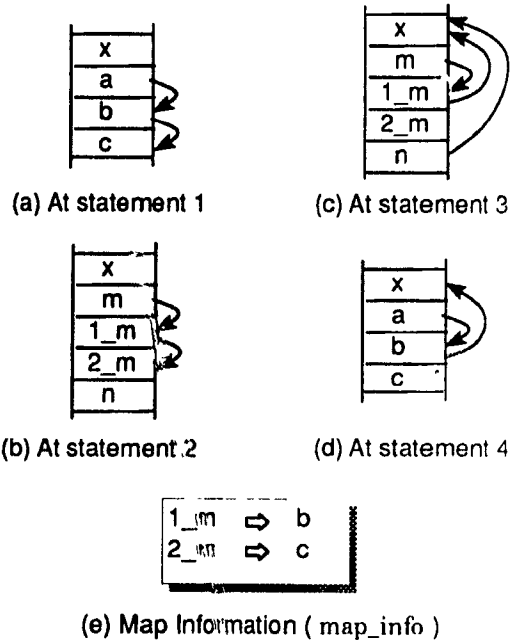


Figure 4.13: An example of unmap process in the presence of invisible variables

Through this example, we see that variables which are not in the scope of a function may get changed through a pointer variable. One can not take care of this side effect without using the map information (`map_info`).

In the next section, we extend the map and the unmap processes to more accurate methods which are implemented in this thesis.

### Extended Map and Unmap Processes

The map/unmap algorithm plays an important role in the accuracy of the inter-procedural analysis. We have designed an accurate map/unmap algorithm based on the following two facts: (i) It is safe to say that an invisible variable represents more than one variable. However, since this may result in some extra possibly point-to relationships, we would like to reduce the number of such cases. (ii) Two abstract stack locations are not allowed to represent the same variable and such cases should be avoided.

The following are the major points of these algorithms which will be subsequently discussed in this section:

1. If a variable is a pointer to a structure, it should have a higher priority in the map process. Note that structures are explained in Chapter 5. In the case of having difficulty understanding this part, it might be beneficial to return to this section after reading Chapter 5.
2. In the map process, the definitely-points-to relationships have higher priority than possibly-points-to relationships.
3. In the map and unmap processes, as far as it is possible, definitely points to relationships are not changed to possibly-points-to.

The first point concerning our map/unmap process is related to pointers to structures. The goal is to avoid cases that two invisible variables stand for the same location in the abstract stack. When a variable `y` points-to an invisible variable of structure type `1_y` with fields `.f1` and `.f2`, if `1_y` stands for structure variable `a` in the caller, clearly `1_y.f1` stands for `a.f1` and `1_y.f2` stands for `a.f2`. In order to make sure that no other invisible variable is already standing for `a.f1` and `a.f2`, we give a higher priority to pointers to structures. We first assign names for the invisible variables of structure type (at this level we do not assign the points-to information). Then, we apply the map process to all the pointer type variables. In this manner, we guarantee that the double naming for the fields of a structure can not appear. This point is essential for the correctness of the concept of invisible variables. If this point is not considered, two invisible variables can stand for one variable which contradicts the main concept. This point is explained in the example given in Figure 4.14.

At statement 1 of Figure 4.14, if `c` is processed before `a`, we get two invisible variables `1_x` and `1_y.f1` standing for the variable `b.f1`. While in our map process, the naming of variable `a` (a pointer to structure) is applied first. This results in the map information represented in Figure 4.14(e). Then, the map process is applied to `c` and `a`. Variable `c` has a definitely-points-to relationship with `b.f1`. Since name `1_y.f1` stands for `b.f1`, the relationship  $(x, 1_y.f1, D)$  is resolved and the problem of having another name (`1_x`) for `b.f1` can not appear anymore. The related abstract stack representation and map information for this example are given in Figure 4.14(a) to 4.14(e).

The second point of our map/unmap process is related to the kind of points to relationship (definitely or possibly). In the map process algorithm explained earlier, there is a potential of changing definitely-points-to relationships to possibly points to and/or generating some extra points-to information. This happens when an invisible variable, e.g. `1_x`, stands for more than one variable and a points-to relationship to `1_x` is generated because of a relationship to one of the variables that `1_x` stands



```

typedef struct{
    int f1,
        f2 ;
} F00 ;

int *z ;

main() {
    F00 *a, b ;
    int *c ;

    a = &b ;
    c = &b.f1 ;
    f(c, a) ;      /* stmt1 */
}                  /* stmt2 */

f(x, y)
int *x ;           /* stmt3 */
F00 *y ;
{
    z = &(y->f2) ; /* stmt4 */
}                  /* stmt5 */

```

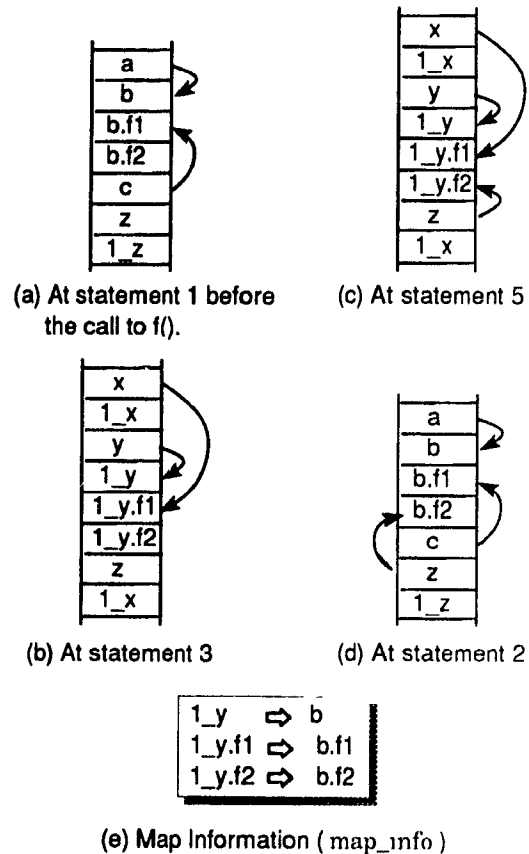


Figure 4.14: An example of our accurate map process when an invisible variable stands for a structure.

for (not to all the variables that 1\_x stands for). An example of this inaccuracy is represented in Figure 4.15.

The points-to information available before the function call `f` at statement 2 is shown in Figure 4.15(a). The result of the map process of function `f` is shown in Figure 4.15(b). Since `b` definitely-points-to `c` and `1_a` represents `c` and `d` (shown in Figure 4.15(e)), the relationship between `b` and `1_a` is established. The fact that the invisible variable `1_a` stands for more than one variable (in this case `c` and `d`) and `b` points-to `1_a` but not to all the variables that `1_a` stands for (in this case `b` points-to `c` and not to `d`), results in some inaccuracy. After the unmap process, the relationships `(b,c,P)` and `(b,d,P)` are propagated to the function `main` and replace `(b,c,D)`. The

```

int *a, *b ;
main() {
    int c, d, e ,

    if (...)
        a = &c ;
    else
        a = &d ;
    b = &c ; /* stmt1 */
    f() ;    /* stmt2 */
}           /* stmt3 */
f()         /* stmt4 */
{
    int *z, w ;

    z = &w ;

}           /* stmt5 */

```

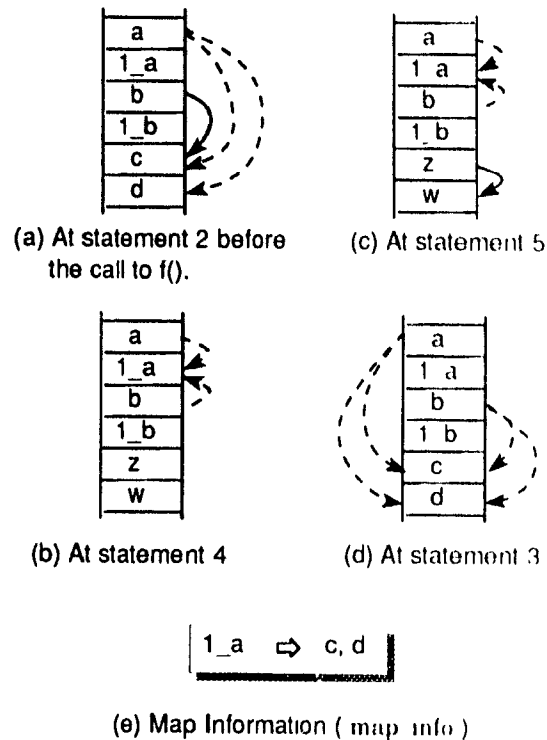


Figure 4.15: An example of general map process when an invisible variable stands for more than one variable.

result of unmap process is represented in Figure 4.15(d). As the result, the definitely points-to relationship (b,c,D) is replaced by possibly-points-to relationship and an extra triple is generated.

In order to reduce the number of such cases, we associate only one variable to an invisible variable (as long as it is possible). This is achieved in our map process by giving priority to the order of processing the parameters of a function. We first apply the map process to the pointer parameters and global variables which have a definitely-points-to relationship, and then to the pointer parameters and global variables which have possibly-points-to relationship. Using this method on the example of Figure 4.15, the global variable b will be processed before a. As the result of the map process, 1\_b represents c, and 1\_a represents d. The relationships after the map process of function f are as follows:

$$\{(b, 1\_b, D), (a, 1\_b, P), (a, 1\_a, P)\}$$

In this case, the unmap process does not change the points-to information of the caller ( $\{(b,c,D), (a,c,P), (a,d,P)\}$ ).

This improvement is not always applicable. When the following two cases exist at the same time, one can not avoid the generating of extra triple(s):

1. two invisible variables, `1_a` and `1_b`, stand for more than one variable. and,
2. there is a variable that points-to both of the invisible variables (`1_a` and `1_b`) and not to all the variables that `1_a` and `1_b` stand for.

In the example of Figure 4.15, this case appears by the replacement of statement 1 with the following statements:

```
if (...)
    b = &c ;
else
    b = &e ;
```

The related abstract stacks and map information of this case are represented in Figure 4.16.

Combining the first and second points of our map/unmap process results in the following priority for map process:

1. map naming process for definitely-points-to relationship to structure variables.
2. map naming process for possibly-points-to relationship to structure variables
3. map process for definitely-points-to relationship.
4. map process for possibly-points-to relationship.

It should be mentioned that for each of these cases all the variables that a parameter or global variable is pointing to are recursively processed to make sure that all the indirectly accessed variables are checked. Therefore, all the relationships related to parameters and global variables are processed four times. Consider the following example:

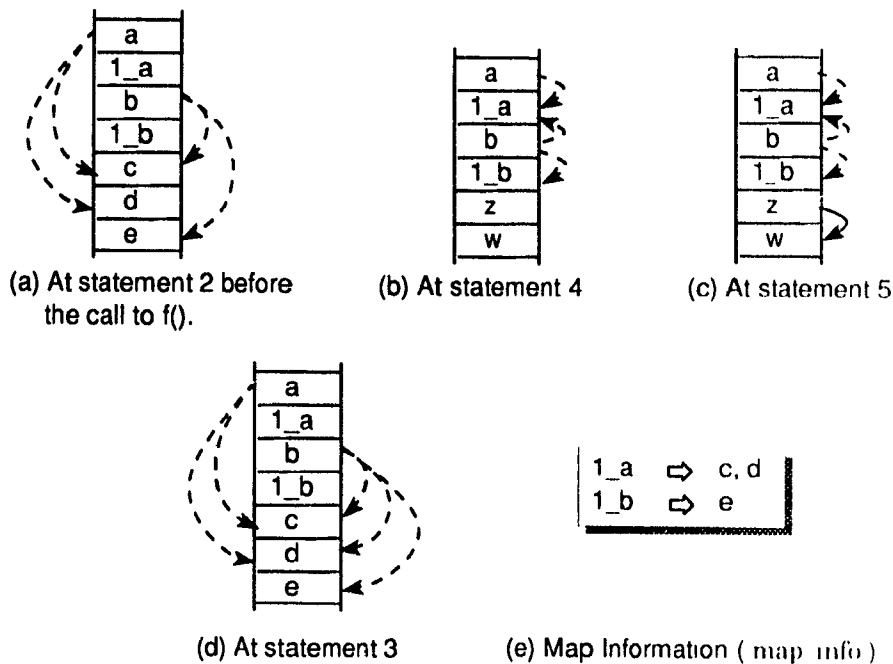


Figure 4.16: An example of our accurate map process when two invisible variable stand for more than one variable.

```

main() {
    int **a, *b, c ;

    if (...)
        a = &b ;
    b = c ;
    f(a) ;    /* stmt1:  {(a,b,P), (b,c,D)} */
}
f(int **x)   /* stmt2:  {(x,1_x,P), (1_x,2_x,D)} */
{
    ...
}

```

The points-to information before the function call `f` is represented at statement 1. The points-to information after the map process is represented at statement 2. In the third step of our map process, the relationship  $(1_x, 2_x, D)$  is generated which is directly resulting from  $(a, b, P)$ . In the fourth step of our map process, the relationship

(x,1\_x,P) is generated which is directly resulting from (a,b,P).

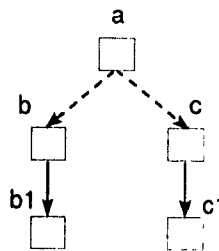
The third point of our map/unmap process has the same goal as the second point which is keeping the definitely-points-to relationships as long as possible. This case is related to multi-level pointers when the following two cases are valid at the same time:

1. An invisible variable **1\_x** of pointer type stands for more than one variable (e.g. **a** and **b**, and,
2. **a** and **b** have definitely-points-to relationship with some other variable(s) (e.g. if they both definitely-points-to **c**).

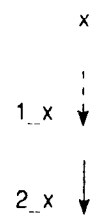
Since a variable (in this case **1\_x**) can not have a definitely-points-to relationship with two variables, such a relationship should be replaced by possibly-points-to. This replacement can be avoided when the variables that **1\_x** definitely-points-to are represented by a unique variable or by a unique invisible variable. An example of this case is given in Figure 4.17.

```
int *y ;
main() {
    int **a, *b, *c, b1, c1 ;

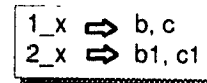
    if (...)
        a = &b ;
    else
        a = &c ;
    b = &b1 ;
    c = &c1 ;
    f(a) ;      /* stmt1 */
}
f(int **x) {   /* stmt2 */
    y = *x ;
}
```



(a) At statement 1 before the call to f.



(b) At statement 2



(c) Map Information ( map\_info )

Figure 4.17: An example for the third point of the map process

The memory representation before/after the map process and the map information are shown in Figure 4.17(a) to (c). Variable **1\_x** stands for two variables **b** and **c** which have a definitely-points-to relationship with **b1** and **c1**, respectively. Since **2\_x** represents both **b1** and **c1**, we keep the definitely-points-to relationship between **1\_x**

and 2\_x. By applying a proper unmap process, we avoid generating extra information like (b,c1,D) .

Our unmap process does not affect the definitely-points-to information of the caller that already exists in the callee. This is shown in the 'gen' set of the following unmap process.

```

/* our accurate unmap process */
unmap_process( caller_input, callee_output, map_info)
{
    kill = {(x, y, rel) : (x, y, rel) ∈ caller_input ∧
                        (x_map, y_map, rel) ∉ callee_output}

    gen = {(x, y, P) : (x_map, y_map, P) ∈ callee_output} ∪
          {(x, y, D) : (x_map, y_map, D) ∈ callee_output ∧
                        (x, y, D) ∉ caller_input}

    caller_output = gen ∪ (caller_input - kill)

    return(caller_output)
}

```

The precision of this new condition is obvious when an invisible variable stand for only one variable. Following is an informal proof for the correctness of this new condition in our unmap process when an invisible variable stands for more than one variable: If there is no change in the points-to relationship of an invisible variable say 1\_x, the unmap process is correct (because it will not change the points to information of caller). Otherwise, since an invisible variable is not directly accessible, the corresponding change has to be done through an indirect reference, say \*x. Since 1\_x stands for more than one variable, x and 1\_x has a possibly-points-to relationship. Therefore, any change of 1\_x (\*x), changes all its definitely-points-to relationship to possibly-points-to (this is a basic points-to rule, refer to Chapter 3). In the example of Figure 4.17, if \*x changes, the relationship (1\_x,2\_x,D) changes to (1\_x,2\_x,P) ∨ the result, the newly change in the unmap process will not be applicable. We conclude that the new changes of the unmap process are only for the caller when the definitely-points-to information is not changed in the callee.

The algorithm for the precise map process is given in Appendix B.

### 4.2.3 Return Statement

All our previous discussions concerning function calls were based on the form `f( var1, var2, ect. )`. In this section, we discuss the form `x = f( var1, var2, ect. )`. This form happens along with at least one occurrence of `return` statement in the body of function `f` (e.g. `return(var)`).

For each function `f` returning a pointer-type variable, we define a global variable `return_f` with the same type as `f`. Using this newly defined variable, we treat (the C-program is not changed) `return(var)` as:

```
return_f = var ;  
return ;
```

and we treat `x = f( var1, var2, ..., varn )` as:

```
f( var1, var2, ..., varn ) ;  
x = return_f ;
```

The above lines are processed in the usual way as described before (note that `x` can be replaced by `*x`). The `return` statement is similar to `break` statement in the sense that it returns the control to the caller. To process the `return` statement, we use the structure `return-list` which is a list of points-to information reaching `return` statements. The following should be done in the presence of a `return` statement:

- Merge the points-to information entering to a given `return` statement with the corresponding `return-list`.
- Pass BOTTOM as the output. In this case, BOTTOM means that the statements after `return` does not contain any valid information. In other words they are dead code.

At the end of each function, we merge the points-to set reaching the end of function with the information saved in `return-list`. This would be the merge of all potential outputs.

Figure 4.18 shows the corresponding algorithm together with a graph representation of `return` statement.

Figure 4.19 illustrates an example of `return` statement. Figures 4.19(a) and (b) are the main program and its simplified version. The simplifier replaces the address operand occurred in a `return` statement by a newly defined variable, in this case `temp0` and `temp1` at statements 4 and 8. Figure 4.19(c) to (n) shows the abstract

```

process_return(in) {
    result_list = merge_info(result_list, in) ;
    return(BOTTOM) ;
}

```

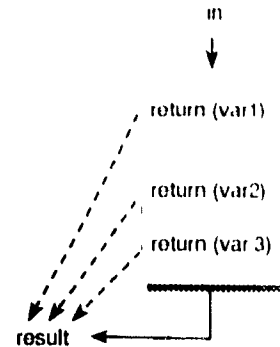


Figure 4.18: The algorithm for **return** statement

stack corresponding to each statement during our analysis. We explain some of these cases in detail. After the execution of the statement 4, location **f** return points to the same location as **temp0** is pointing to (Figure 4.19(f)). This information is saved in **return-list** and **BOTTOM** is returned (Figure 4.19(g)). The information entering statement 6 is the same as the information entering statement 2 (Figure 4.19(h)). The same process as explained above is repeated for the **return** appearing at statement 8. This is composed of saving the information in **return-list** and passing **BOTTOM**. Statement 10 contains the merge of the information at statements 5 and 9. When the process of the function is completed, we merge the final result (Figure 4.19(k)) with the points-to information in **return-list** (Figures 4.19(f) and (j)). This results in the points-to information shown in (Figure 4.19(l)). Now we treat statement

```
a = f() ; /* stmt12 */
```

as if it were composed of the following two statements:

```

f() ;          /* stmt12a */
a = return_f ; /* stmt12b */

```

The result after unmaping of the function **f** is shown in Figure 4.19(m). The last step is the process of statement 12b. When the process of statement 12b is completed, a the points-to information of **return\_f** has no further use, it will be removed. The final result is shown in (Figure 4.19(n)).



```

int x, y ;
main()
{
    int *a ;
    a = f() ;
}

int *f()
{
    if (x == 1)
        return(&y) ;

    else
        return(&x) ;
}

int x, y ;
main()
{
    int *a ;
    a = f() ;
}

int *f()
{
    int *temp0, *temp1 ;

    if (x == 1)
    {
        temp0 = &y ;
        return(temp0) ;
    }
    else{
        temp1 = &x ;
        return(temp1) ;
    }
}

```

(a) A C-program

(b) The simplified program

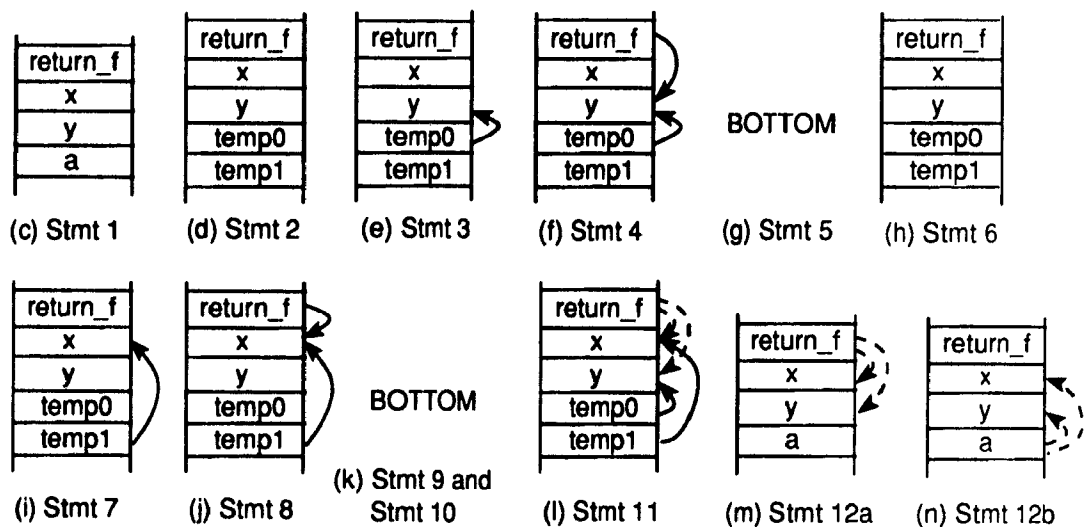


Figure 4.19: An example of **return** statement.

### 4.3 Interprocedural Analysis for Recursive Function Calls

Now that the process of non-recursive function calls has been completed, we discuss the processing of recursive function calls. The recursive calls are more complicated because the number of iterations is not known.

The difference between recursive and non-recursive calls in our analysis is reflected in the corresponding invocation graphs. Recall from Section 4.1 that the invocation graph is composed of three different kinds of nodes, ordinary nodes, recursive nodes, and approximate nodes; whereas edges could be either calling edges or approximate edges. If the invocation graph node corresponding to a function call is an ordinary node, we apply the map process, analyze the body of the callee, merge the output with the **return-list**, and perform the unmap process, as explained in Section 4.1.

If the invocation graph node corresponding to a function call is a recursive node, a fixed-point has to be found. A fixed-point for a recursive node is reached when, (i) there is only one input to the recursive node (otherwise the inputs are merged and the process of the body restarted), (ii) the input does not generate a different output from the previous iteration to the recursive function. This is the same idea as the fixed-point for **while** loop statement.

Unlike recursive and ordinary nodes, an approximate node does not cause a process to the body of the function. It just approximates the output. If the input (**in**) to an approximate node (**f**) is already entered to the recursive node, it means that this input is already processed, therefore, the related output will be taken; otherwise **in** is a new possible input to **f** implying that **f** should be processed accordingly.

The process of recursive calls does not make any difference in map and unmap processes. The only difference is in the way that we treat recursive/approximate node in the invocation graph (every call has to pass through invocation graph).

Now that the idea behind our approach is explained, in the following we explain our algorithm.

If the call node corresponding to the function call is a recursive node, we have to iterate until a fixed-point is reached. We have a list of input-output pairs of information stored in a recursive node. These pairs correspond to all the different types of inputs and outputs which can possibly occur during one iteration of the fixed-point calculation. In each iteration, we check if there is more than one input to the node under consideration. If the answer is positive, then we conclude that a recursive call is found with an input not included in the current approximation of input. In this case, we merge all these inputs, store this newly merged input in the tree, and start

again with the newly merged input. If there is a single input-output pair left, and that output and the newly created output from the most recent iteration are not the same (fixed-point is not reached), we merge the two outputs, store the newly merged output in the tree, and start again with the input available in the input-output pair. Finally, when a fixed-point is reached, there is a single input-output pair which corresponds to a superset of all the input-output pairs possible for this function. Intuitively, this pair summarizes all the possible input-output pairs of the invocation graph.

