PRACTICAL POINTER ALIASING ANALYSIS

BY XIANG-XIANG SEAN ZHANG

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Barbara Gershon Ryder

and approved by

____

____

____

____

New Brunswick, New Jersey

October, 1998

ABSTRACT OF THE DISSERTATION

Practical Pointer Aliasing Analysis

by XIANG-XIANG SEAN ZHANG

Dissertation Director: Barbara Gershon Ryder

Two names are aliased if they refer to the same location at a program point during execution. Pointer aliasing analysis for C programs is essential for compile-time analyses and optimizations. Many techniques have been proposed in the literature. Some are fast, but not so precise; some are quite precise, but not fast in some cases.

We have developed a technique which decomposes program statements into independent sets in terms of their effects on pointer aliasing. Each set implies a program segment which includes the statements in the set and some control statements. Each segment can be analyzed for pointer aliasing independently. Therefore the program decomposition allows the use of more than one analysis algorithm on a same program and the use of appropriate aliasing analyses based on characteristics of pointers being analyzed. We handle features in C such as function pointers, indirect calls through functions pointers, unions, and type casting with some restrictions.

We have also developed a points-to analysis algorithm and an aliasing analysis algorithm. Both are flow-insensitive and context-insensitive; thus they are efficient and can be used on large programs. For each assignment lhs=rhs in a program, the aliasing analysis effectively assumes there is also an assignment rhs=lhs while the points-to analysis does not make such assumptions. Therefore the points-to analysis is more precise than the aliasing analysis.

We have implemented prototypes of the program decomposition, the points-to and the aliasing analysis algorithms; empirical results on a set of C programs are given. We have also experimented with combining two or more aliasing/points-to analyses for a same program by using the program decomposition; we present empirical results of our preliminary experiment.

Acknowledgements

I would like to thank my advisor Barbara Ryder for her support over the years. I thank Laurie Hendren, Bill Landi, Tom Marlowe, and Phil Stocks for their comments and suggestions for my thesis work. I also want to thank the PROLANGS group for providing an excellent research environment. Finally, I thank my wife Yuyun for her encouragement and patience.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

When two names refer to the same location, we say they are *aliased* and the two names constitute an *alias*. In languages such as FORTRAN, aliases can be created by call-by-reference parameter passing at procedure calls. For example, with the call $proc(a)$ for procedure $proc(f)$, where a is the actual parameter and f is the formal parameter, f and a refer to the same location during the call. In languages with general-purpose pointers, such as C, aliases can be created by assignments to pointers. For example, with the assignment $p = \&a$, where p and a are variables, p points to a afterwards, in other words, $*p$ and a refer to the same location and $(*p,a)$ is an alias. In languages such as C^{++}, aliases can be created by both call-by-reference and pointers. An alias is a *may* alias if it may hold at a program point along some execution path; an alias is a *must* alias if it holds at a program point along all execution paths to the point.

Compile-time alias information is crucial to other compile-time analyses (such as modification side effect analysis, semantic change analysis), compile-time optimizations (such as code motion, register allocation), and software tools for testing, debugging, program integration. Without the alias information, conservative assumptions have to be made in these analyses, optimizations and tools.

The call-by-reference may alias problem has been well studied [Ban78, Coo85]. There has been much research on the pointer may alias problem as well [Wei80, CR82, Cou86, LH88, Coo89, LR92, CBC93, AL94b, Deu94, AL95, NPD87, SFRW90, EGH94, And94, BCCH95, Ruf95, WL95, Ste95, Ste96a, Ste96b, SH97b]; some of the work calculate points-to information, from which may aliases can be derived. In this thesis, we will be focusing on the may alias problem for pointers in C-like languages.