If the call node corresponding to the function call is an approximate node, we find its corresponding recursive node in the invocation graph. We check the list of input-output pairs to see if an output exists for this particular input. If it does, we just return this output. Otherwise, we store this new input in the recursive node and return **BOTTOM**. Here, **BOTTOM** has essentially the same properties as the **BOTTOM** described in the previous chapter. In specific, it means that we still do not know the output for this particular input.

General algorithm for the appearance of a call expression is as follows.

```

/*
 * Function name : func_points_to
 * Purpose : find out the points-to information after
 *           processing function call (that can be a recursive call),
 *           using the input information to the call
 * Parameters : call_expr_node -- the call node in caller
 *             in_data -- the input information to
 *                       'call_expr_node' before it is processed
 *             Note that the field 'in' represents the
 *             points-to relationships.
 * Return : out_data -- the output information of caller
 *             after processing function call
 */
func_points_to( call_expr_node, in_data)
{
    /* get the related invocation graph node to the callee by using the
     * invocation graph node of caller ('in_data.cur_ig_node') and
     * related call number of callee ('call_expr_node.call_expr_num').
     * For example if 'f(a)' is the third function call in the body of 'g',
     * the related invocation graph node to callee 'f(a)' is the third child
     * of the related invocation graph node of caller 'g' */
    ig_node = get_related_ig_node( in_data.cur_ig_node,
                                   call_expr_node.call_expr_num)

    /* get the function node of the callee from the 'call_expr_node' */
    func_node = get_func_node( call_expr_node) ;

    /* get the list of arguments (actual parameters) */
    arg_lst = get_arg_lst( call_expr_node) ;

    /* 'map_process' maps the points-to information ('in_data.in'), returns
     * the mapped points-to information ('func_in_data.in') and the map
     * information ('map_info'). */
    [map_info, func_in_data.in] = map_process( func_node, arg_lst, in_data.in)

    /* save the map information and the entry points-to relationships in the
     * invocation graph node related to the function call (the output points to
     * relationships is saved as BOTTOM). */
    save_in_info( ig_node, map_info, func_in_data.in) ;

    /* get the body of the callee. */
    func_body = get_func_body( func_node) ;

```

```

switch (get_mode( ig_node)){ /* check for the mode of the invocation graph node */
  case "ordinary":
    /* if the invocation graph node is ordinary , analyze the body of callee */
    func_out_data = points_to( func_body, func_in_data) ;
    /* merge the result with all the possible output from return statement */
    func_out_data = merge_info( func_out_data, return_lst) ;
    break ;
  case "approximate":
    /* get the recursive node to the approximate node, by following
     * the approximate arc. */
    recursive_node = ig_graph.approximate_arc ;
    /* search for 'func_in_data.in' in 'recursive_node'. If it is
     * found (if 'func_in_data.in' is a subset of already exist
     * input), return the related output. Otherwise return BOTTOM
     * and save 'func_in_data.in' in the 'recursive_node' as a new
     * possible entry. */
    func_out_data.in = search_info( recursive_node, func_in_data.in)
    break ,
  case "recursive":
    /* process the body of the callee. */
    func_out_data = points_to( func_body, func_in_data) ;
    /* merge the result with all the possible output from return statement */
    func_out_data = merge_info( func_out_data, return_lst) ;
    /* the points-to information from the previous iteration is BOTTOM
     * (because no previous iteration exist for the first iteration) */
    ig_output = BOTTOM ;
    /* check for the fixed-point. A fixed-point is reached when there
     * is one entry to the invocation graph node and for a given input
     * the output is not changed compare to the previous iteration */
    while (( num_inputs_in( ig_node) > 1) ||
           ( ig_output not contained in func_out_data.in))
    {
      if ( num_inputs_in( ig_node)) > 1) {
        /* there is more than one input. Merge all the inputs and
         * save the result in 'func_in_data.in', and in the
         * 'ig_node' as the next input. */
        func_in_data.in = ig_lst_merge( get_inputs_from( ig_node))
        /* save new input-output (func_in_data.in,BOTTOM) in the ig_node */
        save_ig_node( ig_node, func_in_data.in, BOTTOM) .
      }
      else
        /* there is a different output compare to the previous
         * iteration. So merge them. */
        ig_output = Merge( func_out_data.in, ig_output) ,
        /* process the body for the next iteration */
        func_out_data = points_to( func_body, func_in_data) ,
        func_out_data = merge_info( func_out_data, return_lst) .
    }
}

```

```

/* save the points-to result of analysis of callee, into related
 * invocation graph node */
save_out_info( ig_node, func_out_data.in) ;

/* the unmap process get the points-to information of caller
 * and callee and returns the new set of points-to information
 * for caller ('.in' is the field related to points-to info) */
out_data.in = unmap_process( in_data, func_out_data.in, map_info) ;

return( out_data) ;
}

```

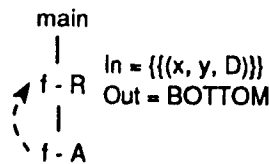
An example is given in Figure 4.20. The process starts from the `main` function. The intraprocedural analysis is done until reaching the function call `f(a)` which invokes the map process. The set of points-to information is saved in the related node in the invocation graph (`f-R`). This is shown in Figure 4.20(b). The information available at statement 1 is shown in Figure 4.20(c). The recursive call `f(n)` invokes the map process again. The information after the map process is saved in the related invocation graph node (`f-A`) as shown in Figure 4.20(d). The 'In' set of `f-R` is searched for the 'input' set of `f-A`. Since it is not found, the 'input' set will be saved in the 'In' set of `f-R` (this is considered as a possibly new input), and `BOTTOM` will be returned. Figure 4.20(e) shows the result after unmapping. The process will be continued by following statements 2 and 3. Since the 'In' set of `f-R` has more than one set, we merge all the elements in 'In' set of `f-R`. The processing of function `f` is restarted with the result of the merge as a new input (refer to Figure 4.20(f)). The intraprocedural analysis is continued until reaching the recursive call to `f(n)`. Set  $\{(x,y,D), (m,y,D)\}$  is the result of map process. A more general case of this set (which is  $\{(a, y,D), (m,y,P)\}$ ), is available in the 'In' set of `f-R`. Therefore, we return the related output which in this case is `BOTTOM`. The process continues by statements 2 and 3. The points-to information reaching the end of the function `f`, which is denoted as 'output', is shown in Figure 4.20(g). At this iteration, there is only one element left in the 'In' set. But, since the 'output' set from Figure 4.20(g) is different from the 'Out' set of `f-R` (which is `BOTTOM`), we assign 'output' to the 'Out' set of `f-R` as the new output. Then, we start the third iteration. Figure 4.20(h) represents the information at this stage. The same process, as explained for other iterations, has to be done once again. The final result is shown in Figure 4.20(i). There is only one set in the 'In' set, and the 'output' set is same as the 'Out' set. Consequently, the fixed point is reached and 'output' set is returned as the result of the recursive function `f()`.

```

int *x, y ;
f (m)
int *m ;
{
    int *n ;
    if (y == 1)
    {
        n = &y ;
        /*stmt1*/
        f (n) ;
        /*stmt2*/
    }
} /*stmt3*/
main () {
    int *a ;
    x = &y ;
    f (a) ;
}

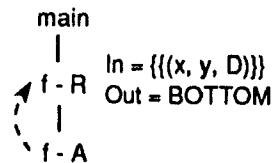
```

(a) The C program



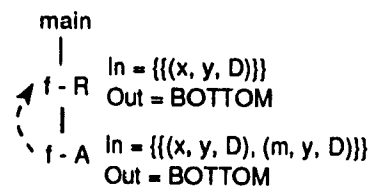
output = {(x, y, D)}

(b) first iteration



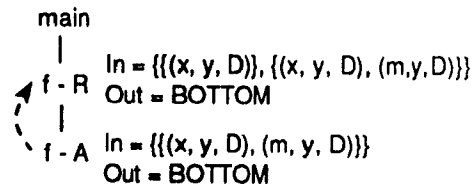
output = {(x, y, D), (n, y, D)}

(c) at stmt 1



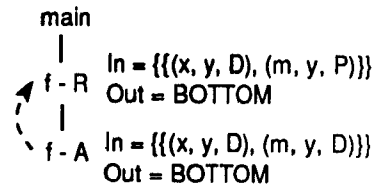
output = {(x, y, D), (m, y, D)}

(d) after map process



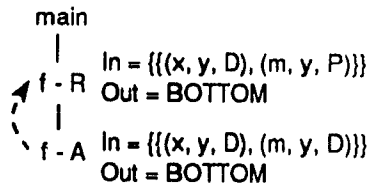
output = BOTTOM

(e) after unmap process



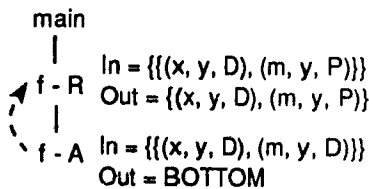
output = BOTTOM

(f) second iteration



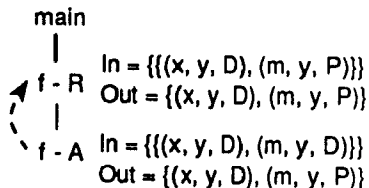
output = {(x, y, D), (m, y, P)}

(g) at stmt 3



output = {(x, y, D), (m, y, P)}

(h) third iteration



output = {(x, y, D), (m, y, P)}

(i) The fix-point information

Figure 4.20: An example of the interprocedural points-to analysis with recursive function call.

## Chapter 5

# Aggregate Structures

Our analysis concerning scalar variables was presented in Chapters 3 and 4. In order to analyze real C-language benchmarks, one should also consider arrays and structures. In this chapter, we explain how these more general cases are handled. We first discuss the abstract stack representation for non-scalar variables in Section 5.1. Then, in Section 5.2, we explain how updating structured variables is related to the updating of their components. Finally, in Section 5.3, we give the basic rules for calculating the points-to relationship of each possible case.

### 5.1 Abstract Stack Locations for Arrays and Structures

The major difference between aggregate structures and scalar variables is that aggregate structures are composed of many stack locations. Our objective is to have a name for each location in the abstract stack. We explain our abstract stack representation for arrays in Section 5.1.1, for non-recursive structures<sup>1</sup> in Section 5.1.2, for recursive structures in Section 5.1.3, and for union type in Section 5.1.4.

#### 5.1.1 Arrays

An array is a sequence of variables of the same type. In our analysis, we consider all the elements of an array as one location in the abstract stack. Thus, the points-to analysis finds the relationships between entire arrays, and this information is then

---

<sup>1</sup>A structure is *non-recursive* if none of its fields (directly or indirectly) points to the same structure; otherwise, it is called a *recursive* structure. Linked list is an example of recursive structure.



used in the array dependence tester which reasons about locations within the arrays [Ban76, Wol89, ZC90]. Our points-to analysis is in fact a pre-processing step for a practical array dependence analyzer which computes dependences for array variables in nested loops [Jus93].

### 5.1.2 Non-recursive Structures

Unlike in the case of arrays, each element of a structure is explicitly accessible (e.g. `a.b`). This makes structures more complicated than arrays. Each possible access to a field of a structure should have a unique location, and consequently a unique name, in the abstract stack.

In order to have a unique location for all the possible fields of a structure in the abstract stack, we enumerate the fields of the structure. In the example given in Figure 5.1(a), all possible ways to access the structure `root` (the root of data structure) are `root`, `root.field1`, and `root.field2`. The corresponding abstract stack representation is shown in Figure 5.1(b).

```
struct {
    int      *field1 ;
    float    field2 ;
} root ;
```

(a) The C-program

root
root.field1
root.field2

(b) The abstract stack representation

Figure 5.1: An example of structure representation in the abstract stack.

In the case of a nested structure, this process should be done recursively until all the possible accesses to a structure are considered. Figure 5.2 represents an example of a nested data structure along with its related abstract stack representation.

What we explained is sufficient for any structure variable which is local to a function but it is not sufficient for global variables and actual parameters of structure type that have a field of pointer type. Assume that `a` is a global variable or actual parameter of structure type that has a field of pointer type `b`. Location `a.b` can point to the location(s) that are not in the scope of that function. Therefore, there are not enough names in the abstract stack for accessing `*(a.b)`. To solve this problem, we use the concept of *invisible variables for structures*.

The concept of invisible variables for structures is similar to the corresponding

```

typedef struct{
    int **sub_field1 ;
    float sub_field2 ;
} F00 ;

struct {
    int      *field1 ;
    F00      field2 ;
    char     **field3 ;
} root ;

```

root
root.field1
root.field2
root.field2.sub_field1
root.field2.sub_field2
root.field3

(a) The C-program

(b) The abstract stack representation

Figure 5.2: An example of nested structure representation in the abstract stack

concept for scalar variables which was discussed in Chapter 4. For each actual parameter or global variable of structure type which has a field of pointer type, we add all the invisible variables of corresponding field to the abstract stack. For example the invisible variables corresponding to variable `x.y` with type `int()` are `x.1.y` and `x.2.y` with type `int*` and `int`, respectively.


Figure 5.3 shows a C-program and its corresponding abstract stacks. The invisible variables corresponding to functions `main` and `f`, with their related types, are shown in Figures 5.3(a) and (b), respectively. Using the invisible variables, we guarantee that any combination for the access to a structure will have a location in the abstract stack.

Without having a specific strategy for *recursive data structures*, this construction method results in infinity. The proposed strategy for handling this problem is given in the next section.

### 5.1.3 Recursive Structures

Some data structures are defined recursively. This means that they have a field which points to the same data structure (directly or indirectly). Linked list is the simplest example of a direct recursive data structure. We explain our strategy in the design of the abstract stack for the recursive data structures using the linked list example. A linked list is typically defined as:

```
typedef struct{
    int **z ;
    int w ;
} CAT ;
```

	<u>Related type:</u>
	FOO*

(a) Abstract stack of main()

```
typedef struct{
    int x ;
    CAT *y
} FOO ;
```

```
main(){
    FOO *a ;
    ...
    f(a) ;
}
```

```
void f(m)
FOO *m ;
{ }
```

Related type:	
m	FOO*
1_m	FOO
1_m.x	int
1_m.y	CAT*
1_m.1_y	CAT
1_m.1_y.z	int**
1_m.1_y.1_z	int*
1_m.1_y.2_z	int
1_m.1_y.w	int

(b) Abstract stack of f()

Figure 5.3: An example of the abstract stack representation for invisible variables.

```
struct foo {
    int      data ;
    struct foo *next ;
} cell ;
```

Using the method given in the previous section, we first need to construct the locations shown in Figure 5.4. The process of attaching the fields of the structure

Related type:	
cell	struct foo
cell.data	int
cell.next	struct foo*
cell.1_next	struct foo

Figure 5.4: The incomplete abstract stack for the structure foo

foo to the location cell.1\_next results in an infinite loop. This is due to the fact that there will be always a location with the type of structure foo. The solution to this problem is as follows: Each time that we add a new invisible variable of structure

type, we first check the type of the previously added fields. If the type already exists, a flag will be set and the new invisible variable points to the corresponding location in the abstract stack. Figure 5.5 shows the abstract stack for the above example. A

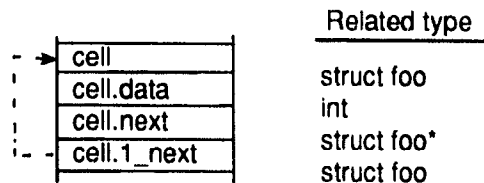


Figure 5.5: The complete abstract stack for the structure foo

the type of cell.l\_next already exists (**struct foo**), we do not further add the field of structure cell.l\_next.

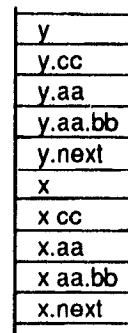
An example of a multi-level structure is shown in Figure 5.6(a). The abstract stacks corresponding to functions **main** and **f** are shown in Figures 5.6(a) and (b) respectively.

```
struct cat_type {
    struct foo_type *bb ;
} ;
```

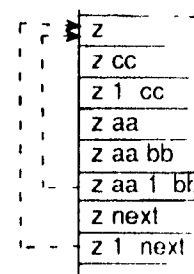
```
struct foo_type{
    int *cc ;
    struct cat type aa ;
    struct foo_type *next ;
} ;
```

```
main()
{
    struct foo_type x, y ;
    ...
    f(x) ;
}
```

```
f(z)
struct foo_type z ;
{...}
```



(a) Abstract stack of main()



(b) Abstract stack of f()

Figure 5.6: An example of a recursive data structure

In this way, we are actually approximating a recursive data structure to be composed of one cell with a pointer pointing back to itself.

#### 5.1.4 Union Type

Union type variables are not considered in our present version. However, since these are an extension to structures, they can be easily included using the same method. This is achieved by attaching all possible reference names to a union type variable. The only difference between unions and structures is that in the case of unions more names are created. We plan to include unions to our analysis as part of our future work.

### 5.2 Handling Complex Structure References

A complex structure reference is a variable of structure type. Such variables can appear in a basic statement (e.g.,  $a = b$  when  $a$  and  $b$  are of structure type), or as a parameter of a function (e.g.  $f(a)$  when  $a$  is of structure type). In this section, we show how our analysis for complex structure references is converted into a sequence of simpler statements that involve only simple structure references. Then, in the next section, we explain our rules to get points-to information for the basic cases involving structure references. We first explain it for the case of intraprocedural analysis (basic statements) and then for interprocedural analysis (function calls).

A basic statement is of structure type if its left hand side (lhs) and right hand side (rhs) are of structure type. Each basic statement of structure type is converted into several simpler basic statements. This is achieved by attaching the fields of the structure to the variables appearing in lhs and rhs. If the newly generated basic statement is of structure type, the same process will be repeated recursively. Finally, we get a set of basic statements of either basic type (integer type, float type, ect.) or pointer type. Then, we apply our basic rules to each of the newly generated basic statements which are of pointer type. This is shown in the following example

```

void main() {
    struct foo{
        int *subfield1 ;
    } a, b ;
    struct{
        float      *field1 ;
        int        field2 ;
        struct foo *field3 ;
    } x, *y ;

    a = b ; /* stmt 1 */
    x = *y ; /* stmt 2 */

```

Statement 1 is of structure type. This statement is treated as equivalent to the following basic statement:

```
a.subfield1 = b.subfield1 ;
```

Statement 2 is also of structure type. This statement is treated to be equivalent to the following three basic statements:

```

x.field1 = *y.field1 ; /* stmt 2a */
x.field2 = *y.field2 ; /* stmt 2b */
x.field3 = *y.field3 ; /* stmt 2c */

```

The newly generated basic statement 2a is of pointer type and one of the basic cases. Statement 2b is of basic-type and therefore it does not make any difference in the points-to information. Statement 2c is of structure type, and, consequently, the breaking down process is performed which results in the following basic statement

```
x.field3.subfield1 = *y.field3.subfield1 ;
```

The above basic statement is of pointer type and one of the basic cases. As there are no more basic statements of structure type left, the process is completed.

For interprocedural analysis, the process is the same as explained in Chapter 4. The only difference is in assigning the actual parameters to formal parameters. If a function is called as  $f(avar_1, avar_2, \dots)$  and defined as  $f(fvar_1, fvar_2, \dots)$ , we

should compute the points-to information resolved from:

$$\text{fvar}_1 = \text{avar}_1$$

$$\text{fvar}_2 = \text{avar}_2$$

...

If any of the above statements is of structure type, we should do the same thing as we did for the basic statements of structure type. This means that for each equation  $\text{fvar}_i = \text{avar}_i$  of structure type with  $n$  fields  $\mathbf{f}_1, \mathbf{f}_2, \dots, \mathbf{f}_n$ , we should add the points-to relationships obtained from the following statements:

$$\text{fvar}_i.\mathbf{f}_1 = \text{avar}_i.\mathbf{f}_1$$

$$\text{fvar}_i.\mathbf{f}_2 = \text{avar}_i.\mathbf{f}_2$$

...

$$\text{fvar}_i.\mathbf{f}_n = \text{avar}_i.\mathbf{f}_n$$

The same process will be repeated for any newly generated basic statement which is of structure type.

We conclude that the problem of intraprocedural and interprocedural points-to analysis for complex structures can be broken into smaller problems concerning basic statements of pointer type with aggregate structures in lhs and rhs. In the next section, we explain how these basic statements can be solved.

### 5.3 Points-to Analysis for Aggregate Structures

In the previous sections, we explained the stack representation for aggregate data structures. In this section, we expand our points-to analysis for aggregate structures. Figure 5.7 shows an overall view of our points-to analysis. The only case that aggregate structures can affect interprocedural analysis is when a parameter is of structure type. In Section 5.2, we showed that this case can be divided into some basic cases and then the general interprocedural rules given in Chapter 4 will be applied. The intraprocedural analysis is divided into two major groups: (i) basic statements, (ii) compositional control statements. As compositional control statements are independent of the type of data structure, there is no need to consider them in this chapter. Consequently, it is sufficient to find a solution for all the basic statements. Table 5.1 summarizes all basic statements that affect points-to relationships (when the statement is of pointer type). This is a generalized version of the Table 3.3 which

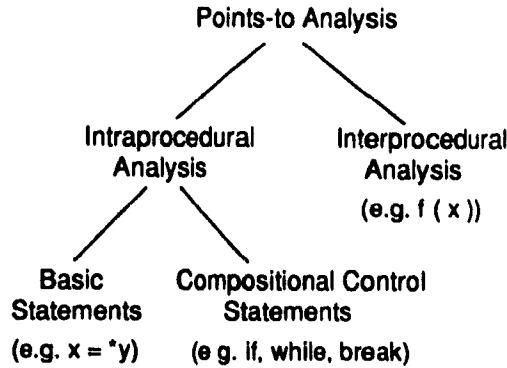


Figure 5.7: The overall view of points-to analysis.

is only for scalar variables. The cases which are denoted by the bold face notation correspond to the entries of Table 3.3.

lhs ↓ rhs →	<b>&amp;y</b>	<b>&amp;y[j]</b>	<b>&amp;y.b</b>	<b>y</b>	<b>y[j]</b>	<b>y.b</b>	<b>*y</b>	<b>(*y).b</b>	<b>(*py)[j]<sup>2</sup></b>	<b>&amp;(*y).b</b>	<b>&amp;(*py)[j]<sup>3</sup></b>
<b>x</b>	<b>1.1</b>	1.2	1.3	<b>2.1</b>	2.2	2.3	<b>3.1</b>	3.2	3.3	7.1	7.2
<b>x[j]</b>	1.4	1.5	1.6	2.4	2.5	2.6	3.4	3.5	3.6	7.3	7.4
<b>x.a</b>	1.7	1.8	1.9	2.7	2.8	2.9	3.7	3.8	3.9	7.5	7.6
<b>*x</b>	<b>4.1</b>	4.2	4.3	<b>5.1</b>	5.2	5.3	<b>6.1</b>	6.2	6.3	8.1	8.2
<b>(*x).a</b>	4.4	4.5	4.6	5.4	5.5	5.6	6.4	6.5	6.6	8.3	8.4
<b>(*px)[j]<sup>2</sup></b>	4.7	4.8	4.9	5.7	5.8	5.9	6.7	6.8	6.9	8.5	8.6

Table 5.1: All possible cases of basic statements which affect the points-to information when basic statement is of pointer type.

The different kinds of basic statements that can affect points-to analysis are divided into eight cases which are numbered from 1 to 8 in Table 5.1. Each case, which is divided into some subcases, has its own general rule which applies to both scalar variables and aggregate structures. The left hand side number in each entry of the table corresponds to the case and the right hand side number corresponds to the subcase. For example, the entry 7.3 refers to subcase 3 of case 7. To increase the readability of the table, multi-indices such as  $x[i_1][i_2][\dots][i_n]$  are represented as one index such as  $x[i_1]$ , also multi-fields such as  $x.f_1.f_2.\dots.f_n$  are represented as one field such as  $x.f_1$  in the table. This abbreviation does not impose any limitation on the generality and one can easily replace the abbreviated representation by its original form.

<sup>2</sup> $(*py)[j]$  is the semantic representation of  $py[j]$  where 'py' is a pointer to an array. This is not a valid syntax in C-language. We just use this notation to differentiate between pointers to arrays and ordinary arrays.

<sup>3</sup> $\&(*py)[j]$  is the semantic representation of  $\&py[j]$  where 'py' is a pointer to an array. We just use this notation to differentiate between pointers to arrays and ordinary arrays.