The work in this thesis addresses several unsolved research problems. The first problem is the conflict between precision and scalability. Some analyses are fast and can handle programs of large size, but are quite approximate sometimes; others are quite precise, but are too costly for large programs in terms of space or time. The second problem is that most of the extant analyses do not handle features such as unions and type casting. This may affect scalability as large programs tend to use unions or type casting. The third problem is that most of the extant analyses treat different kinds of pointers (function pointers, finite-level pointers, pointers for dynamically allocated data structures, etc.) in exactly the same way. This may not be such a good idea because no single approach may be effective for different kinds of pointers in terms of cost and precision. The exceptions are the work on shape analysis [JM81, JM82, HN89, CWZ90, HN90, Ghi95, GH96, SRW96], where they only focus on pointers to dynamic data structures, and the work done at McGill University [EGH94, Ghi95, GH96], where they first proposed different analyses for pointers to the stack, pointers to the heap and pointers to dynamically allocated arrays. In this thesis, we provide some solutions to the aforementioned problems.

A Points-to Analysis Algorithm Inspired by Andersen's work [And94], we developed a points-to analysis algorithm that handles unions and type casting. Instead of using constraints and constraint solving techniques, as Andersen did, we formulate the analysis as a data flow analysis. In the analysis, assignments are treated asymmetrically, that is, given an assignment $lhs = rhs$, lhs points to whatever rhs points to, but not vice versa. Declared unions are considered by calculating their *common initial sequences* [KR88] and treating names in the common initial sequences as aliases. Type casting is treated as a way of creating unions at run-time, that is, by casting, a location can be accessed as a type different from its declared type and thus is effectively a union; they are handled similarly as the declared unions. In Chapter 4, we present the points-to analysis algorithm and compare it empirically with the Landi/Ryder aliasing algorithm [LR92]. Our empirical results show the points-to analysis is fast and almost as precise as the Landi/Ryder algorithm for some test programs.

Program decomposition We developed an algorithm that calculates an equivalence relation on the names in a program. Each equivalence class of the relation has a set of assignments associated with it, which use names in the class. The equivalence relation can be depicted as a graph, where nodes represent equivalence classes and directed edges represent a relation between names in these classes; for example, there is an edge from the class for the name p to the class for the name $*p$. The graph can be decomposed into weakly connected components. Using the weakly connected components, the equivalence relation partitions the assignments in the program into independent sets with respect to their aliasing effects; each set of assignments induces a program segment including the assignments and other control statements, which can be analyzed for pointer aliasing separately. Within a weakly connected component, the equivalence relation makes explicit the dependences among sets of assignments for aliasing analysis; this information is important, for instance, for finding aliases for function pointers and constructing call graphs without analyzing the whole program.

This idea of program decomposition was motivated by the work by Altucher and Landi at Siemens Corporate Research that separates the analysis of function pointers from other pointers for call graph construction [AL94a]. In Chapter 5, we present the program decomposition algorithm for programs without indirect calls, unions or casting; in Chapter 6, we extend it to handle programs with indirect calls, unions and casting by using the points-to analysis on selected parts of programs to resolve indirect calls and type casting.

An Aliasing Analysis Algorithm From the equivalence relation calculated for program decomposition, aliasing information can be derived. This analysis treats assignments symmetrically, that is, if there is an assignment $lhs = rhs$ in a program, the analysis will make $*lhs$ aliased to whatever $*rhs$ is aliased to and vice versa, thus effectively assume there is also an assignment $rhs = lhs$. Because of this, the analysis may be approximate sometimes. We have empirical results that show this analysis is more approximate than the points-to analysis algorithm, but it may be good enough for some applications.

Combined Analysis Experiment Using results of the program decomposition, we apply two aliasing analyses to different program segments to explore the tradeoff of precision and efficiency. In Chapter 7, we explain the experiments conducted and show the empirical results.

The rest of the thesis is organized as follows. Chapter 2 explains some of the terminology used throughout this thesis. Chapter 3 is an overview of the research work related to this thesis. Chapter 4 through Chapter 7 present the work explained above. Chapter 8 gives conclusions and future work. The two appendices provide proofs for the aliasing analysis algorithm in Chapter 5: Appendix A proves that a graph representation captures the alias relation defined. Appendix B proves the alias relation is a safe estimate of the run-time alias information.

The main contributions of this thesis are as follows:

- a program decomposition technique. We first propose the idea of program decomposition for pointer aliasing analysis. There are many potential applications for the decomposition; we have explored some of them in this thesis, such as partial aliasing analysis for function pointers and combined analysis.
- a uniform way of handling union and casting in aliasing analysis. Our approach is consistent with the C standard about unions. We are the first to consider type casting as a way of creating run-time unions.
- a points-to analysis algorithm that explicitly handles unions and type casting.
- an aliasing analysis algorithm. The version of the algorithm for programs without indirect calls, unions, or type casting is similar to Steensgaard's work [Ste95, Ste96b], but was discovered independently. The version for programs with indirect calls, unions, and type castings is different from Steensgaard's algorithms in the way that unions and casting are handled.
- empirical results for the above algorithms. We have implemented the above algorithms and studied them empirically on a set of 21 real C programs. These programs contain unions and type casting; some of them are of considerable size (>10,000 lines of code).

Chapter 2

Terminology

2.1 Object Names

For analysis purposes, programs are represented in an intermediate form. In the intermediate representation, we refer to memory locations and addresses of these locations through what we call *object names*. An object name for a memory location starts with either a variable name or a heap name, followed by a sequence of applications of structure/union member accesses (.member), array reference ([ ]), or pointer dereferences ($*$). Variable names are declared in the source program. Heap names are created explicitly for locations dynamically allocated in the program and they are of the form $heap_{id}$, where id is the statement ID of the corresponding heap allocated. Statically, some object names always refer to the same memory locations (e.g., p, $x.g$); we call them *location names*. Others can refer to different memory locations (e.g., $*p$, $p{\rightarrow}g$) depending on what locations pointers point to during program execution[1]; we call them *dereferenced names*. The address operator (&) can be applied to object names to get addresses of memory locations. Object names without & can be used as either *l-values* or *r-values* in a program, but object names with & can only be used as *r-values*. The syntax of object names and location names is given in Figure 2.1. Note that $p{\rightarrow}f$ is same as $(*p).f$ in C.

Each object name has its declared type associated with it implicitly so that we can talk about object names of pointer type or structure type. Only well-typed object names will be considered, that is, a member access can only be applied to a name of structure type with that member name and $*$ can only be applied to a name of pointer

[1] One of the purposes of pointer aliasing analysis is to determine the locations to which these names may refer.

ObjAux ::=	VarName	(variable name)
	| HeapName	(heap name)
	| ObjAux.member	(structure or union member)
	| *ObjAux	(pointer dereference)
	| ObjAux[ ]	(array element)
ObjName ::=	ObjAux	
	| &ObjAux	(address operator)

LocName ::= VarName | HeapName | LocName.member | LocName[ ]

precedence: $* > [\,] >$ .member $>$ &

associativity:	$*$	right-associative	(i.e., $**\mathrm{p} = *(*\mathrm{p})$)
	[ ] , .member	left-associative	(i.e., $\mathrm{s.f_1.f_2} = (\mathrm{s.f_1}).\mathrm{f_2}$)

Figure 2.1: Object Names and Location Names

type. The address operator can be applied to any name.

We assume each structure or union member in a program has a unique name. We call structure/union member accesses (*.member*), array reference ([ ]), and pointer dereference ($*$) *accessors*. We define three useful functions in Figure 2.2. Given an object name and an accessor, *apply* returns the object name obtained after application of the accessor; *apply** applies a sequence of accessors to an object name and returns the resulting name. The function NumOfDerefs($a_1 a_2 ... a_n$) is defined to be the number of pointer dereferences ($*$) in the sequence of accessors $a_1 a_2 ... a_n$. For example, we have:

$$apply(\&(p\rightarrow f), *) = p\rightarrow f$$
$$apply(*p, f) = (*p).f$$

$$apply^*(p, * .f) = (*p).f$$

$$\text{NumOfDerefs}(* .f) = 1$$

$apply(o, *) = *o$
$apply(o, [\,]) = o[\,]$
$apply(\&o, *) = o$
$apply(o, member) = o.member$

$apply^*(o, \epsilon) = o$
$apply^*(o, a_1 a_2 ... a_n) = apply^*(apply(o, a_1), a_2 ... a_n)$

NumOfDerefs($a_1 a_2 ... a_n$) = the number of $*$ in the sequence of accessors, $a_1 a_2 ... a_n$.