The following is the list of aggregate structures that can appear in a basic statement. We first explain the effect of each of the aggregate structures and then give the rules corresponding to each case.

1.  $z[i_1][i_2][\dots][i_n]$  and  $\&z[i_1][i_2][\dots][i_n]$

In this representation, 'z' is explicitly defined as an array. Although in our abstract stack there is only one location related to each access of an array, we treat the first element of an array differently from other elements (the first element is used when indices  $i_1$  to  $i_n$  are equal to zero). The reason behind this approach is that the analyses which needs to know when a pointer variable points to the beginning of the array and when it may point to somewhere in the middle of the array. This is particularly important for the array dependence module which needs to ensure that pointers to arrays point to the beginning of an array.

In the following example after processing statement 1, we would say that b definitely-points-to a if i is zero, otherwise, we would say that b possibly-points-to a.

```
main() {
    int a[7], *b ;
    int i ;
    ...
    b = &a[i]    /* stmt1 */
}
```

The replacement of arrays by scalars in a basic statement does not make any difference to our points-to rules when the first index of an array is used. If the middle index is used, the corresponding relationship should be changed to possibly-points-to.

The notation  $\&z[i_1][i_2][\dots][i_n]$  refers to the address of the abstract stack location related to 'z'.

2.  $(*z)[i_1][i_2][\dots][i_n]$  and  $\&((*z)[i_1][i_2][\dots][i_n])$

In the C-language, there are two ways to access an array. First is a direct access which is done through the name of the array, e.g.  $a[i]$  where a is an array. Second is an indirect access which is done through a pointer to an array, e.g.  $pa[i]$  where pa is a pointer to array(s). These two ways of accessing are syntactically the same, however they have a complete different meaning and are represented differently in SIMPLE (as explained in Chapter 2). The notation  $pa[i]$  means the i'th element of variable(s) that pa points-to. We use the notation  $(*pa)[i]$  to represent the actual meaning of  $pa[i]$ .

This case behaves the same as the general category  $*x$  when the first index of an array is used. When the middle element of an array is used, it behaves differently. In case of indirect reference to arrays, two factors should be considered, (i) the use of the first index of the array, (ii) the possibly/definitely points to relationship of the array pointer to other variable.

3.  $z.f_1.f_2 \dots f_n$  and  $\&z.f_1.f_2 \dots f_n$

Variables of this form have a unique location in the abstract stack. Their only difference with scalar variables is that they have a more complicated name. After finding their location in abstract stack, they are treated in the same way as scalar variables.

4.  $(*z).f_1.f_2 \dots f_n$  and  $\&(*z).f_1.f_2 \dots f_n$

This case is similar to  $*z$ . The only difference is that to get the actual location one should attach fields  $.f_1.f_2 \dots f_n$  to the variable(s) that  $z$  points to

In the rest of this section, we give the detailed rules for the cases 1.8 and 8.1 of Table 5.1. These cases are selected to explain the idea. Some more rules are given in Appendix C. We use the following notations in conjunction with these rules

- D: represents definitely-point-to relationship.
- P: represents possibly-point-to relationship.
- $\bowtie$ : represents merging which is defined in Table 3.1.
- first\_elem: is defined to distinguish the first element of an array. The definition is as follows:

$$\text{first\_elem}([i_1][i_2][\dots][i_n]) = \begin{cases} D & \text{if } i_1 = i_2 = \dots = i_n = 0 \\ P & \text{otherwise} \end{cases}$$

In case of scalar variables, first\_elem is always D.

- input: the points-to relationships at the entry point of a basic statement
- gen: the newly generated points-to relationship(s) by a basic statement
- kill: the points-to relationship(s) that are killed in a basic statement
- changed\_input: is an updated version of 'input' where some of the definitely points-to relationships are changed into possibly-points-to.

**Case 1.8:** The general format of this case is:

```

    struct{
        int *a ;
    } x ;
    int y[size] ;
    . . .
    x.a = &y[j] ;

```

The rules applied in this case are as follows:

```

kill = { (x.a,x1,rel) | (x.a,x1,rel) ∈ input }
gen = { (x.a,y, first_clem([j])) }
return( gen ∪ (input - kill) )

```

Consider the following example:

```

main() {
    struct{
        int *f1 ;
    } x, *z ;
    int y[70], w ;

    z = &x ;           /* stmt 1 */
    (*z).f1 = &w ;      /* stmt 2 */
    x.f1 = &y[0] ;       /* stmt 3 */
}

```

Case 1.8 appears at statement 3. Using the given rules, the sets related to this statement are as follows:

```

input = {(z,x,D), (x.f1,w,D)}
kill = {(x.f1,w,D)}
gen = {(x.f1,y,D)}
output = {(z,x,D), (x.f1,y,D)}

```

**Case 8.4:** The general format of this case is:

```

struct foo{
    int *a ;
} *x ;
int *py ;
. . .
(*x).a = &(*py)[j] ;

```

The rules applied in this case are as follows:

```

kill = { (x1.a,x2,rel) | (x,x1,D), (x1.a,x2,rel) ∈ input }
gen = { (x1.a,y1,rel1 ∞ rel2 ∞ first_elem([j])) | (x,x1,rel1), (py,y1,rel2) ∈ input }
changed_input = (input - { (x1.a,x2,D) | (x,x1,P), (x1.a,x2,D) ∈ input })
                ∪ { (x1.a,x2,P) | (x,x1,P), (x1.a,x2,D) ∈ input }
return( gen ∪ (changed_input - kill) )

```

Consider the following example:

```

f(float b, int i) {
    struct{
        float *a ;
    } *x, z ;
    float *py, w[70] ;

    x = &z ;          /* stmt1 */
    if (b == 0) ;
        z.a = &b ;    /* stmt2 */
    py = &w[0] ;      /* stmt3 */
    (*x).a = &py[i] ; /* stmt4 */
}

```

Case 8.4 appears at statement 4. Since `py` is a pointer to an array, we treat `&py[i]` as `&(*py)[i]`. Using the given rules, the sets related to this statement are as follows:

```

input = {(x,z,D), (z.a,b,P), (py,w,D)}
kill = {(z.a,b,P)}
gen = {(z.a,w,P)}
changed input = input
output = {(x,z,D), (z.a,w,P), (py,w,D)}

```

### 5.3.1 Interprocedural Points-to Analysis for Recursive Data Structures

In this section, we explain the interprocedural analysis in the case of a recursive data structure. The major point is that, when an invisible variable, say `1_x`, is of recursive data structure type, all the instances resolved from `1_x` are represented by `1_x`. This compact representation can result in some imprecision. This is shown in the example given in Figure 5.8 Readers can skip the rest of this section without losing the continuity.

<pre> typedef struct foo{     int      data ;     struct foo *next ; } F00 ;  F00 *a, d ;  main() {     F00 b, c ;     a = &amp;b ;     b.next = &amp;c ;     f() ; } f() {     a-&gt;next-&gt;next = &amp;d ; } </pre>	<pre> typedef struct foo{     int      data ;     struct foo *next ; } F00 ;  F00 *a, d ;  main() {     F00 b, c ;     a = &amp;b ;           /* stmt 1 */     b.next = &amp;c ;       /* stmt 2 */     f() ;              /* stmt 3 */ } f() {     F00 *temp0 ;     temp0 = (*a).next ; /* stmt 4 */     (*temp).next = &amp;d ; /* stmt 5 */ } </pre>
(a) The C-program	(b) The simplified version

Figure 5.8: An example of interprocedural points-to analysis in the presence of recursive data structures.

The points-to information at statement 3 of Figure 5.8 (before the function call) is shown in Figure 5.9. The points-to information are shown in the following three different formats: (i) abstract stack representation, (ii) memory representation, and (iii) points-to triples.

The map process should pass the information concerning the global variables to the function `f`. From the relationship  $(a, b, D)$ , we resolve  $(a, a.l, D)$  for function `f` (because `b` is not in the scope of `f`). From the relationship  $(b.next, c, D)$ , we resolve

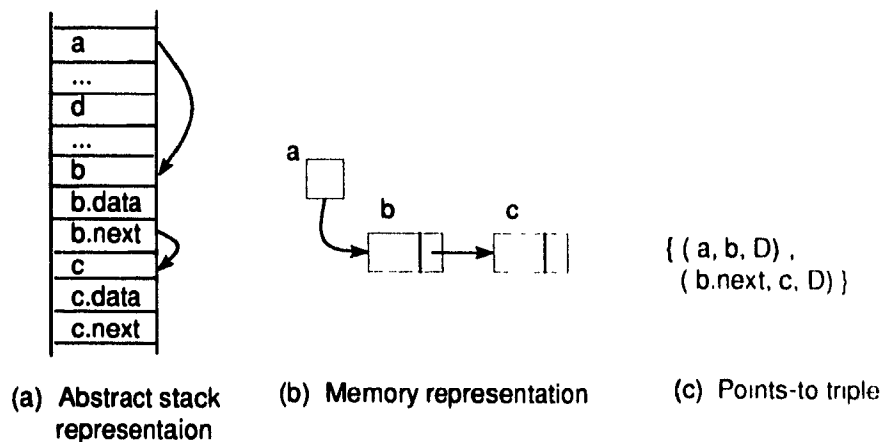


Figure 5.9: The points-to information of function `main` before the map process

the relationship  $(1\_a.next, 1\_a.1\_next, P)$ . Note that as variable `c` is not in the scope of function `f`, we have incorporated the new invisible variable  $(1\_a.1\_next)$ . The reason that the definitely-points-to relationship in caller `main` is changed to possibly-points-to relationship in callee `f` is that the location `1_a` is of recursive structure type (explained in Section 5.1.3), and, consequently, has the potential of standing for more than one stack location. As a result, the points-to relationships related to this type of location must be possibly-points-to relationships. Figure 5.10 shows the corresponding points-to information. Figure 5.10(b) shows the way that we consider the memory locations. This is called as the ‘simulation of memory location’ where the word ‘simulation’ reflects the fact that it is different from the way that the actual memory looks like.

Statement 4 resolves the relationship  $(temp0, 1\_a.1\_next, P)$ . Statement 5 refers to location  $(*temp).next$  and location  $*temp$  refers to `1_a.1_next`. As it is shown in Figure 5.11(a), this location does not have any other fields. However, as this location is related to the recursive data structure `1_a`, it will take the `next` field corresponding to `1_a`. That is why we get the relationship  $(1\_a.next, d, P)$  (the reason for having a possibly-points-to relationship is already explained). The points to information after finishing the process of function `f` and before unmap process is shown in Figure 5.11.

Next, we apply the unmap process. Since locations `1_a` and `1_a.1_next` stand for variables `c` and `b`, respectively (the corresponding information is saved in the `map-info` and is represented in Figure 5.10(d)), we get the relationships  $(b.next, c, P)$  and  $(b.next, d, P)$ . The location `1_a.next` also stands for location `1_a.1_next.next` (because it is a recursive data structure). This fact resolves the relationships  $(c.next, c, P)$

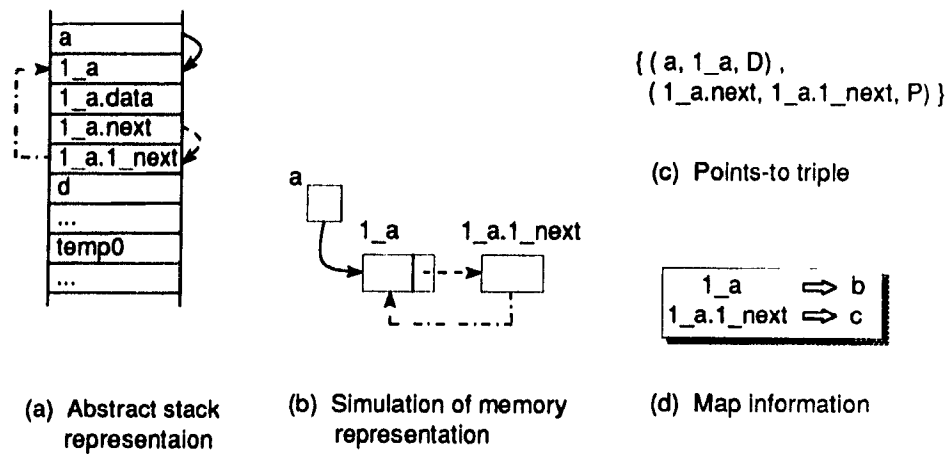


Figure 5.10: The points-to information of function  $f$  after the map process

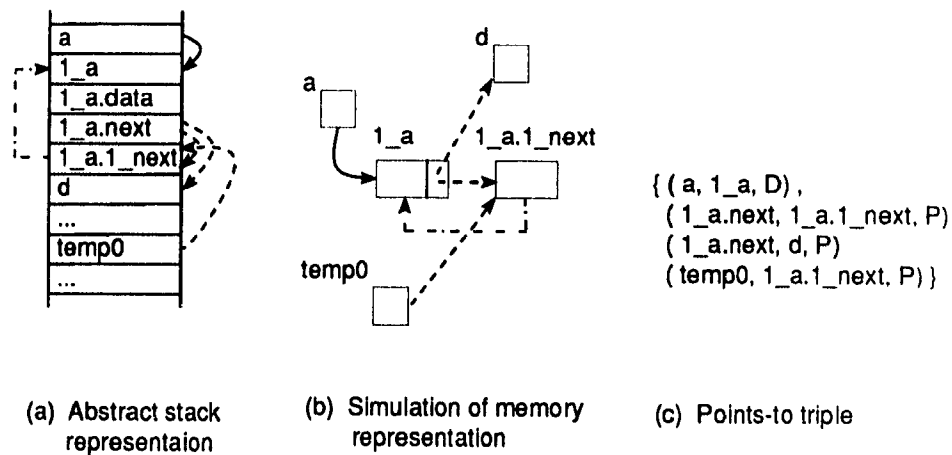


Figure 5.11: The points-to information of function  $f$  before the unmap process.

and  $(c.next, d, P)$ . The final points-to information of function `main` is shown in Figure 5.12.

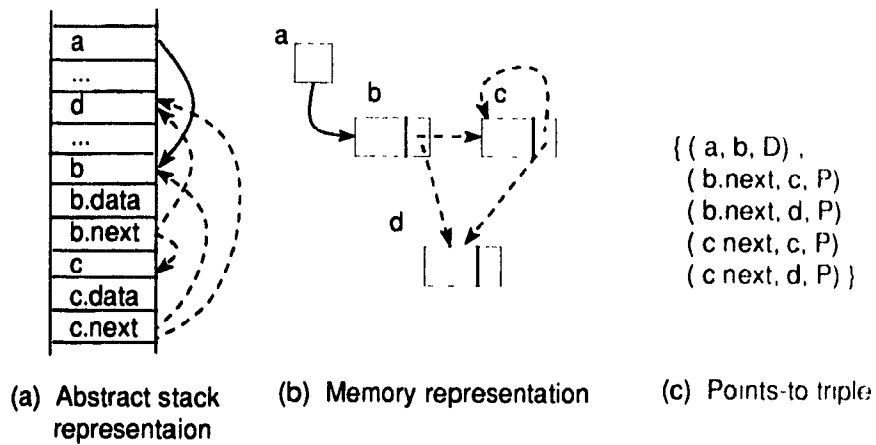


Figure 5.12: The points-to information of function main after the unmap process



## Chapter 6

# Handling other Pointer Features in C

In the previous chapters, we discussed points-to analysis in detail. In the C-language there are many other kinds of pointer manipulation that can effect points-to analysis. In this chapter, we briefly discuss these cases and give a simple solution for them. These solutions have been implemented to enable us to perform experiments on a reasonable set of benchmarks. In Section 6.1, we discuss the pointer arithmetic, in Section 6.2 we explain our analysis for type casting, and in Section 6.3, we consider heap-allocated data structures.

### 6.1 Pointer Arithmetic

The C-language allows arithmetic operations on variables having pointer-type. Here is a simple example of this case:

```
int *a, i ;  
a = a + i ;
```

In most cases, instead of increasing the array index, a pointer to array is increased. In other words, arithmetic operations are used to access different elements of an array. Due to this fact, we treat the arithmetic operations achieved on the pointer to arrays differently. We basically divide different cases into the following two basic groups.

1. The arithmetic operation involves a pointer to a variable which is not of array type. In this case, there is no limitation in the corresponding arithmetic operations. The only potential problem is that we may access the wrong memory location. Since there is no consistent way of following the variables after the arithmetic operations, we use the worst case assumption. This assumption says that the pointer variable has a possibly-points-to relationship with all the variables of its scope.
2. The arithmetic operation involves a pointer to a variable of array type. If the pointer **x** points-to the array **a**, the corresponding relationship is divided to the following two subgroups:
  - (a) **x** definitely-points-to **a**. This case happens when location **x** points to the first element of array **a**. If we have an arithmetic operation on **x**, then **x** does not point to the first element any more. In this case, the definitely points-to relationship should be changed to possibly-points-to. The following is an example:

```
main() {
    int *x, a[7] ;
    x = &a[0] ; /* stmt1 */
    x = x + 1 ; /* stmt2 */
}
```

After processing statement 1, we get the relationship (**x.a.D**) as the result while processing statement 2 results in the relationship (**x.a.P**).

- (b) **x** possibly-points-to **a**. This case happens when location **x** points to one of the elements of the array **a** (not the first one), or when we have a control statement. Obviously, after performing an arithmetic operation on **x** in either of these two cases, we still do not know which element of array **a** is pointed to by **x**. This means that the corresponding relationship remains unchanged. Here is an example of this case:

```
main() {
    int *x, a[7], i ;
    ...
    x = &a[i] ; /* stmt1 */
    x = x + 1 ; /* stmt2 */
}
```

After processing statement 1, we get the relationship (**x.a.P**) as the result. The process of statement 2 does not make any difference in the final result.

In accessing the arrays, we assume that the index expression is within the corresponding boundaries.

## 6.2 Type Casting

Type casting in the C-language allows one to assign variables of different types to each other. Since the type of a basic statement is a determining factor in deciding whether that statement should be considered in the points-to analysis or not, type casting plays an important role in this analysis. In the C-language, the programmer is allowed to change the types with no limitation. This flexibility of the C-language necessitates special precautions in the analysis.

Any type casting appearing as an argument in a function call is removed by the simplifier program (by using a temporary variable). Thus all arguments match the type of their corresponding formal parameter. Furthermore, any type casting appearing in the left hand side of an assignment statement is moved to the right hand side of the assignment. As a result, type casting is an issue only in the intraprocedural analysis.

Define the *pointer-level* of a variable  $x$ , denoted as *pointer-level*( $x$ ), as the maximum number of indirect references that  $x$  may have. For example, the pointer-level of  $x$  and  $y$  with types `float***` and `int` are 3 and 0, respectively. The points-to relationship ( $x, y, rel$ ) is devised in such a way that the type of the variable  $x$  is always a pointer to the type of the variable  $y$ . In other words:

$$\text{pointer-level}(x) = \text{pointer-level}(y) + 1, \text{ and, } \text{type}(*x) = \text{type}(y)$$

The only exception is in the presence of type casting, where one may have points-to relationship between variables of different type and pointer-level. In this case assuming a points-to relationship ( $x, y, rel$ ), any of the following relationships between  $x$  and  $y$  can appear:

- $\text{pointer-level}(x) = \text{pointer-level}(y) + 1$ , and,  $\text{type}(*x) \neq \text{type}(y)$

There might be a points-to relationship between the variables of correct difference on pointer-level, but with different base-type. Consider the following example:

```

typedef struct{
    int a ;
} F00 ;

float y ;
F00 *x ;
x = (F00 *) &y ; /* stmt 1 */

```

Here, **x** has the type **F00\*** while **y** has the type **float**. After processing statement 1, we get the relationship (x,y,D).

- **pointer-level(x) = pointer-level(y)**

There might be a points-to relationship between the variables of the same pointer level. Consider the following example,

```

int *x
float *y ;
x = (int*) &y ; /* stmt 1 */

```

Here, pointer-levels of **x** and **y** are equal to 1. After processing statement 1 we get (x,y,D).

- **pointer-level(x) < pointer-level(y)**, or, **pointer-level(x) > (pointer-level(y) + 1)**  
There might be a points-to relationship between the variables of different pointer level. Consider the following example,

```

int *x, **y ;
x = (int*) &y ; /* stmt 1 */

```

Here, **x** has the type **int\*** and **y** has the type **int\*\***. After processing statement 1, we get the relationship (x,y,D).

There are special cases of type casting that are more complicated to handle. These cases happen when user forces a base-type (e.g. **int**, **float**, etc.) variable to get the address of another variable. In other words, user pushes a non-pointer variable to get the address of another variable. Following shows an example of this case

```

main() {
    int a, b ;
    a = (int) (&b) ;
}

```

In this example, we should get (a,b,D). At present, we do not handle statements which have a lhs of scalar type. Our analysis can be easily extended to take care of this case, however, we are not sure if it is worthwhile.

### 6.3 Dynamic Data Structures

In this section, we discuss the last possible case that results in a points-to relationship. This is the points-to information obtained from the *heap allocation*. There are some library functions in the C-language which get a location of a given size from the heap and assign it to a variable during the execution time. The following is an example

```
int *x ;
x = (int*) malloc (sizeof(int)) /* stmt 1 */
```

After the execution of statement 1, a heap location of size 'integer' will be assigned to the variable **x**. This means that **x** is pointing to a location in the heap. As there is no consistent way for using the heap, it is considered as a bunch of memory locations with no specific type which can be pointed to by any type of variable; or can point to any type of variable. Figure 6.1 is devised to explain our way of looking at the heap

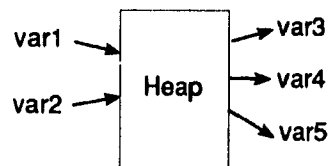


Figure 6.1: An overview of the abstract heap representation.

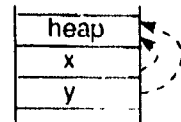
To incorporate heap into our analysis, we define a single location in the abstract stack, called **heap**, which has no specific type. This location represents the whole heap. This approach of approximating the heap as one location introduces conservative approximations for relationships within the heap. However, these relationships can be further refined by a heap points-to analysis that is run after the stack points-to analysis [Ghi93]. As we have found that it is very rare to have pointers back from the heap to the stack, this division into stack and heap analysis phases is very reasonable. All the points-to relationships dealing with **heap** location are of type possibly-points-to. The reason is that we do not exactly know which location of heap is considered. This idea is further explained using the example given in Figure 6.2(a).

```
typedef struct foo{
    int a ;
    int *b ;
} F00 ;
```

```
void main()
{
    F00 *x ;
    int *y ;

    x = (F00*) malloc (sizeof(F00)) ; /* stmt1 */
    y = & (*x).a ;                    /* stmt2 */
}
```

(a) The C program



(b) The related abstract stack

Figure 6.2: An example of dynamic data allocation.

In this example, statement 1 allocates some memory locations from heap. As the result, we get the relationship (x,heap,P). The  $\&(*x).a$  at statement 2 is a location in the heap. Due to this fact, after processing this statement, we get the relationship (y,heap,P). The final stack representation of this program is shown in Figure 6.2(b). Functions causing the heap allocations are: **malloc**, **calloc**, **valloc**, **realloc**, **memalign**, **mallopt**, and **alloca**. The same process as explained above has to be done for them.

## Chapter 7

# Implementation Details and Limitations

In this chapter, we explain some details of our implementation. This is composed of our data structure for the abstract stack representation in Section 7.1, some special features of our analysis in Section 7.2, some details of map and unmap processes in Section 7.3, and finally the library functions in Section 7.4.

### 7.1 Abstract Stack Data Structure

The concept of an ‘abstract stack’ is introduced in the previous chapters. In this section, we explain its actual data structure and the reasoning behind this implementation.

For several reasons which will be explained later, we use a bit matrix data structure for our abstract stack representation. As a bit matrix requires an array of a fixed size, we first pass through the program to collect the variables corresponding to each function. To minimize the size of the bit matrix, we collect only the variables which are involved in points-to analysis. These are: (i) the actual/formal parameters of pointer types, and (ii) the variables appearing in the lhs/rhs of the assignment statements represented in Table 5.1. Each function has its own bit matrix where the size of this matrix depends on the number of collected variables. If a global variable  $x$  is one of the collected variables of any of the functions, it should be also included in the collected variables of all other functions (because it is indirectly accessible by those functions). After collecting the variables for each function, we perform the points-to

analysis starting from the `main` function. This analysis is composed of the following steps:

1. Building the points-to bit matrix for the function `main` using the collected variables,
2. Calling the 'points-to' function (given in Chapter 4).

Each index of a bit matrix represents a variable in the related function. The global variables keep the same index in all the bit matrices. There are two types of bit matrices, one for definitely-points-to relationships and the other one for possibly points-to relationships. If the  $(i, j)$ 'th element of the definitely-points-to bit matrix is equal to 1, it means that the variable corresponding to the  $i$ 'th element has a definitely-points-to relationship with the variable corresponding to the  $j$ 'th element. If the content of the  $(i, j)$ 'th element in both definitely-points-to and possibly points-to bit matrices is equal 0, it means that the variables related to  $i$  and  $j$  have no relationship.

Figure 7.1 shows a graph representation of the corresponding data structure. Since dynamically allocated functions (like `malloc`) are costly, the whole memory required for the complete data structure (as represented in Figure 7.1) is computed and allocated only once. The necessary links are computed by performing some arithmetic operations involving sizes of the fields.

The following are the advantages of using the bit matrix data structure

- **Memory Space:** As a single bit is used to indicate if two variables are related or not, we are using the minimum space for our representation. For example, if there are 60 variables in a function which are involved in points-to analysis and an integer is composed of 32 bits, the related bit matrices are two arrays of integer type with 60 rows and 2 columns (this can be further improved to a 2 by 2 matrix).
- **Speed:** These operations are fast and efficient because they are mainly composed of bit operations. Some examples are given in the following:
  - The merge of two matrices is the 'logical or' of the two.
  - One may need to access the  $(i, j)$ 'th element of a bit matrix. In this case, index  $i$  provides a bit vector to look for the index  $j$ . The integer part of the ratio of  $j$  to the number of bits per word (in this case 32) gives the actual column number for  $j$ . The remainder in dividing  $j$  by 32 gives the bit number in that column. For example, for  $j = 65$ , the column number is



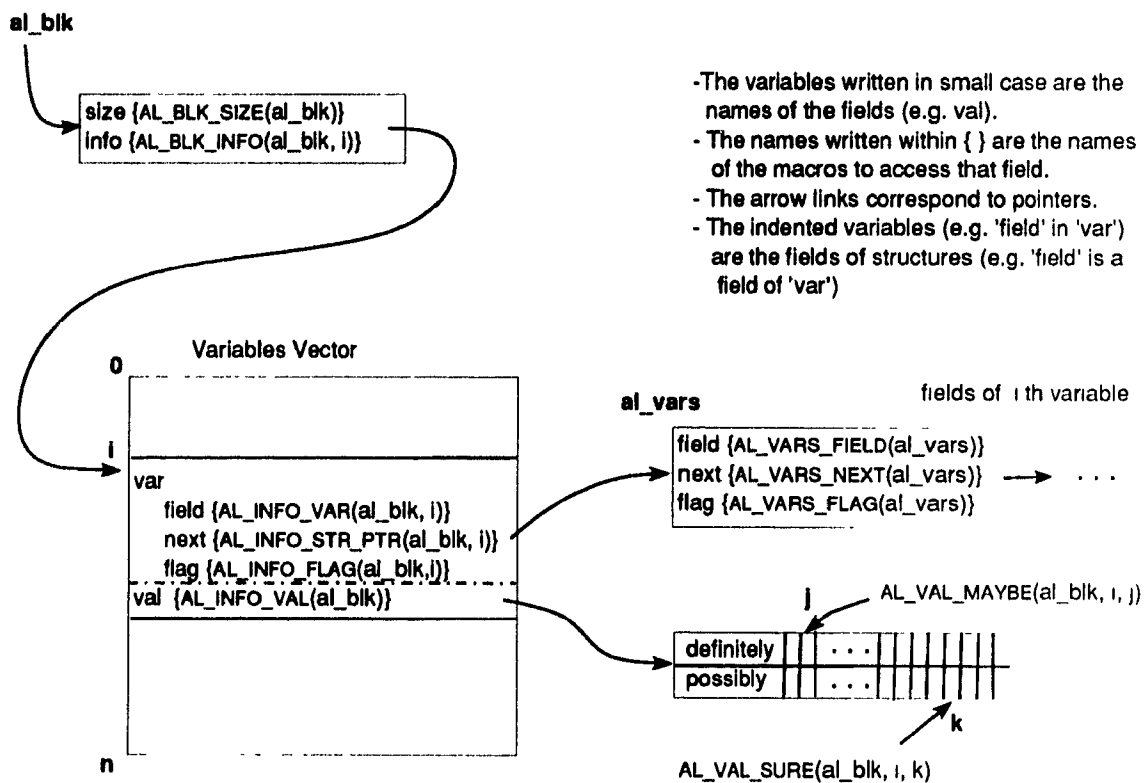


Figure 7.1: The data structure of the abstract stack.

2 and the bit number in 1. This means that the first bit in (1,2)th element of the corresponding bit matrix is demanded. Masking is used to access a given field. An array of masks is defined to speed up the accessing process. The size of this array (maximum number of masks) is equal to the number of bits per integer (in this case 32).

- To determine if \*a and \*b refer to the same locations, one needs only to know if two array pointers are pointing to the same array(s). In this case, a logical 'xor' operand is used. If the result of 'xor' of the rows which are related to the pointer variables is 0, we conclude that the variables are pointing to the same location(s). This is particularly useful for the array dependency module that needs to know if two array pointers point to the same array
- **Accessibility:** Since we need to frequently check the points-to information, it is important to have a fast access from a variable to its index in the bit matrix and vice versa. In our case, this is quite fast because there is a correspondence between each index of the bit matrix and a variable node in the tree.

## 7.2 Some Special Features of Our Analysis

- As the result of points-to analysis is used by other analyses, the related information should be saved for later use. The corresponding bit matrices are saved at the statement level. If a function is called more than once, to save some space, the information from different calls is first merged together and then the result is saved. Therefore, the possibly-points-to information may be due to the merge between two different calls and not due to a loop or to a condition. If space were not an issue, one could easily save the points-to information for each statement separately (each containing the related invocation graph node). We are planning to further investigate this issue in our future work. Presently, the points-to information can be used in one of the following two ways:
  1. Other analyses are done simultaneously with the points to analysis. In this way, those analyses obtain the precise points-to information. The disadvantage of this approach is that it is more complicated from the implementation point of view.
  2. Other analyses are done after the points-to analysis is completed. The advantage of this approach is that the implementation is straight forward. Its disadvantage is the possibility of getting more possibly-points to relationships.
- As mentioned before, the points-to information is saved for each statement. The sequence of statements that do not affect the points-to analysis have the same points-to information. To save some memory space, this sequence of statements point to the same bit matrix.
- Our method can perform some correctness checking for the program. This is normally done at run-time. These checks produce some error/warning messages to the user. For precise checking, we initialize all the pointer variables to point to NULL at the beginning of each function. The following is the list of the checks which are performed:
  1. If the user applies an indirect reference operand to a variable which definitely points to NULL, an error message together with the corresponding line number is reported and that line is ignored. The following are two examples of this case:

```
main() {
    int **a, c ;
    *a = &c ;
}
```

```
main() {
    int **a, c ;
    if (...)
        *a = &c ;
}
```

In these examples an error message appears for statement `*a = &c`. The error occurs because `a` does not point to any variable (it definitely points to NULL) and, consequently, `*a` does not exist.

If the indirect reference operand is applied to a variable that possibly points to NULL, it is not guaranteed to be an error. To clarify this point, consider the following example:

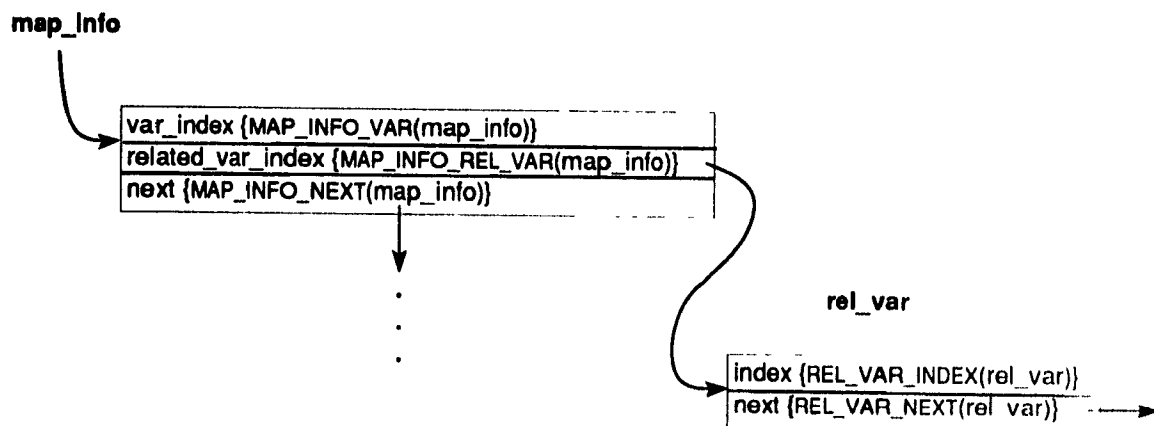
```
main() {
    int **a, *b, c ;
    if (...)
        a = &b ;
    if (...)          /* s1 */
        *a = &c ;    /* s2 */
}
```

At `s1`, we get `(a,b,P)`, `(a,NULL,P)`. Although at `s2`, `*a` might be NULL, but since this is not definite, no message is given.

2. If there is an arithmetic operation on a pointer variable which is not pointing to an array, the worst case assumption is made, that is, the variable points to all other variables of its scope. As such arithmetic operation might be due to a programming error, a warning message is generated.
- Since system commands are not part of the C-language, they are not supported by our analysis, e.g., `exit()`. This means that our analysis may be overly conservative since we could include non-realizable paths after the `exit()`.
  - *Union* type variables are not considered in the present version of our analysis. However, since these are an extension to structures, a possible solution is given in Section 5.1.4.
  - Functions with a variable number of parameters are not considered

### 7.3 Some Implementation Details of Map and Unmap Processes

As it was mentioned in Chapter 4, the map information (`map_info`) is used to save the relationship between the invisible variables of callee and the local variables of caller. The data structure of `map_info` is a two dimensional linked list which is represented in Figure 7.2.



- The variables written in small case are the names of the fields (e.g. `var_index`)
- The names written within `{ }` are the names of the macros to access that field
- The arrow links correspond to pointers.
- Following is a description of 'map\_info' (map information) fields:
  - `var_index` : is the index of an invisible variable in callee
  - `related_var_index` : is a pointer to the variable(s) in caller which are represented by 'var\_index'
  - `next` : is the pointer to the next invisible variable
- Following is a description of 'rel\_var' (related variables) fields.
  - `index` : is the index of a variable in caller
  - `next` : is the pointer to the next variable in caller

Figure 7.2: The data structure of map information (`map_info`)

Any interprocedural analysis that uses the points-to information, may need to access the invisible variables. Therefore, we need to save the map information. Following shows an example of constant propagation where the map information is used.

```

main()
{
    int *a, b ;

    a = &b ;
    f(a) ;
}
void f(int *x)
{
    *x = 1 ;      /* stmt 1 */
}

```

Since at statement 1 we have  $(x, 1_x, D)$ , after constant propagation  $1_x$  is set equal to constant 1. After returning to the `main` function, we should conclude that `b` is equal to constant 1. This can be done only if we know that  $1_x$  is equivalent to `b`.

The map information (`map-info`) related to each function call is saved in the related node in the invocation graph. This information can be easily used by other analyses.

## 7.4 Library Functions

All the library functions except those listed in the following are side effect free

`malloc`, `calloc`, `valloc`, `realloc`, `memalign`, `mallopt`, `alloca`

The above functions generate a points-to relationship between variables and the heap which is discussed in Chapter 5. Note that function `alloca` allocates a location from stack and releases it after returning from the function. In this case, we still give the points-to relationship to heap, because it does not change the generality of the heap analysis. Note that we do not handle `memcpy` because we believe it is rarely used to affect the points-to information. However, we will send a message to the user. If this case happens frequently, we can easily handle it.

## Chapter 8

# Experimental Results and Practical Uses of Points-to Analysis

In this chapter, we demonstrate the effectiveness of our points-to analysis by providing some experimental results. We also discuss how other analyses implemented in the framework of the McCAT compiler benefit from our points-to information. The experimental results obtained by running the analysis on some standard benchmarks are presented in Section 8.1. In Section 8.2, we discuss other analyses which are currently using the results of our analysis.

### 8.1 Experimental Results

This section contains our experimental results on some standard benchmark. Table 8.1 contains the following information about the benchmark files:

1. **Benchmark:** is the name of the benchmark
2. **Lines:** is the number of lines in the benchmark including all the comment line
3. **#of stmt in SIMPLE:** is the total number of C-statements in the SIMPLE representation, excluding the comments
4. **#of func:** is the number of functions in the benchmark
5. **Min #of var:** is the minimum number of variables among all abstract *fact*. (this also includes the invisible variables and all the fields in case of a structure.)

6. Max #of var: is the maximum number of variables among all abstract stacks (this also includes the invisible variables and all the fields in case of a structure)
7. Description: is a brief description of the benchmark

Note that each function has its own bit-matrix where the size of the matrix depends on the total number of variables involved in the points-to analysis. The columns 'Min #of var' and 'Max #of var' give an idea of the size of this matrix. The largest bit-matrix size for the given benchmarks is equal to 187 (for the program 'config'). Assuming that each word is composed of 32 bits, only six words are needed to store one row of this bit matrix. Seven programs need a maximum of one word, five need two words, and the rest need four to six words. The smallest size of the bit-matrix is one word for fourteen of the benchmarks and two words for the remaining three. Considering that these numbers include all the invisible variables and all the possible combinations of structures, the required memory size is quite small.

All pointer type variables are initialized to NULL. This initialization increases the number of points-to relationships. Since these relationships are not necessarily initialized by the user, the points-to information contributed by this initialization is not counted. The relevant statistics is provided in Tables 8.2 and 8.3. We first analyze Table 8.2 which essentially contains the number of indirect references in a program. This table represents the actual use of points-to information and contains the following items (from left to right):

1. Benchmark: is the name of the benchmark
2. #of ind ref for Def rel: is the number of indirect references that have definitely points-to relationship.
3. #of ind ref for 1 Pos rel: is the number of indirect references that have only one possibly-points-to relationship with other variables.
4. #of ind ref for 2 Pos rel: is the number of indirect references that have two possibly-points-to relationships with other variables.
5. #of ind ref for 3 Pos rel: is the number of indirect references that have three possibly-points-to relationships with other variables.
6. #of ind ref for  $\geq 4$  Pos rel: is the number of indirect references that have four or more, possibly-points-to relationships with other variables.
7. #of ind ref: is the total number of indirect references in the benchmark
8. Avg: is the average number of variables that an indirect reference is pointing to. This is computed by dividing the number of definitely/possibly-points-to

Benchmark	Lines	#of stmt in SIMPLE	#of func	Min #of var	Max #of var	Description
genetic	506	449	18	28	54	A genetic algorithm implementation to test sorting
swap1	25	25	3	3	8	Indirectly swaps two pointer variables with no condition
swap2	30	28	3	3	8	Indirectly swaps two pointer variables with condition
dry	826	206	13	21	43	Dhrystone benchmark
intmm	71	44	5	4	10	Integer matrix multiplier
clintpack	1231	919	11	10	106	The C version of the famous Fortran benchmark Linpack
config	2279	4531	52	18	187	Checks all the features of the C language
toplev	1637	811	31	48	97	One of the files of GNU C free software. It is the top level of ccl. It parses command lines, opens files, invokes the various passes in the proper order, and records the time used by each. Error messages and low level interface to malloc are also handled here
compress	1923	1275	11	41	176	UNIX utility program for compress and uncompress of the files
mway	699	871	23	51	125	A unified version for all best algorithms for mway partition
hash	256	110	5	15	30	An implementation of a hash table
misr	276	235	5	10	44	This program creates two MISR - one which contains the true outputs and the other in which the outputs are not corrupted with the probability given in the input. The values of the MISR's are compared to see if the introduced error have canceled themselves
xref	146	137	7	25	60	This program is copied from advanced C programming on the IBM pc. This cross reference program builds a tree of items
stanford	885	848	48	31	44	Stanford baby benchmark suite from John Hennessy. This file includes: Perm (permutation generator), 'Intmm' (integer matrix multiplier), 'Mm' (matrix multiplier for real numbers), 'Queen' (8 queens problem), 'Tower' (tower of Hanoi - recursive program), 'Puzzle', 'Bubble' (sort), 'Quick' (sort), 'Tree' (sort, using dynamic data structure), 'Ocar' (Fast Fourier Transform)
fixoutput	400	382	5	15	29	A simple translator
allroots	225	245	6	2	8	Find all the roots of a polynomial

Table 8.1: Characteristics of benchmark files.



relationships by the number of indirect references. For example, this number for the 'genetic' benchmark is equal to:  $(5+33+2*2+2*3)/42=1.11$ .

An indirect reference can appear in the form of  $*x$ ,  $(*x).y.z$ , and  $x[i][j]$  (when  $x$  is a pointer to array). We have collected three rows of information for each benchmark. The first row in each case is related to the indirect references of the form,  $*x$  and  $(*x).y.z$ . The second row in each case is related to the semantically indirect references of the form  $x[i][j]$  ( $(*x)[i][j]$ ). This information is particularly important in the array analysis. The third row in each case is the sum of the first two rows. The results in this table are generally quite impressive. This indicates the effectiveness of our point-to analysis. Following are some specific conclusions derived from this table.

- The average number of variables that an indirect reference is pointing to is quite close to one, where one indicates the best possible case. The overall average is equal to 1.04.
- Overall, 47.10% of the points-to relationships are definitely-points-to.
- The number of indirect references of the form  $x[i][j]$  (when  $x$  is a pointer to an array) that have a definitely-points-to relationship, plays an important role in array dependence analysis. For most of the benchmarks, specially "linpack" and "stanford", this is a large number compared to the total number.
- There is no case with more than four possibly-points-to relationships and there is only one case with three possibly-points-to relationships.

Table 8.3 contains the general points-to information. The numerical results represent the sum of the points-to information for all the program points after our analysis is completed. This table is composed of the following items (from left to right):

1. Benchmark: is the name of the benchmark
2. Total: is the total number of points-to relationships of all the statements in the corresponding benchmark
3. Max per stmt: is the maximum number of points-to relationships per statement
4. Avg per stmt: is the average number of points-to relationships per statement
5. From parm: is the number of points-to relationships from a parameter to a variable
6. To parm: is the number of points-to relationships from a variable to a parameter
7. From invi var: is the number of points-to relationships from an invisible variable to a variable

Bench- mark	#of ind ref for Def rel	#of ind ref for 1 Pos rel	#of ind ref for 2 Pos rel	#of ind ref for 3 Pos rel	#of ind ref for $\geq 4$ Pos rel	Tot #of ind ref	Avg
genetic	0	6	0	0	0	12	1.11
	5	27	2	2	0		
	5	33	2	2	0		
swap1	8	0	0	0	0	8	1.00
	0	0	0	0	0		
	8	0	0	0	0		
swap2	4	0	4	0	0	8	1.50
	0	0	0	0	0		
	4	0	4	0	0		
dry	2	32	1	0	0	36	1.02
	11	0	0	0	0		
	13	32	1	0	0		
intmm	0	3	0	0	0	6	1.17
	2	0	1	0	0		
	2	3	1	0	0		
clinpack	7	0	0	0	0	150	1.03
	123	16	4	0	0		
	130	16	4	0	0		
config	9	33	0	0	0	42	1.00
	0	0	0	0	0		
	9	33	0	0	0		
toplev	0	0	0	0	0	15	1.00
	0	15	0	0	0		
	0	15	0	0	0		
compress	0	20	0	0	0	27	1.00
	0	7	0	0	0		
	0	27	0	0	0		
mway	31	14	0	0	0	74	1.05
	24	0	5	0	0		
	55	14	5	0	0		
hash	7	7	0	0	0	14	1.00
	0	0	0	0	0		
	7	7	0	0	0		
misr	1	8	27	0	0	36	1.69
	3	0	0	0	0		
	4	8	27	0	0		
xref	0	20	9	0	0	34	1.29
	0	2	0	0	0		
	0	22	9	0	0		
stanford	6	35	0	0	0	101	1.02
	61	0	2	0	0		
	67	35	2	0	0		
fixoutput	5	1	0	0	0	6	1.00
	0	2	0	0	0		
	5	3	0	0	0		
allroots	0	0	0	0	0	12	1.55
	0	18	24	0	0		
	0	18	24	0	0		

Table 8.2: Statistics for the number of indirect reference access in a benchmark. The first line in each case is related to cases like  $*x$  and  $(*x).y.z$ . The second line in each case is related to  $x[i][j]$  when  $x$  is a pointer to array. The third line is the sum of the first two lines.

8. To invi var: is the number of points-to relationships from a variable to an invisible variable
9. From struct: is the number of points-to relationships from a structure to a variable
10. To struct: is the number of points-to relationships from a variable to a structure
11. From heap: is the number of points-to relationships from heap to a variable
12. To heap: is the number of points-to relationships from a variable to heap

It is seen that sometimes we get large values. This is partially because the same information is counted for many statements. Here are some comments on these numbers:

- The substantial number of points-to relationships from a parameter to a variable emphasizes the necessity of the interprocedural point-to analysis.
- An interesting observation is that we have not encountered a single instance where a pointer points to a parameter variable.
- Another interesting observation is that there are very few cases of an invisible variable pointing to other variables. The few encountered cases happen when one has a parameter (or global variable), say  $x$ , which is a multi-level pointer type and  $*x$  is an invisible variable which points to another variable. Unlike this case, our statistics shows that it is quite probable that a parameter (or global variable) points to a variable which is not in the scope of that function. This conclusion is derived from the numbers in the 'To invi var' column in Table 8.3.
- There are not many relationships from/to variables of structure type to other variables.
- Another important result is that there are no points-to relationships from a dynamically-allocated location (we use the location 'heap' for all of them) to any stack-resident variable. This means that pointers in heap-allocated objects only point to other heap-allocated objects. This observation supports our approach of separating the points-to analysis of stack-allocated and heap-allocated data structures, and developing different abstractions for the two cases. Further, the substantial number of points-to relationships to heap, underlines the benefits achievable from a powerful heap analysis.

Bench- mark	Total	Max per stmt	Avg per stmt	From parm	To parm	From invi var	To invi var	From struct	To struct	From heap	To heap
genetic	339	9	—	29	—	—	—	—	—		1066
	1889	12	4	99	—	—	—	—	—		
swap1	60	5	2	8	—	8	19	—	—		
	—	—	—	—	—	—	—	—	—		
swap2	26	2	—	8	—	—	8	—	—		
	59	4	2	—	—	8	11	—	—		
dry	294	7	1	102	—	—	68	—	—		97
	814	13	3	61	—	—	27	—	—		
intmm	55	4	1	16	—	—	—	—	—		
	66	3	1	24	—	—	—	—	—		
clinpac	13394	66	14	1299	—	—	1307	—	—		
	4907	29	5	207	—	—	665	—	—		
config	33030	24	7	91	—	—	91	—	—	136 200	18
	96196	95	21	—	—	—	—	—	—		
toplev	996	5	1	—	—	—	—	—	—		6
	2799	12	3	—	—	—	—	—	—		
compress	4815	14	3	—	—	—	—	—	—		1031
	17694	66	13	511	—	—	630	—	—		
mway	10708	48	12	967	—	—	1044	—	—		
	5733	28	6	78	—	—	173	—	—		
hash	405	12	3	63	—	—	90	—	—		199
	371	10	3	—	—	—	—	—	—		
misr	780	12	3	229	—	—	236	—	—		507
	1046	13	4	18	—	128	114	165	—		
xref	1	1	—	—	—	—	—	—	—		
	45	1	—	25	—	—	—	—	—		
stanford	654	5	—	538	—	—	235	—	—		311
	489	6	—	115	—	—	—	—	—		
fixoutput	741	4	1	—	—	—	—	—	—		786
	3116	12	8	—	—	—	—	—	—		
all	164	2	—	159	—	—	159	—	—		595
	605	4	2	420	—	—	210	—	—		

Table 8.3: General statistics of the points-to analysis. The first line in each case contains the statistics of the definitely-points-to information while the second line contains the statistics of the possibly-points-to information.