Figure 2.2: Definition of *apply*, *apply** and NumOfDerefs

2.2 Types

We will use the declared types in source programs in our analyses. The following are the possible types.

- *void* type

  This type represents an empty set of values. It is usually used as the return type of functions. The type *void* $*$ can be used for generic pointers.

- primitive types

  These types are *char*, *short int*, *long int*, *int*, *float*, *double* and *long double*. Specifiers *sign* and *unsigned* can be applied to *char* and *int*.

- pointer types

  These types include pointers to any type.

- array types

  One-dimension and multiple-dimension arrays of any type are allowed.

- structure types

  A structure consists of an ordered sequence of named members of various types.

- union types

  A union type defines several named members of various types; a location of the union type may contain any one of the members.

declaration	type representation	interpretation
$int\ a[10][10]$	$int\ [\,][\,]$	array of array of integer
$int\ *p[10]$	$int\ *[\,]$	array of pointer to integer
$int\ (*p)[10]$	$int\ [\,]*$	pointer to array of integer
$int\ **p$	$int\ **$	pointer to pointer to integer
$void\ (*fp)()$	$void\ ()*$	pointer to function returning void
$void\ func()$	$void\ ()$	function returning void

Figure 2.3: Examples of Types

Types like *int*, *float*, and *struct foo* are called *simple* types. More complicated types are represented by a simple type and a sequence of modifiers such as [ ] (array), $*$ (pointer), and () (function). Note that array size information is not important in type representation[2]. For example, $int\ *[\,]$ represents the type of arrays of pointers to integer. Some examples are given in Figure 2.3.

Type specifiers in C such as auto, register, static, signed, unsigned, const, or volatile, will be ignored when we consider whether or not two types are the same. We say two types are the *same* if they are name equivalent without any type specifiers. Because array sizes are not taken into consideration here, two array types of different sizes are considered same. Thus our notion of same types is different from the conventional one.

Now we define two functions to obtain types of union members, structure members or array elements. Given a union or structure type t and a union or structure member m, function $member\text{-}type(t\ ,\ m)$ returns the type of the union or structure member. For example,

$$member\text{-}type(\ struct\ foo\ \{\ int\ a;\ float\ b;\ char\ c;\ \}\ \ ,\ b) = float$$

To obtain types of array elements, we define the function *elem-type*. Given an array type t, function $elem\text{-}type(t)$ returns the type of array elements. For example,

$$elem\text{-}type(int\ [\,]) = int$$

[2]For access patterns (Chapter 4), we need to know array sizes.

2.3 Typed Object Names

We may associate type information with object names explicitly; we call them *typed object names*. Some examples are:

$$<u,\ int>,\ <y,\ float>, \text{ and } <s,\ struct\ foo>$$

We have three accessors: m for members of structures or unions, [ ] for array references, and $*$ for pointer dereferences. We explicitly associate types with accessors. Thus we have three kinds of typed accessors:

- $<m,\ t>$, where t is a structure or union type and m is a member of the type
- $<[\,],\ t\,[\,]>$
- $<*,\ t\ *>$

In Figure 2.4, we give the definitions of three functions: $apply_{\mathrm{T}}$, $apply^*_{\mathrm{T}}$, and NumOfTypedDerefs. Given a typed object name and a typed accessor, $apply_{\mathrm{T}}$ derives a new name. Note that the types of the typed object name and the typed accessor have to be compatible with each other; otherwise the function will return *error*. An array name of type $t\,[\,]$ can be converted into a pointer name of type $t\ *$. Thus we can apply both $<[\,],\ t\,[\,]>$ and $<*,\ t\ *>$ to a name $<o,\ t\,[\,]>$ as illustrated in Figure 2.4. $apply^*_{\mathrm{T}}$ applies a sequence of typed accessors to a typed object names and returns either the derived name or *error*. Finally the function NumOfTypedDerefs returns the number of pointer dereferences ($*$) in a sequence of typed accessors. In Figure 2.5, we show some examples of these functions.