## 8.2 Practical Applications of Points-to Analysis

Points-to analysis acts as the initial step for many other analyses performed by the McCAT compiler and improves their accuracy. In the following, we briefly explain some of these analyses which are developed by other members<sup>1</sup> of our research group. We summarize the impact of accurate points-to information on their methods.

Section 8.2.1 discusses the replacement of indirectly referenced variables with their actual variable. Section 8.2.2 is on dependence analysis for the ALPIA representation [HGS92]. Section 8.2.3 is on extending the invocation graph (relatively points-to analysis) in the presence of function pointers parameter [Ghi92]. Section 8.2.4 is on generalized constant propagation [LH93]. Section 8.2.5 is on array dependence analysis [Jus93]. Section 8.2.6 is on reaching definition analysis [Sri92]. Section 8.2.7 is on live variable analysis [Sri92]. Section 8.2.8 is on loop unrolling which is implemented as a course project by M. Iarocci and C. Pateras.

### 8.2.1 Replacement of Indirectly-referenced Variables

In accessing an indirect reference variable (e.g.  $*a$ ), different analyses behave differently depending on having possibly-points-to or definitely-points-to relationship. To avoid redundant checking, we can replace an indirectly access variable, say  $*x$  with a variable  $y$  when they have the relationship  $(x, y, D)$ , and  $y$  is not an invisible variable. The fact that replacement is not done for an invisible variable does not mean that they can not be useful. As an example, they are useful for register allocator when the life time of keeping a variable in a register can become longer for definitely-points-to variables. They will be also useful in propagating any data flow information from callee to caller.

Table 8.4 represents some statistics on benchmarks of Table 8.1. The first column is the total number of indirectly reference pointer variables that have definitely-points-to relationship with any variable. While the second column represents total number of indirectly reference pointer variables that have definitely-points-to relationship with regular variables. This number shows how many cases of indirect references can be replaced by their points-to variable. One can increase this number by the use of function inlining<sup>2</sup>. This is due to the fact that callee has access to a local variable  $x$  of caller (which is not in the scope of callee) through a pointer  $px$ . If the function call from caller is replaced by its body, this limitation in scoping does not exist anymore.

---

<sup>1</sup>Except the method in Section 8.2.1 which is due to the author.

<sup>2</sup>Replacing a function call by its body.

Benchmark	#indirect ref to regular & invisible variables	#indirect ref to regular variables
genetic	5	5
swap1	8	4
swap2	4	0
dry	13	9
intmm	2	2
clinpack	130	0
config	9	0
topev	0	0
compress	0	0
mway	55	0
hash	7	0
misr	4	0
xref	0	0
stanford	67	51
fixoutput	5	5
allroots	0	0

Table 8.4: The statistic results for indirectly accessed variables.

### 8.2.2 Dependence Analysis for ALPHA

In this section, we briefly present the work of Hendren, Gao, and Sreedhar on ALPHA [HGS92]. ALPHA is an intermediate representation that allows one to perform sparse analyses. The first step in constructing ALPHA is to generate read/write sets for each statement using the points-to information. The read/write set specifies all the abstract stack location read/written at each statement in SIMPLE. The way to generate this information for pointer variables is to follow the links in the abstract stack of the points-to analysis. This is done using the rules given in Table 8.5 where  $s$  is a statement,  $PI$  is the points-to information,  $W_p(s, PI)/W_d(s, PI)$  is the possible/definite write at  $s$  with  $PI$  as input,  $R(s, PI)$  is the read set at  $s$  with  $PI$  as input,  $P_p(x, PI)/P_d(x, PI)$  is the set of variables that  $x$  definitely/possibly-points to in  $PI$ , and  $P(y, PI)$  is the set of variables that  $x$  definitely/possibly-points to in  $PI$ .

The example represented in Figure 8.1(a) shows a C-program, Figure 8.1(b) shows its SIMPLE representation, Figure 8.1(c) shows the related abstract stack, and Figure 8.1(d) shows the read/write set related to each statement. Now we have a closer look at statement 4. Clearly `temp0` will be in the read set. All the location(s) that `temp0` is pointing to will be in the write set, in this case variable `c`.

$s$	$W_p(s, PI)$	$W_d(s, PI)$	$R(s, PI)$
$x = y$	$\phi$	$\{x\}$	$\{y\}$
$x = \text{unop } y$	$\phi$	$\{x\}$	$\{y\}$
$x = y \text{ binop } z$	$\phi$	$\{x\}$	$\{y, z\}$
$*x = y$	$P_p(x, PI)$	$P_d(x, PI)$	$\{x, y\}$
$*x = \text{unop } y$	$P_p(x, PI)$	$P_d(x, PI)$	$\{x, y\}$
$*x = y \text{ binop } z$	$P_p(x, PI)$	$P_d(x, PI)$	$\{x, y, \text{tt } z\}$
$x = \&y$	$\phi$	$\{x\}$	$\phi$
$x = *y$	$\phi$	$\{x\}$	$\{y\} \cup P(y, PI)$

Table 8.5: Read and Write Sets for statement  $s$  relative to point-to information  $PI$

```

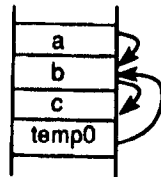
main() {
    int **a, *b, c ;
    a = &b ;
    b = &c ;
    **a = 7 ;
}

main() {
    int **a, *b, c ;
    int temp0 ;
    a = &b ;    /* stmt1 */
    b = &c ;    /* stmt2 */
    temp0 = *a ; /* stmt3 */
    *temp0 = 7 ; /* stmt4 */
}

```

(a) A C-program

(b) The SIMPLE representation



(c) Stack Matrix

$S$	$Write(S)$	$Read(S)$
$a = \&b$	$\{a\}$	$\phi$
$b = \&c$	$\{b\}$	$\phi$
$temp0 = *a$	$\{temp0\}$	$\{a, b\}$
$*temp0 = 7$	$\{c\}$	$\{temp0\}$

(d) Read/Write Sets

Figure 8.1: A simple example of dependence analysis.

Once the read/write set is computed for each statement, *exposure analyses* are performed. Exposure analyses, which consist of *willbe\_read*, *willbe\_written*, *was\_read* and *was\_written*, identify program points where  $\alpha$ -nodes are introduced. The  $\alpha$ -nodes essentially capture data-dependences crossing different control regions. Informally,  $\alpha$ -nodes are like *identity assignments* (e.g.  $x = x$ ). Once  $\alpha$ -nodes are inserted, dependence analysis is performed to generate dependence information and the resulting representation is called the ALPHA representation.

The accuracy of dependence information depends on the points-to information. Without points-to information, one has to make conservative assumptions regarding

the dependence information and this in turn generates a 'denser' ALPHA dependence representation.

### 8.2.3 Function Pointer Parameters

The presence of function pointers complicates the interprocedural analysis. Function pointers are special pointer variables which point to functions. These are referenced at run time to call the pointed-to functions. Unlike a normal function call, a call through a function pointer can not be bounded to a unique function at compilation time. The function invoked from such a call-site depends on the address contained in the function pointer when program execution reaches that point. Thus different functions can be invoked from the same call-site in different executions of the program. Under these circumstances, the invocation graph becomes a dynamic property of the program and can not be constructed by a simple pass through the program.

However, a precise and safe static approximation of the invocation graph can still be obtained using flow-analytic techniques. The key idea is that the set of functions invocable from a function pointer call-site is identical to the set of functions that the function pointer can point to at the program point just before the call-site. If the complete points-to information for each function pointer is available at the corresponding function pointer call-site, the invocation graph can be effectively constructed. In this case, different interprocedural analyses can be efficiently performed. To provide precise and useful information, points-to analysis itself needs to be interprocedural. The problem is how to build the invocation graph for points-to analysis itself.

The solution lies in updating the invocation graph while performing the points-to analysis. First, the invocation graph is constructed using the normal depth first algorithm described in Chapter 4. This construction is left incomplete at the points where a function-pointer-call is encountered. Next, points-to analysis is performed using this incomplete invocation graph. Once encountering a function pointer call, all the functions it can point to according to the current points-to information are determined. The invocation graph is updated accordingly to indicate that all these functions are invocable from the given call-site. The points-to analysis assumes all these functions to be invocable from the call-site under consideration and merges their output points-to sets to compute the points-to information at the program point after this call. Thus the points-to analysis keeps updating the invocation graph, while calculating the points-to information, and constructs the complete invocation graph to be used by other interprocedural analyses.

Consider the example shown in Figure 8.2. Without points-to information, the



actual function call(s) for **fp** at statement **S2** is not known, therefore, a conservative approach should be taken which says that **fp** can be all the possible functions in the program (in this case **f** and **g**). The related invocation graph for the worst case is represented in Figure 8.2(a). Figure 8.2(b) represents the invocation graph decorated with the points-to information after the completion of points-to analysis. As it is shown in this figure, in case of having more than one possible case for a function pointer, the result from different calls should be merged. This merging results in some extra points-to information that will propagate an inaccurate result throughout the analysis.

```

int *x, y ;
int *a, b ;
main() {
    void (*fp)() ;
S1: fp = g ;
    ...
S2: fp() ;
}
g()
{
    x = &y ;
}
h()
{
    a = &b
}

```

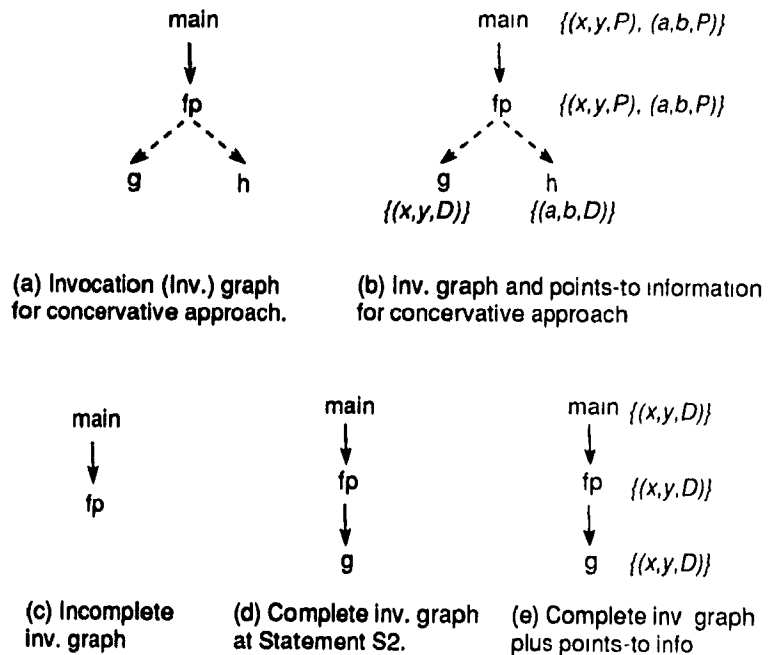


Figure 8.2: An example of extending the invocation graph and points-to analysis in the presence of function pointer.

Using the points-to information, the invocation graph is completed at the same time as the points-to analysis. First an incomplete invocation graph is built, as it is shown in Figure 8.2(c). Next, points-to analysis is performed. After encountering the function pointer call **fp()** at statement **S2**, the points-to information (**fp.g.D**) is used for updating the invocation graph, by adding the call **g** as the child of **fp**. The result is represented in Figure 8.2(d). Next is to follow the newly added call to continue the points-to analysis. This results in a more accurate points-to information that is

represented in Figure 8.2(e).

Note that the structured nature of points-to analysis plays an important role in efficient handling of function pointers without requiring any extra pass through the program. More details can be found in [Ghi92].

#### 8.2.4 Generalized Constant Propagation

Generalized constant propagation is an analysis that attempts to determine what range of values each variable can get at each point in a given program. Each time a variable is assigned a value, the value so given (if it is not user-input) is a function of existing variables. Thus, since most programs contain a liberal sprinkling of constants throughout, and provided one defines appropriate semantic functions for each possible operation, it should be possible to determine the set of the range of values each variable could contain at any point in its lifetime in any conceivable run of the program by performing a top-down semantic analysis of the program.

If one is to apply semantic functions to compute possible values, one of course needs to know the possible values of the variables serving as input, as accurately as possible. For example, if it is known that *a* could have any value in the range of [0,3] and *b* could have any value in the range of [1,2], then *a+b* can be any value in the range [1,5]. The range [1,5] is calculated from the above values using the semantic function corresponding to simple addition of integers. However, given the ubiquity of indirect referencing in popular languages such as C and Pascal, such information may not even be readily available, let alone accurate. In these cases, there are two options: (i) either one assumes the worst case assumption for each dereference variable (i.e., that each dereference variable could assume any value its type allows), or (ii) one makes use of points-to analysis to determine which variables might be the result of each dereference. In the former case, the quality of the information gathered via generalized constant propagation is substantially reduced, so the latter method is naturally preferable. Points-to analysis, then, is of particular importance if generalized constant propagation is to be useful for analyzing any non-trivial program in such pointer-based languages.

We show the effect of points-to analysis in two examples, one with points-to information and one without. In order to understand the examples, we define the following notations:

- GCP comments: represents the information available after the statement would have been executed.

- $\pm \text{Inf}$  : means  $\pm$  infinity.
- $\text{var}:[\text{val1},\text{val2}]$  : means that 'var' can get the values between 'val1' and 'val2'.

The following represents a small program with the result of general constant propagation after each statement of the program. After processing statement S1, `count` gets the constant zero. The range of `ptr` is positive numbers, because `ptr` is a pointer variable and pointers can hold only addresses (which are always a positive number). Since variable `i` is not assigned to any value, it can be any value. If the points-to relationship is not available at statement S3, the worst-case assumption says that `*ptr` can be either `i` or `count`. Therefore, the constant 1 should be added in the range of the variables that they can get. This results in increasing the range of the variable `count` and leaving the value of `i` unchanged (because infinity plus one is still infinity).

```
/* WITHOUT points-to information */
int main() {
    int i ;
    int *ptr ;
    int count ;

S1:    count = 0 ;
        /* GCP = {count:[0,0],ptr:[0,+Inf],i:[-Inf,+Inf]} */

S2:    ptr = &i ;
        /* GCP = {count:[0,0],ptr:[0,+Inf],i:[-Inf,+Inf]} */

S3:    *ptr = 1 ;
        /* GCP = {count:[0,1],ptr:[0,+Inf],i:[-Inf,+Inf]} */
}
```

The following shows the result of the same program while points-to information is available. At statement S3, the points-to relationship is  $(\text{ptr}, i, D)$ , therefore `*ptr` is equivalent to `i`. The result of general constant propagation would be that `count` is constant zero and `i` is constant one.

```

/* WITH points-to information */
int main() {
    int i ;
    int *ptr ;
    int count ;

S1:    count = 0 ;
        /* GCP = {count:[0,0],ptr:[0,+Inf],i:[-Inf,+Inf]} */

S2:    ptr = &i ;
        /* GCP = {count:[0,0],ptr:[0,+Inf],i:[-Inf,+Inf]} */

S3:    *ptr = 1 ;
        /* GCP = {count:[0,0],ptr:[0,+Inf],i:[1,1]} */
    }

```

This simple example shows the strong affect of points-to information on general constant propagation.

Table 8.6 represents the statistical result of general constant propagation in the presence and absence of points-to analysis. This table contains the following information about the benchmark files:

1. Benchmark: is the name of the benchmark
2. Exact const: is exact constants, like [3,3]
3. Bounded ranges: both ends of the range are constant, like [0,10]
4. Half ranges: one end of the range is infinite, like [2,Inf]
5. Unbounded ranges: both ends of the range are infinite, like [-Inf, Inf]
6. Range instances (total of the above 4)
7. Description: is the description of the benchmark

Referring to Table 8.6, it is observed that the presence of points to analysis has generally resulted in a noticable improvement. In specific, there is a large difference between the values of exact constant which is the most important constant information in a program.

Benchmark	Exact const	Bounded ranges	Half ranges	Unbounded ranges	Range instances	Description
spanning	74	81	0	26267	26422	Minimum spanning circle calculation
	84	93	648	25597	26422	
3sorts	1	0	29	3765	3795	3 Sorts: Bubble, Quick, Quick (Median of 3)
	2	3	204	3586	3795	
determinant	20	18	77	3375	3490	Matrix determinant calculation
	57	113	159	3161	3490	
knight_tour	19	57	50	2362	2488	Recursive Knight's tour solution calculation
	30	248	0	2210	2488	
root_complex	9	15	46	17548	17618	Complex roots of polynomials
	59	159	215	17185	17618	
heap_sort	34	68	20	6257	6379	Heap sort
	103	89	82	6105	6379	
Mersenne_prime	54	104	141	2906	3205	Mersenne primes
	61	240	157	2747	3205	
2sorts	13	8	9	2457	2487	2 Sorts: Bubble, Quick
	21	12	79	2375	2487	

Table 8.6: Statistics of general constant propagation in the absence (first line) and presence (second line) of points-to analysis.

### 8.2.5 Practical Array Dependence Analysis

This analysis determines whether dependences exist between *two subscripted references to the same array in a loop nest*. This is a two-step process. The first step is to set up a system of dependence equations and inequalities. In the second phase, a decision algorithm determines if the system has a solution [BC86]. The goal of the dependence testing is to disprove dependence of array subscripted pairs as many as possible and as early as possible. If dependence exists, it tries to find *distance and direction vectors*<sup>3</sup>. There are three cases where pointer alias information is used by array dependence analysis. These are listed in the following:

- To distinguish a good-loop<sup>4</sup> in the program. If a loop is not a good-loop the dependence analysis does not continue.

At statement S1, there is a relationship (a,b,D). Therefore the modification to \*a is independent of loop index i, consequently, the following loop is a good-loop.

<sup>3</sup>Distance and direction vectors represent the access pattern between loop iterations

<sup>4</sup>A good-loop is a loop-nest with no statements in between two loop statements and loop indices not modified in the loop body.

```

main()
{
  int  w[100] ;
  int  *a, b, i ;

      a = &b ;
      for ( i = 1; i <= 99 ; ++i )
      {
S1:    *a = i * 2 ;
        w[i] = w[i+1];
      }
}

```

However, in the lack of points-to information, any loop with a modification to an indirect reference would not be considered as a good-loop

- To replace, if necessary, any indirect reference variable with its corresponding points-to variable(s) along the backward analysis path. This replacement is used to increase the possibility of a successful *canonical transformation* which is a transformation that converts array subscript expressions into *canonical expressions*<sup>5</sup>.

In the lack of this information, one can not continue the backward analysis when an indirect reference variable is found. Therefore, the canonical form can not be constructed.

- To detect whether two pointer references point to the same array reference or a pointer reference points to an array reference. This detection increases the number of array pairs to be analyzed by practical array dependence tester. In the following example, without points-to information, one could not detect that **s[i+1]** (from statement S1) and **a[i+1]** are the same location. Therefore, one should take the conservative assumption that says: **s** can refer to any array; consequently, **a** is dependent on all the arrays in its current scope (b and c).

---

<sup>5</sup>A canonical expression is a subscript expression that can be expressed as a linear (affine) function of for loop variables

```

main()
{
    int  a[100], b[50], c[7];
    int  i;
    int  *s;

    s = a ;
    for (i = 1 ; i < 100 ; i++)
S1:    a[i] = s[i+1];
}

```

Readers are refer to [Jus93] for more details on this analysis.

### 8.2.6 Reaching Definition Analysis

Reaching definitions have been formally defined in [ASU86] as: A *definition of a variable  $x$*  is a statement that assigns, or may assign a value to  $x$ . A definition  $d$  *reaches* a point  $p$ , if there is an execution path immediately following  $d$  to  $p$  such that  $d$  is not 'killed' along that path. This is a forward analysis where a program is analyzed from top to bottom [ASU86].

In the case of pointers, points-to information is used to make reaching definition information as accurate as possible. Consider the following example:

```

S1:    *t = 3;
S2:    --->

```

When S1 is reached, all the variables that  $t$  points-to are considered. If  $t$  definitely points-to a variable, say  $a$ , it is concluded that the definition S1 of  $a$  definitely-reaches S2. If  $t$  possibly-points-to some variables, say  $a$  and  $b$ , then it is concluded that the definition S1 of both  $a$  and  $b$  possibly-reaches S2.

If the points-to information were not available, then all the variables of the current scope would be considered to possibly-reach to S2.

The general case of this analysis together with an example is given in Chapter 4. Readers are referred to [Sri92] for more details.

### 8.2.7 Live Variable Analysis

A variable  $x$  is *live* at a point  $p$  in the program if the value of  $x$  at  $p$  could be used along some execution path in the program starting at point  $p$  [ASU86]. Thus, during live-variable analysis, at every point in the program, the set of variables that are live at that point in the program are computed. This is a backward analysis (the program is analyzed from bottom to top) and is absolutely essential for optimizations like register allocation.

For pointers, the points-to information is used to determine liveness accurately. Consider the following example:

```

                                <--- 'a' is definitely-live
S1:    ... = *t;
```

When the statement **S1** is reached, all the points-to information of **t** are considered. If **t** definitely-points-to a variable, say **a**, then it is concluded that both **t** and **a** are definitely-live at **S1**. If **t** possibly-points-to some variables, say **a** and **b**, then it is concluded that **a** and **b** are possibly-live and **t** is definitely-live at **S1**.

If the points-to information were not available, then all the variables of the current scope would be considered to be possibly-live at **S1**. This increases the live ranges of variables and, hence, makes the register allocation less efficient.

Another example of this analysis is given in Chapter 4. Readers are refer to [Str92] for more details.

### 8.2.8 Loop Unrolling

Loop unrolling refers to the process of making one or more copies of the loop body whereby the loop control must be suitably modified. Loop unrolling has many advantages including: increasing the number of instructions that can be scheduled in a basic block, decreasing the unnecessary data movement in a memory hierarchy architecture, reducing loop overheads, etc.

One of the conditions for loop unrolling is that the loop index should not be modified inside the body of the loop. Without points-to information, it is difficult to know whether any pointer (whose dereference is being modified in the body of the loop) is indirectly pointing to a loop index or not. For example, consider the following program:



```

main()
{
    int a[100], b[100] ;
    int *x, y ;
    x = &y ;
    ...
    for ( i = 1 ; i < 100 ; i = i + 1 ) {
        a[i] = b[i-1]
S1:      *x = *x + i ;
    }
}

```

Without the points-to information, it is not safe to unroll the loop, because  $*x$  could be  $i$ . With points-to information, it is trivial to notice that at statement S1 the points-to relationship is  $(x,y,D)$ , and the loop can be unrolled as shown below.

```

main()
{
    int a[100], b[100] ;
    int *x, y ;
    x = &y ;
    ...
    for ( i = 1 ; i < 99 ; i = i + 2 ) {
        a[i] = b[i-1] ;
        *x = *x + i ;
        a[i+1] = b[i] ;
        *x = *x + 1 + 1 ;
    }
}

```

## Chapter 9

# Related Work

As mentioned in the introduction, two variables  $a$  and  $b$  are considered to be aliased at a program point  $p$ , if they refer to the same memory location. This is represented as  $\langle a, b \rangle$ . There are two general approaches to solve the aliasing problem: (i) automatic alias analysis in which no input is needed from the programmer and alias analysis is done by the compiler during the optimization phase, and (ii) directive code annotations in which the programmer provides some information about the properties of pointer-based data structures to enable powerful compiler optimizations [HHN92a, HHN92b]. In this thesis, we have focused on the first approach.

In this chapter, we first give the historical work that has been done in the area of automatic alias analysis, in Section 9.1. Then, we focus on the comparison of our work with the latest work that has been done on interprocedural aliasing in the presence of pointers by Landi and Ryder [Lan92, LR92], in Section 9.2.

### 9.1 Historical Background on Alias Analysis

An alias analysis is needed for different cases in a programming language. Figure 9.1 shows a general division of these cases.

Traditionally, the alias analysis methods have focused on calculating the aliases generated by the association of actual arguments with call-by-reference parameters. This sort of aliasing is characteristic of languages like Fortran and has the following important properties: (i) only call-sites contribute to the creation of aliases, and (ii) an alias relation valid at the entry of a procedure holds throughout its body. Appropriate techniques for computing such aliases have been developed and described [Bar78].

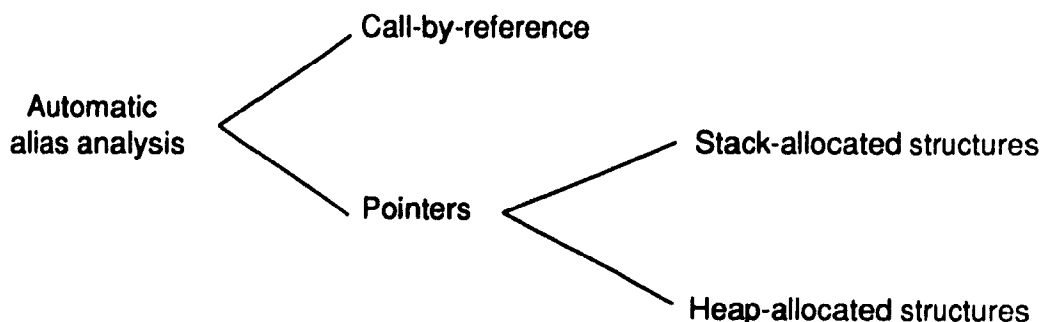


Figure 9.1: A general division of alias analysis.

Ban79, Mye81, Coo85, CK89]. Most of these analyses collect the alias information in two passes over the program: (i) an introductory pass to compute all the obvious aliases generated at call-sites, (ii) a propagation pass to propagate the aliases through the call-graph of the program.

Unlike the call-by-reference mechanism, the presence of pointers considerably complicates the problem of alias computation. This is due to the following factors

- In addition to the call-sites, the alias information can be affected by any program statement modifying a pointer variable.
- In the presence of multi-level pointers, a function can change the alias information of pointer type variables that are not in its scope. The indirect swap of two-level pointer type variables (given in Section 4.2) is an example of this case.

Further, there are number of features particular to the C-language that make this problem even harder. Such features include: pointer arithmetic, pointers to arrays, and type casting. The complications introduced by these features have been explained in Chapter 6.

As shown in Figure 9.1, pointers can be classified into two broad categories. The basis for this categorization is the location of the pointer target in the memory organization. A pointer is categorized as stack-directed, if it points to a memory object resident on the stack. Similarly, a pointer is classified as heap-directed, if it points to an object allocated in the heap. Stack-directed pointers have the nice property that their target always possesses a unique compile-time name. This does not hold for heap-directed pointers as all the objects in the heap are anonymous. In this thesis, we have focused on the analysis of stack-directed pointers, with our points-to abstraction designed to exploit their special property. A good deal of work has been done on the

analysis of heap-directed pointers [JM81, JM82, LH88, HN90, CWZ90, HHN92a]. We now compare our work with the previous works keeping this distinction in mind.

Weihl [Wei80] gave an algorithm for interprocedural data-flow analysis in the presence of pointers. He considers programs with procedure variables and label variables. Therefore, the complete control flow information is not available to him. He first calculates the alias information generated by each relevant program statement, independent of other statements. Then, he performs a transitive closure on this information to obtain the set of aliases that can be valid in the program under analysis. This information is not *program-point-specific* (is not related to a specific point in the program), but is *program-specific* (is related to the whole program). The alias information computed by such a flow-free analysis would be very imprecise. Consider the following example:

```
int *a, b, c ;
```

```
a = &b ;
```

```
a = &c ;
```

Using Weihl's technique, one gets the alias pairs  $\langle *a, b \rangle$  and  $\langle *a, c \rangle$  to be valid for this program. Clearly, the first pair does not hold at the end of the program segment, but this can not be recognized by his analysis. Further, he does *k-limiting*<sup>1</sup>.

Chow and Rudmik [CR82] also developed an alias analysis algorithm for their CHILL Compiling System. They do not perform a flow-free analysis like Weihl, but instead collect program-point-specific aliases. However, they handle only single level pointers. Further, they use the notion of a supergraph which is obtained by linking together the condensed flow graphs (flow graph containing only the nodes relevant to alias analysis) of individual procedures, for interprocedural analysis. They perform intraprocedural analysis on this supergraph to achieve the effects of interprocedural analysis. This approach, however, leads to the incorrect pairing of calls and returns and makes their analysis imprecise. They also do not handle structures and arrays.

Coutant [Cou86] follows the same method as Weihl, of computing program specific aliases as opposed to program-point-specific aliases. However, she handles multi-level pointers, structures and arrays, more precisely. It is not clear how she would handle recursive data structures. She mentions that program-point-specific alias calculation did not prove to be worth the computational effort in her implementation, but she

<sup>1</sup>*k-limiting* specifies the depth of the pointers which are considered [Lan92, LR92]. For example, if  $k = 2$ ,  $***x$  represents  $***x$ ,  $****x$ , and etc. Therefore, if *k-limiting* is used, one obtains less accurate results for pointers with more than *k* indirections.

does not provide any relevant data. She also suggests the use of pragmas to help the compiler do a better analysis.

Cooper [Coo89] described an alias analysis algorithm in his Master thesis. His algorithm maintains explicit path information in the form of alias histories to ensure correct pairing of calls and returns. This method would give precise results, but does not seem to be feasible for implementation.

In the following section, we first give an introduction to Landi and Ryder's work and then compare their work with ours.

## 9.2 Comparison

In Landi and Ryder's analysis [Lan92, LR92], a program is represented as an *Inter-procedural Control Flow Graph (ICFG)*. An ICFG is the union of control-flow graphs of each function with edges from each call-site to the entry of callee and from the exit of callee back to the call-site. Each call-site is replaced by two nodes, a *call node* and a *return node*. Each function has an *entry node* and an *exit node*. There is an edge from each call node to the entry node of related function (which is unique), and there is another edge from each exit node to the related return node(s). If a function is called more than once, its exit node would have more than one successor. As it is shown in Figure 9.2, there are two edges from exit node of function `f` to the call-site nodes of function `main`.

Two variables  $a$  and  $b$  have *may alias* relationship, represented as  $\langle a, b \rangle$ , at a program point  $n$ , iff there exists a path,  $n_1 n_2 \dots n$  in the ICFG, on which  $a$  and  $b$  refer to the same location after execution of the program point  $n$  where  $n_1$  is the entry node of the function `main`.

Two variables  $a$  and  $b$  have *must alias* relationship, at a program point  $n$ , iff for all the paths  $n_1 n_2 \dots n$  in the ICFG,  $a$  and  $b$  refer to the same location after execution of the program point  $n$ .

Their may alias relationship is based on the following *may-hold* information:

$$\text{may-hold}(n_i, AA, PA),$$

where  $n_i$  is a node in ICFG,  $AA$  is a set of aliases, and  $PA$  is a may alias pair (e.g.  $\langle a, b \rangle$ ).  $\text{May-hold}(n_i, AA, \langle a, b \rangle)$  is true, iff  $\langle a, b \rangle$  holds on some path from  $\text{entry}(n_i)$ , the entry node of the procedure containing  $n_i$ , to  $n_i$  assuming there is a

```

int *a ;
main() {
    int b, c ;

    a = &b ;
    f() ;
    a = &c ;
    f() ;
}
f() {
    int *x ;

    x = a ;
}

```

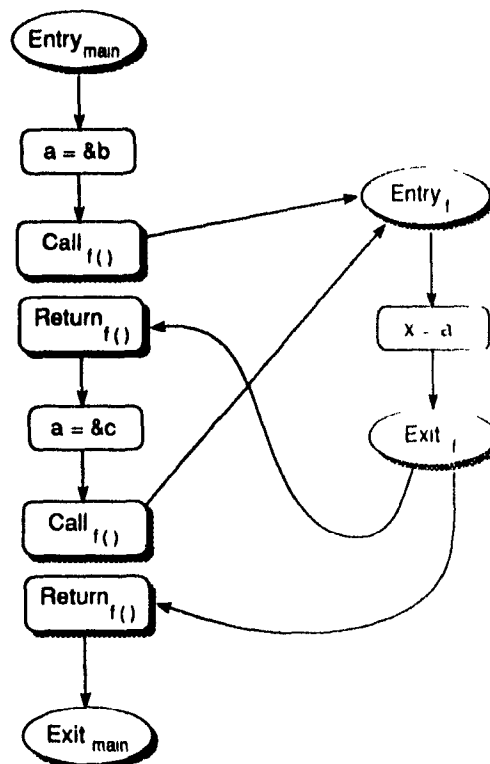


Figure 9.2: An example of ICFG.

path from the entry node of **main** function to  $entry(n_i)$  on which the assumed alias set  $AA$  holds. Having may-hold information, may aliases can be computed using the following formula:

$$may\text{-}alias(n_i) = \{PA \mid \exists (AA) \text{ may-hold}(n_i, AA, PA) \quad true\}$$

Landi and Ryder's algorithm first collects the obvious *may-hold* information, e.g.  $\langle *a, b \rangle$  for  $a = \&b$  (alias introduction), then collects the complete may hold information using a worklist technique, and finally computes the may aliases from may hold information using the above formula. Further, they restrict the cardinality of assumed-alias sets to a maximum of one.

In the following sections, we compare our method with Landi and Ryder's method by discussing alias representation in Section 9.2.1, must aliases in Section 9.2.2, in terprocedural analysis in Section 9.2.3, dynamic allocation and k-limiting in Section 9.2.4, and type-casting in Section 9.2.5.

### 9.2.1 Alias Representation

We collect the alias information in the form of points-to pairs instead of exhaustive alias pairs like Landi and Ryder. Consider the simple example presented in Figure 9.3.

```
int **a, *b, c ;  
a = &b ;  
b = &c ;
```

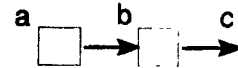


Figure 9.3: A simple C program.

In points-to analysis, we get  $\{(a,b,D), (b,c,D)\}$  as the result, while Landi and Ryder get the following:

$$\{ \langle *a, b \rangle, \langle **a, *b \rangle, \langle **a, c \rangle, \langle *b, c \rangle \}$$

It is evident that points-to pairs represent the alias information in a more compact and informative manner than alias pairs. The example in Figure 9.4 also demonstrates this point<sup>2</sup>. Further, when alias pairs are generated exhaustively, some additional pairs might be reported while handling multi-level pointers. Consider the following example:

```
int **a, b ;  
a = &b ;      { \langle *a, b \rangle, \langle **a, *b \rangle }
```

Landi and Ryder's analysis would give the alias pair  $\langle **a, *b \rangle$ , which is redundant as  $b$  has still not been assigned the address of a memory location. Our points-to analysis gives the precise information that  $a$  points-to  $b$  and  $b$  does not point to any memory location. This also enables us to detect errors in programs, caused by dereferencing a pointer without assigning it the address of a valid memory location. Readers are referred to Chapter 7 for further details.

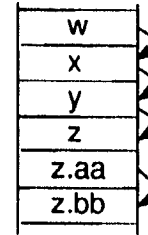
Moreover, points-to abstraction is a natural representation for calculating dependence information. This has been demonstrated in [HGS92]. The authors used the points-to information to build read and write sets for each statement. These sets actually consist of the reads and the writes to the abstract stack locations. Using the read/write sets, it is straight-forward to calculate data dependencies between statements.

<sup>2</sup>Acknowledgment: We are thankful to W. A. Landi who provided us with his results mentioned in Figure 9.4.

Consider the calculation of these sets for the statement 1 of Figure 9.4(a). From the points-to information  $(y, z, D)$ , it can be easily determined that statement 1 reads the stack location  $y$  and writes to the stack location  $z.bb$ . However, with the alias pair representation, it would be necessary to inspect all the alias pairs involving  $y$  and its aliases to compute this information. This would be a considerably more expensive task (specially for multi-level pointers e.g.  $(**x).bb$ ).

```
main()
{
    struct{
        int  *aa ;
        int  bb ;
    } ***w, **x, *y, z ;

    w = &x ;
    x = &y ;
    y = &z ;
    (*y).aa = &((*y).bb) ;
    (*y).bb = 10 ;      /* stmt1 */
}
```



(a) A C-program with its related abstract stack representation

```
{<(**w)->bb,*(y->aa)>,    <z,*y>,
<(**w)->bb,*(z.aa)>,      <y,*x>,
<z,**x>,                  <x,*w>,
<y,**w>,                  <*(z.aa),y->bb>,
<(*x)->bb,*(**w)->aa>,    <*(z.aa),z.bb>,
<*(z.aa),(*x)->bb>,      <*(y->aa),(*x)->bb>,
<*(y->aa),y->bb>,         <(**w)->bb,*(**w)->aa>,
<*(**x)->aa,z.bb>,       <*(y->aa),z.bb>}
```

(b) The result of alias analysis using the Landi and Ryder's approach

$\{(z.aa, z.bb, D), (y, z, D), (x, y, D), (w, x, D)\}$

(c) The result of points-to analysis using our approach.

Figure 9.4: A comparison example of alias representation and points to representation.

Finally, alias pairs can always be generated from points-to pairs, using the conversion algorithm given in Section 3.3.

However, points-to representation can provide extra alias information in certain



cases. Consider the example given in Figure 9.5.

```

int **a, *b, c ;
if (c)
    a = &b ;      <--- {(a,b,D)}
else
    b = &c ;      <--- {(b,c,D)}
S1:                <--- {(a,b,P), (b,c,P)}

```

Figure 9.5: An example of points-to representation.

Landi and Ryder's result at program point S1 would be  $\{ \langle *a, b \rangle, \langle **a, b \rangle, \langle *b, c \rangle \}$ . If one wants to convert our points-to information to their alias pairs, one would get the set  $\{ \langle *a, b \rangle, \langle **a, *b \rangle, \langle *b, c \rangle, \langle **a, c \rangle \}$ , with the extra alias pair  $\langle **a, c \rangle$ . This happens because it can not be detected that the points-to pairs  $(a, b, P)$  and  $(b, c, P)$  are propagated to S1 from different paths and transitive closure operation should not be applied on them.

Similarly, there are cases where points-to abstraction eliminates extraneous alias information, while Landi and Ryder's alias pairs can not. Consider the following example:

	The alias information -----	Memory representation -----
	int **x, *y, z, w ;	
S1:	x = &y ;    {<*x,y>, <**x,*y>}	x->y
S2:	y = &z ;    {<*x,y>, <**x,*y>, <*y,z>, <**x,z>}	x->y->z
S3:	y = &w ;    {<*x,y>, <**x,*y>, <*y,w>, <**x,z>, <**x,w>}	x->y->w

When statement S2 is processed, the alias pair  $\langle **x, z \rangle$  is generated by the interaction of the alias pair  $\langle *x, y \rangle$  and the assignment  $y = \&z$ . At statement S3, the address of  $w$  is assigned to  $y$ , therefore,  $\langle *x, z \rangle$  is not valid anymore. Landi and Ryder's analysis fails to detect this case, because while propagating  $\langle **x, z \rangle$  through S3, they do not have the information that this alias pair was generated because of  $\langle *y, z \rangle$  being valid. While, if alias pairs are generated from the points-to information

available after processing S3, this alias pair will not be generated. A similar observation was made in [CBC93]. They use the notion of transitive reduction which is quite similar to our points-to abstraction.

### 9.2.2 Must Aliases

The concept of may alias is equivalent to our possibly-points-to relationship, and the concept of must alias is equivalent to our definitely-points-to relationship. Must aliases are not mentioned in [LR92], although they are mentioned in [Lan92]. However, they consider the calculation of must aliases separately from that of may aliases. While we use must alias information to improve the precision of may alias computation for multi-level pointers by obtaining better kill information, as explained in Chapters 3, 4, and 5.

In [CBC93], it is mentioned that the extraneous alias information eliminated by transitive reduction can be equivalently removed using must alias information. However, in certain cases, extraneous pairs which can not be removed by using transitive reduction, can be removed by using must alias information. This is demonstrated in the following example:

	The alias information (without transitive closure)	Memory representation
	-----	-----
int **x, *y, z, w ;		
S1: x = &y ;	{<*x,y>}	x->y
S2: y = &z ;	{<*x,y>,<*y,z>}	x->y->z
S3: *x = &w ;	{<*x,y>,<*y,w>,<*y,z>}	x->y->w

At statement S3, if it were not known that **x** definitely points to **y** (i.e., **\*x** is a must alias of **y**), we would not be able to kill the points-to relationship (**y->z**) (which can be killed due to the assignment **\*x = &w**), and get the precise information **x->y->w**. The extraneous pair **<\*y,z>** in the transitively reduced alias information reflect the fact.

Further, must-alias information is very useful for other analyses like constant propagation, register allocation, and dependence analysis, as explained in Chapter 8.

In the presence of multi-level pointers, calculation of must aliases would be quite complicated using the worklist technique of Landi and Ryder. This complexity is due to the fact that the information is processed in random order.

### 9.2.3 Interprocedural Analysis

Landi and Ryder represent the interprocedural structure of the program as an ICFG and use the conditional approach based on assumed aliases to perform interprocedural analysis.

We use the SIMPLE representation in conjunction with the invocation graph (as explained in Chapter 4), where each call-site (related to a given call chain) in the program has a separate invocation graph node associated with it. For each call, we visit the called function, analyze its body in the context of the call, and return the information to the call-site currently under consideration.

Our approach leads to the following advantages over Landi and Ryder's technique.

- The alias information from function exit points, always gets propagated to the correct call-site. It is never propagated along unrealizable interprocedural paths<sup>3</sup>. To affect the correct pairing of calls and returns, Landi and Ryder associate an assumed-alias set with each alias pair, at call-sites. To avoid the exponential growth of this set, they restrict its cardinality to a maximum of one. In the presence of multi-level pointers, this restriction can lead to the propagation of information to unrelated call-sites. Consider the example given in Figure 9.6.

At statement S5, the alias pair  $\langle **a, y \rangle$  results from the interaction of the assignment  $b = x$  and the alias pairs  $\langle *a, b \rangle$  and  $\langle *x, y \rangle$  holding at program point S3, with  $\langle *a, b \rangle$  and  $\langle *x, y \rangle$  as the assumed aliases, respectively. Now, the assumed alias set for  $\langle **a, y \rangle$  should include both of these alias pairs. Since the cardinality of the assumed alias set is restricted to at most one, Landi and Ryder propose to randomly pick one of the pairs. In case,  $\langle *a, b \rangle$  is chosen as the assumed alias pair  $\langle **a, y \rangle$  would be propagated to both the call-sites S1 and S2, as  $\langle *a, b \rangle$  is valid at both these call-sites. However, it should not be propagated to the call-site S2, as the other alias pair  $\langle *x, y \rangle$  is not valid there. Our analysis would propagate it only to the call-site S1, because it would analyze the body of function  $g()$  separately for the two calls.

<sup>3</sup>A path is *realizable* iff it is a path in the program and whenever a procedure on this path returns, it returns to the call-site which invoked it [Lan92, LR92]

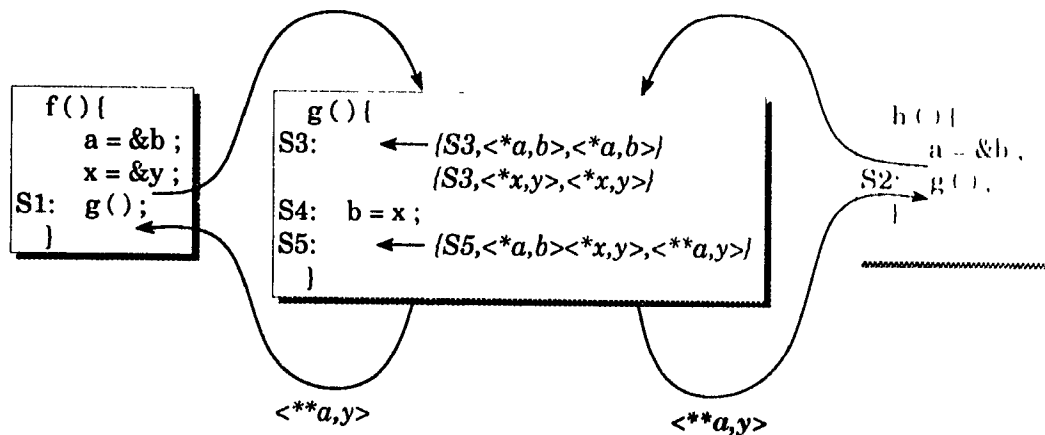
```

int **a, *b ;
int *x, y, z ;
main()
{
    f() ;
    h() ;
}
f()
{
    a = &b ;
    x = &y ;
S1:  g() ;
}

h()
{
    a = &b ;
S2:  g() ;
}
g()
{
S3:
S4:  b = x ;
S5:
}

```

(a) A C-program



(b) The graph representation of intraprocedural analysis

Figure 9.6: An example of intraprocedural analysis for Landi and Ryder's method

- Our analysis always ensures that two alias pairs propagated to a statement by two different calls, are not simultaneously used to generate a new alias pair. This is made possible by the structured nature of our analysis. The exact information for the current call is kept in a special data structure (stack matrix) which is used for the analysis. This information is also simultaneously merged with the pre-existing information saved in the statement nodes.

Landi and Ryder's worklist technique fails to ensure the above mentioned point. For the example given in Figure 9.6, their analysis would infer the existence of

the alias pair  $\langle **a, y \rangle$  at statement S5. This is resolved from the interaction of the assignment statement S4, and the alias pairs  $\langle *a, b \rangle$  and  $\langle *x, y \rangle$  holding at statement S3. But it is possible that these alias pairs get propagated to S3 from two different call-sites and never exist together. In that case, the alias pair  $\langle **a, y \rangle$  would not be valid at S5 in any program execution. They don't have a mechanism to detect such cases, and safely give the imprecise information.

- The structured nature of our points-to analysis, provides an opportunity to perform other data flow analyses simultaneously with it. This would enable other analyses to use the precise alias information available during points-to analysis, instead of the final merged information. To clarify this point, we demonstrate (i) the results of performing constant propagation after the completion of points-to analysis, and (ii) the results of performing constant propagation in conjunction with points-to analysis. Consider the following example:

```

int *a ;
int b = 1 ;
int c = 5 ;
int x, y, z ;
main() {
S1:   a = &b ;
S2:   f() ;
S3:   x = y ; <--- {<x,b,P>,<x,c,P>} = {<x,1,P>,<x,5,P>}
S4:   a = &c ;
S5:   f() ;
S6:   z = y ; <--- {<z,b,P>,<z,c,P>} = {<z,1,P>,<z,5,P>}
      }
      f() {
S7:   y = *a ; <--- {<y,b,P>,<y,c,P>} = {<y,1,P>,<y,5,P>}
      }

```

The arrow links represent the constant propagation information.  $\langle x, y, D \rangle$  means that variable  $x$  is definitely equal to variable  $y$ .  $\langle x, y, P \rangle$  means that variable  $x$  is possibly equal to variable  $y$ . For case (i), since the merged information for points-to analysis is used,  $y$  is either  $b$  or  $c$  (the constant propagation information saved at S7). Therefore,  $x$  and  $z$  can not be concluded to be constant. Below, we show the same information, but for the case that constant propagation is done simultaneously with points-to analysis.

```

int *a ;

```

<pre> int b = 1 ; int c = 5 ; int x, y, z ; main() { S1:   a = &amp;b ; S2:   f() ; S3:   x = y ; &lt;--- {(x,b,D)} = {(x,1,D)} S4:   a = &amp;c ; S5:   f() ; S6:   z = y ; &lt;--- {(z,b,D)} = {(z,5,D)}       }       f() { S7:   y = *a ;       } </pre>	<pre> main() {       a = &amp;b ;       f() ;       x = 1 ;       a = &amp;c ;       f() ;       z = 5 ;       }       f() {       y = *a ;       } </pre>
--	--

In this case, statement S3 is processed when the points-to information is (a,b,D) and therefore, y at S7 would be equal to b, which is constant 1. This gives a more precise information about x at statement S3, that results in the deduction fact that x is constant 1. Similarly, at statement S6, we get precise information because we have (a,c,D). This statement also resolves z as being the constant 5.

- Our interprocedural analysis can be easily extended to handle function pointers without incurring any additional cost. This is facilitated by the structured nature of our points-to analysis and the powerful representation of the interprocedural structure of the program by the invocation graph. The detailed algorithm is described in [Ghi92], and a brief summary is provided in § 2.3.
- By using invocation graph, we are able to save some important properties of each call in the corresponding invocation graph node. These are described below
  - The relationship between the invisible variables of the callee, and the caller variables (map information of each call). This is a useful information for other analyses [HEGV93].