2.4 Thru-deref MOD/REF Problems

We will use the aliasing solution of an analysis to determine the locations modified or referenced through each name containing pointer dereferences (e.g., $*p$, $p \rightarrow f$). Specifically, the two problems are:

$$apply_{\mathrm{T}}(<\&o,\ t\ *>\ ,\ <*\ ,\ t\ *>) = <o,\ t>$$

$$apply_{\mathrm{T}}(<o,\ t\ *>\ ,\ <*\ ,\ t\ *>) = <*o,\ t>$$

$$apply_{\mathrm{T}}(<o,\ t\ [\ ]>\ ,\ <[\ ],\ t\ [\ ]>) = <o[\ ],\ t>$$

$$apply_{\mathrm{T}}(<o,\ t\ [\ ]>\ ,\ <*\ ,\ t\ *>) = <o[\ ],\ t>$$

$$apply_{\mathrm{T}}(<o,\ t>\ ,\ <m,\ t>) = <o.m,\ member\text{-}type(t\ ,\ m)>,$$
where t is a structure or a union type
and m is a member of the structure or union type

$apply_{\mathrm{T}}(<o,\ t_1>\ ,\ <a,\ t_2>) = error$, if none of the above applies

$$apply_{\mathrm{T}}(error\ ,\ <a,\ t>) = error$$

$$apply^*_{\mathrm{T}}(<o,\ t>\ ,\ \epsilon) = <o,\ t>$$

$$apply^*_{\mathrm{T}}(<o,\ t>\ ,\ <a_1,\ t_1>\ <a_2,\ t_2>\ ...\ <a_n,\ t_n>) = apply^*_{\mathrm{T}}(apply_{\mathrm{T}}(<o,\ t>\ ,\ <a_1,\ t_1>)\ ,\ <a_2,\ t_2>\ ...\ <a_n,\ t_n>)$$

NumOfTypedDerefs($<a_1,\ t_1>\ <a_2,\ t_2>\ ...\ <a_n,\ t_n>$) =
the number of $*$ in the sequence of accessors $a_1 a_2 ... a_n$

Figure 2.4: Definition of $apply_{\mathrm{T}}$, $apply^*_{\mathrm{T}}$ and NumOfTypedDerefs

---

$$apply_{\mathrm{T}}(<\&s,\ struct\ foo\ *>\ ,\ <*\ ,\ struct\ foo\ *>) = <s,\ struct\ foo>$$

$$apply_{\mathrm{T}}(<*f,\ float\ *>\ ,\ <*\ ,\ float\ *>) = <**f,\ float>$$

$$apply_{\mathrm{T}}(<*p,\ struct\ foo>\ ,\ <c,\ struct\ foo>) = <p{\rightarrow}c,\ float\ *>$$

$$apply_{\mathrm{T}}(<a,\ int\ [\ ]>\ ,\ <[\ ],\ int\ [\ ]>) = <a[\ ],\ int>$$

$$apply_{\mathrm{T}}(<a,\ int\ [\ ]>\ ,\ <*\ ,\ int\ *>) = <a[\ ],\ int>$$

$$apply_{\mathrm{T}}(<\&y,\ float\ *>\ ,\ <*\ ,\ int\ *>) = error$$

$$apply^*_{\mathrm{T}}(<p,\ struct\ foo\ *>\ ,\ <*\ ,\ struct\ foo>\ <c,\ struct\ foo>) = <p{\rightarrow}c,\ float\ *>$$

NumOfTypedDerefs($<*\ ,\ struct\ foo>\ <c,\ struct\ foo>$) = 1

Figure 2.5: Examples of $apply_{\mathrm{T}}$ and $apply^*_{\mathrm{T}}$

- *Thru-deref MOD* problem: for each dereferenced name appearing as the left hand side of an assignment (e.g., $*p$=..., $**p$=...), the locations whose values may be modified by this assignment due to aliasing is determined.

- *Thru-deref REF* problem: for each dereferenced name used in a non-lhs context in the program (e.g., ...=$**p$[3], $func$(...,$*p$,...)), the locations whose values may be referenced due to aliasing is determined.

We calculate the average number of locations modified or referenced indirectly through dereferenced names for each program as a measurement of the precision of the aliasing solution obtained by the various analyses.

[3] $**p$ is counted as one reference rather than two references (one for $*p$ and one for $**p$).

Chapter 3

Related Work

Many areas of research are related to the work in this thesis. In this section, we summarize them in seven sections. Section 3.1 is concerned with program decomposition techniques for data flow analyses. Section 3.2 is about the complexity results of aliasing-related data flow problems. Both Section 3.3 and Section 3.4 talk about pointer analysis algorithms; the difference is the representation: alias relation or points-to relation. Section 3.5 emphasizes work on analyzing shapes or lifetime of data structures and Section 3.6 focuses on work on call graph construction. Finally, in Section 3.7, some analyses or optimizations that use aliasing or points-to results are briefly mentioned.

3.1 Program Decomposition for Data Flow Analyses

Any of the traditional data flow problems such as reaching definitions and live variables problems [Hec77], when solved for a program without any aliasing, can be thought of as a set of subproblems, one for each variable in the program. Each subproblem can be solved independent of others. The program can be easily decomposed for these subproblems.

Zhang, Ryder and Landi [ZRL96] presented a program decomposition technique, which partitions the assignments in a program with respect to their aliasing effects. The program decomposition allows different aliasing analyses to be applied to independent parts of the program. The idea of the technique is presented in Chapter 5 of this thesis.

Ruf [Ruf97] talked about the idea of program decomposition for data flow analyses. To obtain a decomposition, he suggested using either declared types in programs or using type inference techniques such as [Ste96a, Ste96b]. One of his target problems was points-to analysis.

3.2 Theoretic Classification of Aliasing-related Problems

Weihl [Wei80] proved that the problem of determining possible values for procedure variables (i.e., single-level function pointers in C) for programs without any other aliasing is P-space hard.

Myers [Mye81] used the NP-complete 3-satisfiability problem [Coo71] to prove that the *interprocedural* live variable problem is NP-complete, the *interprocedural* available expression problem and must-summary problem are co-NP-complete in the presence of call-by-reference aliases. These problems are intractable because there may be an exponential number of alias sets.

Using a variation of the proof by Myers, Larus [Lar91] proved that the *intraprocedural* aliasing problem in the presence of dynamic data structures is NP-complete.

Landi and Ryder [LR91] proved that both the may aliasing problem and the must aliasing problem (intraprocedural or interprocedural) are polynomial for single-level pointers (i.e., only one pointer dereference is allowed); they also showed the intraprocedural may aliasing problem is NP-hard and the intraprocedural must aliasing problem is co-NP-hard for for multiple-level pointers (i.e., more than one pointer dereferences are allowed) and recursive data structures. Their proofs of NP-hardness are also variations of Myers's.

Landi [Lan92] showed that the may aliasing problem is P-space hard for any finite number (≥ 2) of pointer dereferences and is undecidable for recursive data structures, where the number of dereferences is not known at compile time. Ramalingam [Ram94] provided a simpler proof of the same results by reducing the Post Correspondence Problem to the may aliasing problem.

3.3 Aliasing Analysis

Aliasing analysis represents aliases by pairs of names, for example, ($*p$,$*q$) means the two names, $*p$ and $*q$, refer to the same location. Most aliasing analyses determine *may* information, that is, the two names may be aliased along *some* execution path. There are few analyses which calculate *must* information, that is, the two names must be aliased along *all* execution paths. Unless noted, all aliases mentioned in this section are may aliases. Few aliasing analyses handle languages features such as type casting and unions in C. In the following discussion, unless mentioned specifically for an analysis, it is assumed that it does not consider these features.