As an example, consider the constant propagation analysis for the example represented in Figure 9.7. Assume that this analysis starts after the completion of points-to analysis. Using statement S5 and the relation  $\text{hp}(\mathbf{x}, 1\mathbf{x}, D)$ , one can deduce that the invisible variable  $1\mathbf{x}$  is constant c. After returning to the call-sites S1 and S3, one needs to know the variable y represented by  $1\mathbf{x}$  in the caller, so that the constant information can be propagated. Using the map information related to S1/S3, we resolve that

at  $b/c$  is constant 7 at S2/S4. Therefore  $b/c$  can be replaced by constant 7 at statement S2/S4.

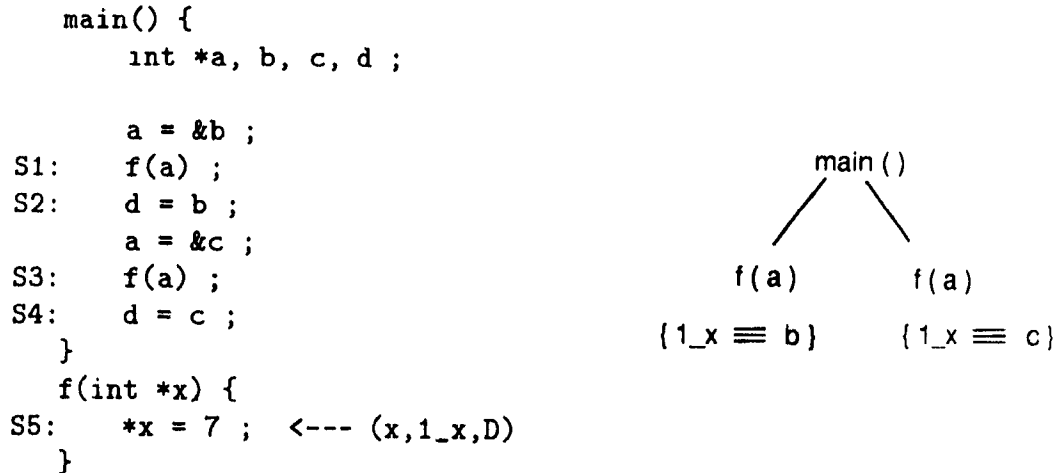


Figure 9.7: An example of the usage of the map and unmap information saved in invocation graph.

- The input and output points-to information for each node of invocation graph can be saved in a data base, so that if a function is called for the second time with the same input, there is no need to recalculate the output for it.
- Our invocation graph is also suitable for *procedure cloning* [CHK92]. This involves duplicating the body of a function with a different name (i.e. making its clone), and replacing some calls to this function, with a call to its clone. This technique is useful when the data flow information propagated from two different calls to a function is significantly different. In this case, if the information from one of the calls is propagated to the clone, the imprecision resulting from merging significantly different information can be avoided. To replace a function call with a call to its clone, we simply need to modify the invocation graph node corresponding to it. Further, a clone for the function containing this call, also needs to be developed. This process would continue recursively till the root of the invocation graph is reached.

#### 9.2.4 Dynamic Allocation and k-limiting

To handle recursively defined dynamic data structures, Landi and Ryder use the notion of k-limited object names. They limit the number of dereferences permissible

in any object name to some constant  $k$ . They provide the alias information in the form of alias pairs between these names, which gives a very imprecise picture of the heap.

A different abstraction is required to properly handle the aliasing for dynamically allocated objects. Presently, we use a single location called 'heap' in our abstract stack for the points-to analysis. All heap-directed pointers point to this location. Development of appropriate abstractions for heap analysis, based on the path matrix model proposed in [Hen90], is currently under development [Ghi93]. There is no need to use  $k$ -limiting for abstract stack analysis, as the number of objects on the stack would always be finite and compile-time determinable.

### 9.2.5 Type Casting

Landi and Ryder do not handle type casting. It is a frequently used feature of the C-language and needs to be taken into consideration. We handle a wide variety of type casting as explained in Section 6.2. The only case that we do not handle, is when the type is transformed to/from a scalar variable, as shown in the following example

```
int *a, b, c ;  
b = (int) (&c) ;  
a = (int*) b ;
```

This sort of type casting can also be handled, but it would make the analysis expensive. Further, this case does not occur frequently in real C programs.

The most recent algorithm for *flow-sensitive* interprocedural<sup>4</sup> alias analysis has been presented in [CBC93]. Intraprocedurally, their algorithm is similar to that of Landi and Ryder, with the exception that they use a sparse representation called *Sparse Evaluation Graph*, instead of control flow graph. To ensure the flow of information along realizable paths during interprocedural analysis, they associate two items of information with each alias pair: (i) the call-site from which the alias pair is propagated, and (ii) a source alias set, which is the set of aliases that induce the alias pair through the call. The call-site information alone is not sufficient to propagate aliases along realizable interprocedural paths. Besides, unless they keep source alias sets of exponential size, our analysis would give more precise results. They also proposed the schemes of using transitive reduction and must alias information, to

<sup>4</sup>A flow-sensitive interprocedural analysis is an analysis which makes use of the intraprocedural control flow information associated with individual procedures.



improve the precision of alias analysis. We have already discussed these issues in the previous sections.

## Chapter 10

# Conclusions and Future Work

### 10.1 Summary and Conclusions

The optimization phase is one of the crucial phases of a compiler. In order to get effective results in this phase, one needs sophisticated flow analysis techniques to collect accurate information about the various variables and aggregate structures used by a program. In the presence of pointer type variables, the performance of most of the analyses depends on the accuracy of alias analysis.

In this thesis, we discussed an accurate and practical alias analysis algorithm that collects the points-to relationship between variables instead of explicit alias information in the form of exhaustive alias pairs. Our analysis is focused on stack allocated data structures. We consider single and multi-level pointers, aggregate structures (arrays, structures, and pointers to arrays), type casting, and arithmetic operation on pointers. The algorithm has been implemented in the framework of the McCAI compiler, and is being used by several high and low-level analyses.

For a given program, we first build the invocation graph with a special representation for recursive function calls. Then, we perform the interprocedural points-to analysis in two phases: (i) a first pass over the program to collect all the variable relevant to our analysis (this is done to minimize the size of our bit matrix data structure), (ii) a second pass to collect the points-to information by developing and using an explicit rule for each possible statement. We handle function calls by following the actual program execution paths using our invocation graph.

The results of points-to analysis is used when there exists an indirect reference (e.g., `*x`) in the program. Our experimental results over some standard benchmarks

show that the average number of variables that a pointer type variable points to, is very close to unity (1.04). In other words, an indirect reference  $*x$  is in almost all cases, either possibly or definitely aliased to a single variable. We also conclude that around 47% of all the points-to relationships are definitely-points-to, this relationships are very valuable for any analyses. These results are due to our accurate analysis methods.

## 10.2 Future Work

There is a considerable scope for further work in the following directions:

- **Extension to dynamic data structures:** Our analysis is designed for stack-based variables. Presently a single location is used to represent all the dynamically allocated objects in order to be able to handle benchmarks. An appropriate abstraction to handle dynamic data structures is currently being developed, based on the path-matrix model proposed in [Hen90, Ghi93].
- **Procedure cloning using points-to information:** When the points-to information propagated from two different calls to a function is significantly different, the information from one of the calls can be propagated to a clone of the function. In this way, the imprecision resulting from merging significantly different information can be avoided. As the input to a function for each call is saved in our invocation graph, this feature can be easily added to our analysis.
- **Precise saved information:** Presently, the merged information gathered during different passes to a function is saved in each statement node of the program under analysis. This leads to some imprecision caused by the merge. In order to get the exact result, one can save the points-to information at each statement for different passes, separately along with the related invocation graph node. Since each node in the invocation graph represents a call-chain, and it is unique for each call, keeping this extra information is sufficient to differentiate points-to information of different calls. Therefore, the only cost would be the size of data. If this size is not an issue, this extension can be naturally added to our analysis.
- **Extension to our array analysis:** Presently, we use one location in our abstract stack for an entire array. However, we treat an array differently depending on, if the address of its first element or some other element is assigned to a pointer. We further plan to refine this scheme and use two locations for each array. One

for the first element and one for the other elements. In our method a possibly points-to relationship<sup>1</sup> for an array occurs due to: (i) a merge at the join points of compositional control statements, or (ii) assigning the address of an element of the array, which is not its first element, to a pointer. Currently, we can not distinguish whether a possibly-points-to relationship for an array is caused by (i) or (ii). While in the proposed method the other array element will be used for case (ii). Therefore, the distinction between this two cases would be possible.

- **Array parameters:** In the C-language, when an array is passed as a parameter, it is changed to a pointer variable in the callee. Therefore, one can not know if a pointer variable is a pointer to array or not. Using our analysis, we can find out if a pointer points to an array. We can also provide the range of the pointed to array. This can be done using our invocation graph and the map information which is saved in each node of the invocation graph. Suppose a pointer variable  $x$  points-to a variable  $y$ . If  $y$  is in the scope of the callee, one can easily know if it is an array and also what is its range. Otherwise, it is an invisible variable<sup>2</sup> and the map information saved in each invocation graph node can be used to get the equivalent variable(s) for this invisible variable.

The result of this extension would be useful in handling pointer arithmetic, since we handle arithmetic operation on pointers to arrays more precisely than on pointers to scalar variables. Further, this information is very useful for array dependence analysis.

- **Union type:** Our analysis can be easily extended to handle union types. We just need to generate names for all the possible access to a union type.
- **Points-to information saved in invocation graph:** The abstract stack representation before and after processing a function call, is saved in each node of an invocation graph. This information can be used as a data base, to reduce the passes to a function call. If an input abstract stack has already occurred and an output abstract stack is computed for it, in the second appearance of the same input, one can use the already computed output.

---

<sup>1</sup>Refer to Chapter 3 for definition.

<sup>2</sup>Refer to Chapter 4 for definition.

## Appendix A

# The SIMPLE Grammar

```
all_stmts : stmtlist stop_stmt
          | stmtlist
```

```
stmtlist : stmtlist stmt
          | stmt
```

```
stmt : compstmt
      | expr ';'
      | IF '(' condexpr ')' stmt
      | IF '(' condexpr ')' stmt ELSE stmt
      | WHILE '(' condexpr ')' stmt
      | DO stmt WHILE '(' condexpr ')'
      | FOR '(' exprseq ';' condexpr ';' exprseq ')' stmt
      | SWITCH '(' val ')' casestmts
      | ';'

```

```
compstmt : '{' all_stmts '}'
          | '{' '}'
          | '{' decls all_stmts '}'
          | '{' decls '}'

```

```
/** decls denotes all possible C declarations. The only difference is that**/
/** the declarations are not allowed to have initializations in them.    **/
```

```
condexpr : val
```

```

        | val relop val

exprseq : exprseq ',' expr
        | expr

casestmts : '{' cases default'}'
          | ';'
          | '{' '}'

cases : cases case
      | case

case : CASE INT_CONST ':' stmtlist stop_stmt

default : DEFAULT ':' stmtlist stop_stmt

stop_stmt : BREAK ';'
          | CONTINUE ';'
          | RETURN ';'
          | RETURN val ';'
          | RETURN '(' val ')' ';'

expr : modify_expr
     | rhs

modify_expr : varname '=' rhs
            | '*' ID '=' rhs

rhs : unary_expr
    | binary_expr

unary_expr : simp_expr
           | '*' ID
           | '&' varname
           | call_expr
           | unop val
           | '(' cast ')' varname

```

```

/** cast here stands for all valid C typecasts */

simp_expr : varname
           | INT_CONST
           | FLOAT_CONST
           | STRING_CONST

call_expr : ID '(' arglist ')'

arglist : arglist ',' val
         | val
         |

unop : '+'
      | '-'
      | '('
      | '~'

binary_expr : val binop val

binop : relop
       | '-' | '+' | '/' | '*' | '%'
       | '&' | '|' | '<<' | '>>' | '^'

relop : '<' | '<=' | '>' | '>=' | '==' | '!='

varname : arrayref
         | compref
         | ID

arrayref : ID reflist

reflist : '[' val ']'
         | reflist '[' val ']'

val : ID
     | CONST

```

```
compref : '(' '*' ID ')' '.' idlist  
         | idlist
```

```
idlist : idlist '.' ID  
        | ID
```



## **Appendix B**

# **The Interprocedure Algorithms**

This appendix is devoted to: (i) the depth-first algorithm for building invocation graph, and, (ii) the precise map process algorithm. These are given in the following:

```

/*
 * Function name : build_invocation_graph
 * Purpose : building an invocation graph for the program
 * Parameters : main_func -- the node related to 'main' function
 *              in the program
 * Return : init_ig_node -- the root of invocation graph
 */
build_invocation_graph( main_func)
{
    /* build the first invocation graph node related to the 'main' function
     * which, at this point, has neither parents nor children */
    init_ig_node = make_ig_node ( null, main_func ) ,

    /* get the list of the functions that get called by the 'main' */
    init_func_lst = get_func_lst( main_func ) ,

    /* generate the rest of the invocation graph */
    gen_ig( init_ig_node, init_func_lst) ;

    return( init_ig_node) ,
}

/*
 * Function name : gen_ig
 * Purpose : recursively build the invocation graph, in depth first
 *           manner
 * Parameters : ig_node -- the current invocation graph node
 *              func_lst -- the list of functions called by the
 *              related function to 'ig_node'
 */
gen_ig( ig_node, func_lst)
{
    for each function call 'func' in the 'func_lst' do
    {
        /* make a new invocation graph node related to function 'func'
         * and assign 'ig_node' as its parent */
        new_ig_node = make_ig_node( ig_node, func ) ,

        /* getting the list of functions that 'func' is calling */
        new_func_lst = get_func_lst( func ) ,

        /* call the same function recursively to build the invocation
         * graph in the depth-first manner */
        gen_ig( new_ig_node, new_func_lst) ,
    }
}

```

```

/*
 * Function name : make_ig_node
 * Purpose : allocate an invocation graph node corresponding to func_call
 *           and define 'parent_ig_node' as the parent
 * Parameters : parent_ig_node -- the parent node of newly generated
 *                               invocation graph node
 *             func_call -- the function call related to the newly
 *                           generated invocation graph node
 * Return : new_ig_node -- a newly generated invocation graph node
 */
make_ig_node( parent_ig_node, func_call)
{
    /* assign the related function call to 'new_ig_node' */
    new_ig_node.func = func_call ,

    /* assign the parent (for the root 'parent' would be null) */
    new_ig_node.parent = parent_ig_node ,

    /* add the newly generated node as the child of the parent node
     * (if the 'parent_ig_node' is not the root).
     * Note: the invocation graph is an invocation tree with some
     * backward edges. For the efficiency purpose, an invocation tree
     * is implemented as a binary tree, each node has one child and
     * one sibling. The sibling of a node has the same parent as
     * the node. Following shows an example of this representation
     *
     *      main                main
     *      / | \      -- represented as --> /
     *      f g h                f__g__h
     *
     * It is only for the ease of understanding that we
     * do not limit the number of children of a node */
    if ( parent_ig_node != null)
        add_child( parent_ig_node, new_ig_node) ,

    /* check if the function call 'func_call' is a recursive call */
    recursive_node = get_recursive_func( new_ig_node) ,

    if (recursive_node == NULL)
        new_ig_node.mode = "ordinary" ,
    else{
        /* the func call is a recursive-call */
        new_ig_node.mode = "approximate" ;
        recursive_node.mode = "recursive" ,
        new_ig_node.approximate_edge = recursive_node ,
    }
    return (new_ig_node) ,
}

```

```

/* _____
 * Function name : get_recursive_func
 * Purpose : check if the given function call related to 'cg_node'.
 *           is a recursive call (it already occurs in the path back
 *           to the root of invocation graph).
 * Parameters  cg_node -- current invocation graph node
 * Return  NULL is returned if the related function to 'cg_node' does not
 *         occur in the path back to the root of the invocation graph.
 *         otherwise, the related invocation graph node is returned.
 * _____ */
get_recursive_func( cg_node)
{
    /* get the function call related to the 'cg_node' */
    func = cg_node.func ;

    /* check all the path back to the root of invocation graph,
     * to check if the same function call has already occurred */
    while (cg_node.parent != NULL){
        cg_node = cg_node.parent ;
        if (cg_node.func == func)
            return (cg_node) ;
    }

    return (NULL) ,
}

```

```

/*
 * Function name : map_process
 * Purpose : our accurate map process
 * Parameters  func_node -- the function declaration node of callee
 *              arg_lst -- argument list
 *              caller_in -- the points-to information of caller
 * Return  map_info -- a set of relationship between invisible
 *                   variables of callee and local variables of caller
 *                   callee_in -- the points-to information entering to callee
 * Note: throughout this algorithm we use D for definitely-points-to and
 *       P for possibly-points-to relationship
 */
map_process( func_node, arg_lst, caller_in)
{
    /* initialization */
    map_info = { } ;
    callee_in = { } ;
    map_func_ptr = 0 , /* a function pointer */

    /* the accurate map process is done in four stages 'map_func_ptr' is
     * pointing to the related function that has to be called in each stage */

    /* first step: giving names to invisible variables of structure type
     * accessible through D relationships. */
    map_func_ptr = &map_naming_definitely_struct_process ,
    [map_info, callee_in] =
    map_step( func_node, arg_lst, caller_in, map_info, callee_in, map_func_ptr)

    /* second step: giving names to invisible variables of structure type
     * accessible through P relationships. */
    map_func_ptr = &map_naming_possibly_struct_process ,
    [map_info, callee_in] =
    map_step( func_node, arg_lst, caller_in, map_info, callee_in, map_func_ptr)

    /* third step: giving names to invisible variables accessible through
     * D relationships, and setting all the D relationships of callee */
    map_func_ptr = &map_definitely_process ,
    [map_info, callee_in] =
    map_step( func_node, arg_lst, caller_in, map_info, callee_in, map_func_ptr)

    /* fourth step. giving names to invisible variables accessible through
     * P relationships, and setting all the P relationships of callee */
    map_func_ptr = &map_possibly_process ,
    [map_info, callee_in] =
    map_step( func_node, arg_lst, caller_in, map_info, callee_in, map_func_ptr)

    return( [map_info, callee_in] ) ,
}

```

```

/*
 * Function name  map_step
 * Purpose : this is an intermediate step for the map process to apply
 *           different steps of the map process.
 * Parameters : func_node -- the function declaration node of callee
 *               arg_lst -- argument list
 *               caller_in -- the points-to information of caller
 *               map_info -- a set of relationship between invisible
 *                           variables of callee and local variables
 *                           of caller.
 *               callee_in -- the points-to information entering to callee
 *               map_func_ptr -- a pointer to the function that has to
 *                               be called in the map_step
 * Return  map_info -- the updated version
 *         callee_in -- the updated version
 */
map_step( func_node, arg_lst, caller_in, map_func_ptr)
{
    param_lst = get_param_lst( func_node) ;

    /* doing the map process for each pointer type parameter */
    for each 'param_i' in 'param_lst' and 'arg_i' in 'arg_lst' of pointer type do
        /* 'callee_in' and 'map_info' will be updated. The argument
         * corresponding to 1 is the depth of the pointer type. For
         * the first call, depth is 1 */
        [map_info, callee_in] =
        map_func_ptr( param_i, arg_i, callee_in, caller_in, 1, map_info)

    /* doing the map process for each global variable of pointer type */
    for each global variable 'var_i' of pointer type do
        /* 'callee_in' and 'map_info' will be updated. The argument
         * corresponding to 1 is the depth of the pointer type. For
         * the first call, depth is 1. */
        [map_info, callee_in] =
        map_func_ptr( var_i, var_i, callee_in, caller_in, 1, map_info) .

    return( [map_info, callee_in]) ,
}

```

```

/*
 * Function name : map_definitely_process
 * Purpose : map the invisible names and the points-to relationships
 *           resolved from the D relationship(s) of 'caller_var'
 *           This function is recursively called to assign the D
 *           relationships of all the indirectly accessible variables
 *           through 'caller_var'.
 * Purpose : map the invisible names and the D relationships
 *           resolved from 'caller_var'. This function is recursively
 *           called to assign the D information of all the
 *           indirectly accessible variables through caller_var
 *           It also updates the map information
 * Parameters : callee_var -- the variable in callee
 *             caller_var -- the related variable in caller
 *             callee_in -- points-to information of callee
 *             caller_in -- points-to information of caller
 *             depth -- the depth of current invisible variable
 *                    (for the first call, it is 1).
 *             map_info -- the set of map information
 * Return : map_info -- the updated set of map information
 *         callee_in -- the updated set of points-to information
 *                of callee
 */
map_definitely_process( callee_var, caller_var, callee_in,
                       caller_in, depth, map_info)
{
    for each 'x_caller' that relationship '(caller_var, x_caller, rel)' exist do
    /* 'rel' can be either D or P */
    {
        /* check if a D relationship with a variable exists */
        if (caller_var, x_caller, D)
        {
            if (depth > 1)
            /* only in the first call 'depth' is 1. Since parameters in C
             * are passed by value, they should not be mapped to each other.
             * When 'depth' is more than 1, variables are resolved from
             * a parameter and should be mapped. */
            add_map_info( caller_var, callee_var, map_info) ,

            if (is_in_callee_scope(x_caller))
                x_callee = x_caller ;
            else{ /* variable 'x_caller' is not in the scope of callee */

```

```

/* look at 'map_info' set to check if an invisible variable
 * is already assigned to the variable 'x_caller' */
if (exist_invisible_for( x_caller, map_info))
    /* if an invisible variable is already assigned to variable
     * 'x_caller', get that variable and use the same name for
     * the invisible variable. */
    x_invisible = get_invisible_var( x_caller, map_info) ;

else{ /* an invisible variable does not exist for 'x_caller' */
    /* get an invisible variable for 'x_caller' using 'depth'
     * and 'callee_var'. If the name of variable 'callee_var'
     * is 'data' and 'depth' is 1, the invisible variable
     * would be '1_data', for 'depth' 2, the invisible variable
     * would be '2_data', and etc. */
    x_invisible = define_invisible( callee_var, depth) ;

    /* set the equivalency of 'x_invisible' and 'x_caller' in
     * 'map_info'. This information is used by unmap process.
     add_map_info( x_invisible, x_caller, map_info) ;
    }
    x_callee = x_invisible ;
}

/* add a relationship between 'callee_var' and 'x_callee' in
 * 'callee_in'. Since this function is a recursive function,
 * 'callee_var' could be an invisible variable that already has
 * some other relationship. Therefore, we can not simply assign
 * the relationship (callee_var, x_callee, D). */

/* check if 'callee_var' has a D relationship with a variable
 * except 'x_callee' */
if (exist_another_definitely_points_to( callee_in, callee_var, x_callee))
{
    /* This case appears when 'callee_var' is an invisible
     * variable (which can not happen in the first call to this
     * function) and stands for more than one variable, e.g. 'a'
     * and 'b', that one of them has a D relationship with a
     * variable other than 'x_callee'. Since D should be always
     * unique, we should change other D relationships to P
     * relationship and assign a P relationship between
     * 'callee_var' and 'x_callee' */
    change_D_relation_to_P_relation( callee_in, callee_var) ;
    callee_in = callee_in  $\cup$  {(callee_var, x_callee, P)} ;
}
else
    callee_in = callee_in  $\cup$  {(callee_var, x_callee, D)}
}

```



```

/* recursively check all the relationships of the variables which can
 * be derived from 'caller_var'. In this case, 'x_caller' is the first
 * of such variables
 * As 'x_caller' is one level deeper, depth is increased by one */
[map_info, callee_in] =
map_definitely_process( x_callee, x_caller, callee_m, caller_m, depth+1)

if (is_struct_type( x_caller))
/* if 'x_caller' is of structure type, all its fields has to be
 * processed, because they might be of pointer type */
for each field 'fi' of 'x_caller'
{
    x_callee = get_related_callee_variable(x_caller, callee_m,
                                             caller_m, map_info, fi)

/* 1 represents the depth. 1 is chosen because the depth of an
 * invisible variable related to a field should be started
 * from one */
[map_info, callee_in] =
map_definitely_process( x_callee, x_caller, callee_m, caller_m, 1)
}
}
return([map_info, callee_in]) ;
}

```

```

/*
 * Function name : map_possibly_process
 * Purpose  map the invisible names and the points-to relationships
 *           resolved from the P relationship(s) of 'caller_var'
 *           This function is recursively called to assign the P
 *           relationships of all the indirectly accessible variables
 *           through 'caller_var'.
 * Parameters  same as function 'map_definitely_process'
 * Return  same as function 'map_definitely_process'
 * Note  Since this function is similar to 'map_definitely_process',
 *        we do not explain the common parts
 */
map_possibly_process( callee_var, caller_var, callee_in,
                     caller_in, depth, map_info)
{
    for each 'x_caller' that relationship '(caller_var, x_caller, rel)' exists do
    /* 'rel' can be either D or P relationship. */
    {
        /* check if a P relationship with a variable exists */
        if (callee_var, x_caller, P)
        {
            if (depth > 1)
                add_map_info( caller_var, callee_var, map_info) .

            if (is_in_callee_scope(x_caller))
                x_callee = x_caller ,
            else { /* variable 'x_caller' is not in the scope of callee */

                if (exist_invisible_for( x_caller, map_info))
                    x_invisible = get_invisible_var( x_caller, map_info)

                else { /* an invisible variable does not exist for 'x_caller' */
                    x_invisible = define_invisible( callee_var, depth)
                    add_map_info( x_invisible, x_caller, map_info) .
                }
                x_callee = x_invisible ,
            }
        }
    }
}

```

```

/* add a relationship between 'callee_var' and 'x_callee_in'
 * 'callee_in' Since this function is a recursive function
 * 'callee_var' could be an invisible variable that has already
 * taken some D relationship Therefore, we can not simply assign
 * the relationship (callee_var, x_callee, P) */

/* check if a D relationship exists for 'callee_var' */
if (exist_definitely_points—to( callee_in, callee_var))
    /* This can happen when 'callee_var' is an invisible variable
     * that stands for more than one variable where one of them
     * has already a D relationship Since D should be always
     * unique, we should change the already exist D relationship
     * to a P relationship */
    change_D_relation_to_P_relation( callee_in, callee_var)

/* add the new relationship in 'callee_in' */
callee_in = callee_in  $\cup$  {(callee_var, x_callee, P)} ,
}

[map_info, callee_in] =
map_possibly_process( x_callee, x_caller, callee_in, caller_in_depth+1)

if (is_struct_type( x_caller))
    for each field 'fi' of 'x_caller'
    {
        x_callee = get_related_callee_variable(x_caller, caller_in,
                                                    callee_in, map_info, fi)

        [map_info, callee_in] =
        map_possibly_process( x_callee, x_caller, callee_in, caller_in+1)
    }
}

return([map_info, callee_in]) ,
}

```

```

/*
 * Function name   map_naming_definitely_struct_process
 * Purpose   . assigning names for invisible variables of structure type
 *            which already have a D relationship. This function is recursively
 *            called to assign names for all the structure type variables
 * Parameters   same as function 'map_definitely_process'.
 * Return   . same as function 'map_definitely_process'
 * Note: Since this function is similar to map_definitely_process, we do
 *        not explain the common parts.
 */
map_naming_definitely_struct_process( callee_var, caller_var, callee_in,
                                     caller_in, depth, map_info)
{
    for each 'x_caller' with the relationship '(caller_var, x_caller, rel)' do
    /* 'rel' can be either D or P relationship. */
    {
        /* check if a D relationship with a structure type variable exists */
        if ((caller_var, x_caller, D) and (is_struct_type( x_caller)))
        {
            if (is_in_callee_scope(x_caller))
                x_callee = x_caller ;
            else{ /* variable 'x_caller' is not in the scope of callee */
                if (exist_invisible_for( x_caller, map_info))
                    x_invisible = get_invisible_var( x_caller, map_info) ;
                else{ /* an invisible variable does not exist for 'x_caller' */
                    x_invisible = define_invisible( callee_var, depth) ;
                    /* set the equivalency of 'x_invisible' and 'x_caller' in 'map_info'. This
                     * information is used by unmap process. It also sets the equivalency of
                     * all the fields of 'x_invisible' and 'x_caller'. */
                    add_map_info( x_invisible, x_caller, map_info) ;
                }
                x_callee = x_invisible ;
            }
        }
        [map_info, callee_in] =
        map_naming_definitely_struct_process( x_callee, x_caller, callee_in, caller_in, depth+1)
        if (is_struct_type( x_caller))
            for each field 'fi' of 'x_caller'
            {
                x_callee = get_related_callee_variable(x_caller, caller_in, callee_in, map_info, fi)
                [map_info, callee_in] =
                map_naming_definitely_struct_process( x_callee, x_caller, callee_in, caller_in, fi)
            }
    }
    return([map_info, callee_in]) .
}

```

```

/*
 * Function name  map_naming_possibly_struct_process
 * Purpose  assigning names for invisible variables of structure type
 *           that a P relationship to them exist. This function is
 *           recursively called to assign names for all the structure type
 *           variables.
 * Parameters  same as function 'map_definitely_process'
 * Return  . same as function 'map_definitely_process'
 * Note: Since this function is similar to map_definitely_process and
 *       map_naming_definitely_struct_process, we do not explain the
 *       common parts
 */
map_naming_possibly_struct_process( callee_var, caller_var, callee_in,
                                   caller_in, depth, map_info)
{
    for each 'x_caller' that relationship '(caller_var, x_caller, rel)' exist do
    /* 'rel' can be either D or P relationship */
    {
        /* check if a P relationship with a structure type variable exists */
        if ((caller_var, x_caller, P) and (is_struct_type( x_caller)))
        {
            if (is_in_callee_scope(x_caller))
                x_callee = x_caller ;
            else{ /* variable 'x_caller' is not in the scope of callee */
                if (exist_invisible_for( x_caller, map_info))
                    x_invisible = get_invisible_var( x_caller, map_info)

                else{ /* an invisible variable does not exist for 'x_caller' */
                    x_invisible = define_invisible( callee_var, depth)
                    add_map_info( x_invisible, x_caller, map_info) .
                }
                x_callee = x_invisible .
            }
        }
        [map_info, callee_in] =
        map_naming_possibly_struct_process( x_callee, x_caller, callee_in, caller_in, depth + 1)
        if (is_struct_type( x_caller))
            for each field 'fi' of 'x_caller'
            {
                x_callee = get_related_callee_variable(x_caller, caller_in, callee_in, map_info, fi)
                [map_info, callee_in] =
                map_naming_possibly_struct_process( x_callee, x_caller, callee_in, caller_in, fi)
            }
    }
    return([map_info, callee_in]) .
}

```

## Appendix C

### Rules for the Basic Cases

In this appendix, we give some of the basic rules which were discussed in Section 5.3. For the sake of conciseness, we just explain one subcase (or two subcases if necessary). The case numbers refer to Table 5.1.

**Case 2.5:** The general format of this case is:

```
int *x[size1], *y[size2] ;  
.  
.  
.  
x[i] = y[j]
```

The rules applied in this case are as follows:

```
kill = { (x,x1,D) | (x,x1,D) ∈ input ∧ first_elem(x[i]) = D }  
gen = { (x,y1,rel ⋈ first_elem([i]) ⋈ first_elem([j])) | (y,y1,rel) ∈ input }  
changed_input = (input - { (x,x1,D) | (x,x1,D) ∈ input ∧ first_elem([i]) = D })  
                ∪ { (x,x1,P) | (x,x1,P) ∈ input ∧ first_elem([i]) = P }  
return( gen ∪ (changed_input - kill) )
```

Consider the following example:

```

f(int i) {
    int *x[70], *y[50] ;
    int a, b ;

    x[0] = &a ;      /* stmt1 */
    y[i] = &b ;      /* stmt2 */
    x[0] = y[0] ;    /* stmt3 */
}

```

Case 2.5 appears at statement 3. Using the given rules, the sets related to this statement are as follows:

```

input = {(x,a,D), (y,b,P)}
kill = {(x,a,D)}
gen = {(x,b,P)}
changed input = input
output = {(x,b,P), (y,b,P)}

```

**Case 3.2:** The general format of this case is:

```

struct{
    int *b ;
} *y ;
int *x ;
. . .
x = (*y).b ;

```

The rules applied in this case are as follows:

```

kill = { (x,x1,rel) | (x,x1,rel) ∈ input }
gen = { (x,y2,rel1 ⋈ rel2) | (y,y1,rel1), (y1.b,y2,rel2) ∈ input }
return( gen ∪ (input - kill) )

```

Consider the following example:

```

main()
{
    struct{
        int *f1 ;
    } *y, z ;
    int *x ;
    int a ;

    z.f1 = &a ;    /* stmt 1 */
    y = &z ;        /* stmt 2 */
    x = (*y).f1 ; /* stmt 3 */
}

```

Case 3.2 appears at statement 3. Using the given rules, the sets related to this statement are as follows:

```

input = {(z.f1,a,D), (y,z,D)}
kill = { }
gen = {(x,a,D)}
output = {(x,a,D), (z.f1,a,D), (y,z,D)}

```

**Case 4.4:** The general format of this case is:

```

struct{
    int *a ;
} *x ;
int y ;
. . .
(*x).a = &y ;

```

The rules applied in this case are as follows:

```

kill = { (x1.a,x2,rel) | (x,x1,D), (x1.a,x2,rel) ∈ input }
gen = { (x1.a,y,rel) | (x,x1,rel) ∈ input }
changed_input = (input - { (x1.a,x2,D) | (x,x1,P), (x1.a,x2,D) ∈ input })
                ∪ { (x1.a,x2,P) | (x,x1,P), (x1.a,x2,D) ∈ input }
return( gen ∪ (changed_input - kill) )

```

Consider the following example:



```

main()
{
    struct{
        int *f1 ;
    } *x, y ;
    int a ;

    x = &y ;          /* stmt 1 */
    (*x).f1 = &a ; /* stmt 2 */
}

```

Case 4.4 appears at statement 2. Using the given rules, the sets related to this statement are as follows:

```

input = {(x,y,D)}
kill = { }
gen = {(y.f1,a,D)}
changed_input = input = {(x,y,D)}
output = {(y.f1,a,D), (x,y,D)}

```

**Case 5.6:** The general format of this case is:

```

struct{
    int *a ;
    int *b ;
} *x, y ;
. . .
(*x).a = y.b ;

```

The rules applied in this case are as follows:

```

kill = { (x1.a,x2,rel) | (x,x1,D), (x1.a,x2,rel) ∈ input }
gen = { (x1.a,y1,rel1 ⋈ rel2) | (x,x1,rel1), (y.b,y1,rel2) ∈ input }
changed_input = (input - { (x1.a,x2,D) | (x,x1,P), (x1.a,x2,D) ∈ input })
                ∪ { (x1.a,x2,P) | (x,x1,P), (x1.a,x2,D) ∈ input }
return( gen ∪ (changed_input - kill) )

```

Consider the following example:

```

main()
{
    struct{
        int *f1 ;
        int *f2 ;
    } *x, y, z ;
    int a ;

    y.f2 = &a ;      /* stmt 1 */
    if (...)
        x = &z ;      /* stmt 2 */
    (*x).f1 = y.f2 ; /* stmt 3 */
}

```

Case 5.6 appears at statement 3. Using the given rules, the sets related to this statement are as follows:

```

input = {(x,z,P), (y.f2,a,D)}
kill = { }
gen = {(z.f1,a,P)}
changed_input = input = {(x,z,P), (y.f2,a,D)}
output = {(z.f1,a,P), (x,z,P), (y.f2,a,D)}

```

**Case 5.7:** The general format of this case is:

```

int **px , *y ;
. . .
(*px)[i] = y ;

```

The rules applied in this case are as follows:

```

kill = { (x1,x2,D) | (px,x1,D), (x1,x2,D) ∈ input ∧ first_elem([i])=D }
gen = {(x1,y1,rel1 ∞ rel2 ∞ first_elem([i])) | (px,x1,rel1), (y,y1,rel2) ∈ input }
changed_input = (input - { (x1,x2,D) | [(px,x1,P), (x1,x2,D) ∈ input],
                        [(px,x1,D), (x1,x2,D) ∈ input ∧ first_elem([i])=P] })
                ∪ { (x1,x2,P) | [(px,x1,P), (x1,x2,D) ∈ input] ∨
                        [(px,x1,D), (x1,x2,D) ∈ input ∧ first_elem([i])=P] }
return( gen ∪ (changed_input - kill) )

```

Consider the following example:

```
f(int i) {
    int **px, *y, z ;
    int *a[70], b ;

    a[i] = &b ;      /* stmt1 */
    px = &a[0] ;     /* stmt2 */
    y = &z ;         /* stmt3 */
    px[0] = y ;      /* stmt4 */
}
```

Case 5.7 appears at statement 4. Since **px** at statement 4 is a pointer to an array we treat **px[0]** as **(\*px)[0]**. Consequently, statement 4 is of the type 5.7. Using the given rules, the sets related to this statement are as follows.

```
input = {(a,b,P), (px,a,D), (y,z,D)}
kill = { }
gen = {(a,z,P)}
changed_input = input
output = {(a,b,P), (a,z,P), (px,a,D), (y,z,D)}
```

**Case 6.2:** The general format of this case is:

```
struct{
    int *b ;
} *y ;
int **x ;
. . .
*x = (*y).b ;
```

The rules applied in this case are as follows:

```
kill = { (x1,x2,rel) | (x,x1,D), (x1,x2,rel) ∈ input }
gen = { (x1,y2,rel1 ∞ rel2 ∞ rel3) | (x,x1,rel1), (y,y1,rel2), (y1,b,x2,rel3)
      ∈ input }
changed_input = (input - { (x1,x2,D) | (x,x1,P), (x1,x2,D) ∈ input })
               ∪ { (x1,x2,P) | (x,x1,P), (x1,x2,D) ∈ input }
return( gen ∪ (changed_input - kill) )
```

Consider the following example:

```
main()
{
    struct{
        int *f1 ;
    } *y, z ;
    int **x ;
    int *a, b, c ;

    z.f1 = &c ;    /* stmt 1 */
    y = &z ;        /* stmt 2 */
    a = &b ;         /* stmt 3 */
    if (...)
        x = &a ;    /* stmt 4 */
    *x = (*y).f1 ; /* stmt 5 */
}
```

Case 6.2 appears at statement 5. Using the give rules, the sets related to this statement are as follows:

```
input = {(z.f1,c,D), (y,z,D), (a,b,D), (x,a,P)}
kill = { }
gen = {(a,z.f1,P)}
changedinput = {(z.f1,c,D), (y,z,D), (a,b,P), (x,a,P)}
output = {(a,z.f1,P), (z.f1,c,D), (y,z,D), (a,b,P), (x,a,P)}
```

Cases 7 and 8 did not appear in the previous chapter because they can not be broken down in to smaller cases (the reason was explained in Chapter 2).

**Case 6.3:** The general format of this case is:

```
int **x, **py ;
. . .
*x = (*py)[1] ;
```

The rules applied in this case are as follows:

```
kill = { (x1,x2,rel) | (x,x1,D), (x1,x2,rel) ∈ input }
gen = { (x1,y2,rel1 ∞ rel2 ∞ rel3 ∞ first_elem([i])) |
        [(x,x1,rel1), (py,y1,rel2), (y1,y2,rel3) ∈ input] }
```

```

changed_input = (input - { (x1,x2,D) | (x,x1,P), (x1,x2,D) ∈ input } )
                ∪ { (x1,x2,P) | (x,x1,P), (x1,x2,D) ∈ input }
return( gen ∪ (changed_input - kill) )

```

Consider the following example:

```

f(int i) {
    int **x, **py ;
    int *z, w ;
    int *a[70], b ;

    x = &z ;           /* stmt1 */
    z = &w ;           /* stmt2 */
    py = &a[0] ;       /* stmt3 */
    a[0] = &b ;        /* stmt4 */
    *x = py[i] ;       /* stmt5 */
}

```

Case 6.3 appears at statement 5 in the form of `py[i]`. Since `py` is a pointer to an array, we treat this statement as `*x=(*py)[i]`. Using the given rules, the sets related to this statement are as follows:

```

input = {(x,z,D), (z,w,D), (py,a,D), (a,b,D)}
kill = {(z,w,D)}
gen = {(z,b,P)}
changed input = input
output = {(x,z,D), (z,b,P), (py,a,D), (a,b,D)}

```

**Case 7.1:** The general format of this case is:

```

struct{
    int *b ;
} *y ;
int *x ;
. . .
x = & (*y).b ;

```

The rules applied in this case are as follows:

```

kill = { (x,x1,rel) | (x,x1,rel) ∈ input }

```

```

gen = { (x,y1.b,rel) | (y,y1,rel) ∈ input }
return( gen ∪ (input - kill) )

```

Consider the following example:

```

main()
{
    struct{
        int f1 ;
    } *y, z ;
    int *x ;

    y = &z ;          /* stmt 1 */
    x = & (*y).f1 ; /* stmt 2 */
}

```

Case 7.1 appears at statement 2. Using the above rules, the sets related to this statement are as follows:

```

input = {(y,z,D)}
kill = { }
gen = {(x,z.f1,D)}
output = {(x,z.f1,D), (y,z,D)}

```

**Case 8.1:** The general format of this case is:

```

struct{
    int b ;
} *y ;
int **x ;
. . .
*x = & (*y).b ;

```

The rules applied in this case are as follows:

```

kill = { (x1,x2,rel) | (x,x1,D), (x1,x2,rel) ∈ input }
gen = { (x1,y1.b,rel1 ∞ rel2) | (x,x1,rel1), (y,y1,rel2) ∈ input }
changed_input = (input - { (x1,x2,D) | (x,x1,P), (x1,x2,D) ∈ input })
                ∪ { (x1,x2,P) | (x,x1,P), (x1,x2,D) ∈ input }
return( gen ∪ (changed_input - kill) )

```

Consider the following example:

```
main()
{
    struct{
        int *f1 ;
        int f2 ;
    } *x, y, *z, w ;
    int a, b ;

    x = &y ;           /* stmt 1 */
    y.f1 = &a ;         /* stmt 2 */
    if (...)
        z = &w ;       /* stmt 3 */
    w.f1 = &b ;         /* stmt 4 */
    (*x).f1 = & (*z).f2 ; /* stmt 5 */
}
```

Case 8.1 appears at statement 4. Using the above rules, the sets related to this statement are as follows:

input = {(x,y,D), (y.f1,a,D), (z,w,P), (w.f1,b,D)}

kill = {(y.f1,a,D)}

gen = {(y.f1,w.f2,P)}

changed\_input = input

output = {(y.f1,w.f2,P), (x,y,D), (z,w,P), (w.f1,b,D)}

## Bibliography

- [Amm92] Z. Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(3):237–251, 1992.
- [ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers—Principles, Techniques, and Tools*. Addison-Wesley Publishing Co., 1986.
- [Bak77] B. Baker. An algorithm for structuring flowgraphs. *JACM*, 24(1):98–120, 1977.
- [Ban76] U. Banerjee. Data dependence in ordinary programs. Master's thesis, University of Illinois at Urbana-Champaign, 1976.
- [Ban79] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*. ACM SIGACT and SIGPLAN, January 1979.
- [Bar78] J. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21, 1978.
- [BC86] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 162–175, Palo Alto, California, June 25–27, 1986. ACM SIGPLAN. Also in *SIGPLAN Notices*, 21(7), July 1986.
- [CBC93] J. D. Choi, M. G. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, January 1993.
- [CHK92] K. D. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of the IEEE'92 International Conference on Computer Languages*, pages 96–105, April 1992.
- [CK89] K. D. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles*



- of *Programming Languages*, pages 49-59, Austin, Texas, January 11-13 1989. ACM SIGACT and SIGPLAN.
- [Coo85] K. Cooper. Analyzing aliases of reference formal parameters. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 281-290. ACM SIGACT and SIGPLAN, January 1985.
  - [Coo89] B. G. Cooper. Ambitious data flow analysis of procedural programs. Master's thesis, University of Minnesota, May 1989.
  - [Cou86] D. S. Coutant. Retargetable high-level alias analysis. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 110-118, January 1986.
  - [CR82] A. L. Chow and A. Rudmik. The design of a data flow analyzer. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 106-113, June 1982.
  - [CWZ90] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296-310. White Plains, New York, June 20-22, 1990. ACM SIGPLAN. Also in *SIGPLAN Notices*, 25(6), June 1990.
  - [Deu92] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the IEEE'92 International Conference on Computer Languages*, April 1992.
  - [Don93] C. M. Donawa. A structured approach for the design and implementation of a backend for the McCAT C compiler. Master's thesis, McGill University, expected December 1993.
  - [Ero93] A. Erosa. A program structurer for the McCAT C compiler. Master's thesis, McGill University, expected December 1993.
  - [Ghi92] R. Ghiya. Interprocedural analysis in the presence of function pointers. Technical report, McGill University, December 1992. ACAPS Technical Memo 62.
  - [Ghi93] R. Ghiya. Practical techniques for interprocedural heap analysis. Report for 308-622B project with Prof. G. Gao, 1993.
  - [HDE<sup>+</sup>92] L. J. Hendren, C. Donawa, M. Emami, G. R. Gao, Justam, and B. Sridharan. Designing the McCAT compiler based on a family of structured

- intermediate representations. In *Fifth Workshop on Languages and Compilers for Parallel Computing*, pages 261–275. Yale University, 1992. Also available as ACAPS Technical Memo No. 46.
- [HEGV93] L. J. Hendren, M. Emami, R. Ghiya, and C. Verbrugge. A practical interprocedural analysis framework for C compilers. Technical report, 1993. ACAPS Technical Memo 72.
  - [Hen90] L. J. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell University, January 1990.
  - [HGS92] L. Hendren, G. R. Gao, and V. C. Sreedhar. ALPHA: A family of structured intermediate representations for a parallelizing C compiler. ACAPS Technical Memo 49, School of Computer Science, McGill University, Montréal, Québec, November 1992.
  - [HHN92a] L. J. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 249–260. San Francisco, California, June 17–19, 1992. ACM SIGPLAN. Also in *SIGPLAN Notices*, 27(7), July 1992.
  - [HHN92b] J. Hummel, L.J. Hendren, and A. Nicolau. Applying an abstract data structure description to parallelizing scientific pointer programs. In *Proceedings of the 1992 International Conference on Parallel Processing*, St Charles, Illinois, August 17–21, 1992.
  - [HN89] L. J. Hendren and A. Nicolau. Interference analysis tools for parallelizing programs with recursive data structures. In *Conference Proceedings, 1989 International Conference on Supercomputing*, pages 205–211. Crete, Greece, June 5–9, 1989. ACM.
  - [HN90] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1), January 1990.
  - [HS92] L. Hendren and B. Sridharan. The SIMPLE AST - McCAI compiler. Technical report, October 1992. ACAPS Design Note 36.
  - [JM81] N. D. Jones and S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Program Flow Analysis, Theory, and Applications*, pages 102–131. Prentice-Hall, 1981. Chapter 4.
  - [JM82] N. D. Jones and S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Conference*

*Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 66–74, 1982.

- [Jus93] Justiani. An array dependence framework for the McCAT C compiler. Master's thesis, McGill University, expected December 1993.
- [Lan92] W. A. Landi. *Interprocedural Aliasing in the Presence of Pointers*. PhD thesis, Rutgers University, 1992.
- [LH88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–31, Atlanta, Georgia, June 22–24, 1988. ACM SIGPLAN. Also in *SIGPLAN Notices* 23(1) July 1988.
- [LH93] C. Verbrugge L. Hendren. Generalized constant propagation. ACAPS Technical Memo 71, School of Computer Science, McGill University, Montréal, Québec, expected September 1993.
- [LR92] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the 1992 ACM Symposium on Programming Language Design and Implementation*, 1992.
- [Mye81] E. W. Myers. A precise inter procedural data flow algorithm. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*. ACM SIGACT and SIGPLAN, January 1981.
- [Sri92] B. Sridharan. An analysis framework for the McCAT compiler. Master thesis, McGill University, Montréal, Québec, 1992.
- [Sta90] R. M. Stallman. Using and porting the GNU CC. Technical report Free Software Foundation, Cambridge, Massachusetts, 1990.
- [Wei80] W. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*. ACM SIGACT and SIGPLAN, January 1980.
- [WO75] M. H. Williams and H.L. Ossher. Conversion of unstructured flow diagram to structured form. *Comput. J.*, 21(2), 1975.
- [Wol89] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman, London and MIT Press, Cambridge, MA, 1989. In the series: Research Monographs in Parallel and Distributed Computing. Revised version of the author's Ph.D. dissertation, Published as Technical Report UIUCDCS R 82-1105, University of Illinois at Urbana-Champaign, 1982.

- [ZC90] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, 1990.