Weihl [Wei80] probably developed the first algorithm for finding pointer aliases. His algorithm calculated aliases due to both pointers and call-by-reference. He did not take control flow information into account and considered all pointers as if they were single-level; the name $*p$ represents any location accessible through the variable p. The main idea is to calculate an *asymmetrical* relation $AFFECT$; (X,Y) in $AFFECT$ means that X is aliased to Y and to any name that is aliased to Y, but there may be some names that are aliased to X but not to Y. This models exactly the call-by-reference aliasing; if F is a formal parameter of a procedure and A is the corresponding actual parameter at a call, then (F,A) is in the $AFFECT$ relation. For a pointer assignment $p = q$, ($*p$,$*q$) is in $AFFECT$; but this is not enough with pointer assignments because there may be names that are aliased to p. So an iterative algorithm is used to calculate the $AFFECT$ relation by considering all pointer assignments and aliases for their left hand sides. The alias solution is $AFFECT^* \circ (AFFECT^*)^\top$.

Chow and Rudmik [CR82] presented an algorithm for finding aliases at program points. Similar to Weihl's work, they handled aliases caused by both pointers and call-by-references, and used the name $*p$ to represent any location accessible through the variable p. Their approach differed from Weihl's in that intraprocedural control flow information was taken into account. They defined equations relating the alias solution after the execution of a node to the solution before the node. Interprocedural propagation of aliases in their algorithm is imprecise as aliases at the exit of a procedure are propagated to all calls of the procedure and thus all calling contexts are merged together.

Coutant [Cou86] gave a staged analysis for finding program aliases. In the first stage, a set of language-specific alias rules are applied and a set of equations are set up. In the second stage, a transitive closure is performed on the equations and alias solutions are transformed into a form suitable for a global optimizer. By changing the alias rules, the algorithm can be easily adapted for different languages. The analysis handles multiple-level pointers and pointers in aggregates such as arrays and structures, but it is very conservative about aliasing effects of procedure calls; for instance, pointers returned by calls are assumed to point to any global or local whose address is taken, and thus it is really an intraprocedural analysis. To remedy this, she allowed user-provided specifications of effects of procedures. The alias rules such as the assignment rule and the delayed-action rule for $*$, are very similar to our constraints or data flow equations for points-to analysis in Chapter 4.

Larus and Hilfinger [LH88] calculated alias graphs at program points. Nodes in alias graphs represent variables or structure locations and edges represent pointers. Edges are labeled with structure member names and nodes are labeled with paths in alias graphs. Intuitively, alias graphs describe relations between variables, structures, and pointers. The labels in alias graphs are very important because they allow accesses at different statements to be compared and conflicts to be detected. Their analysis is mainly about how to maintain these labels and is quite complicated. They used the k-limiting technique [JM81] and summary nodes to deal with unboundedness of alias graphs. The interprocedural version of the analysis propagates along unrealizable paths.

Cooper [Coo89] developed an aliasing algorithm that used the last call site and a set of aliases at procedure entries to ensure that aliases are propagated interprocedurally along realizable paths. This approach may not be practical because of the potentially exponential cost in time and space.

Landi and Ryder [LR92] gave a flow-sensitive and context-sensitive algorithm for pointer aliasing analysis. The algorithm calculates program-point-specific may aliases. Interprocedurally, it uses *assumed aliases* for calling contexts. This is good enough for single-level pointers as the algorithm is proved precise for programs with single-level pointers, but is approximate with aliases involving multi-level pointers because aliases from different calls may be combined for safety. They presented empirical results of the analysis on a set of test programs and discussed various cases in which approximation must be made for the safety of the analysis. The analysis explicitly represents all possible aliases using the k-limiting scheme [JM81], which may not be efficient with programs using recursive data structures.

Choi, Burke, and Carini [CBC93] had a pointer aliasing algorithm similar to Landi and Ryder's [LR92]. There are two main differences. First, they used a compact representation of aliases, that is, aliases that can be derived from others are not explicitly represented. This may improve analysis precision in some cases and may deteriorate precision in others [EGH94]. Second, they used an alias set and a call site to represent a call context; by doing this, they were able to differentiate calling contexts better than Landi and Ryder [LR92]. However, even with this representation, a mix of aliases from different calls may still happen as the call sites in their representation are just simple call chains of length 1. Furthermore, the effect of this context representation in practice is not clear as they did not provide any empirical results of their analysis. Marlowe, etc. [MLR$^+$93] gave a good comparison of this algorithm and Landi and Ryder's [LR92].

Altucher and Landi [AL94b] implemented an algorithm for finding program aliases. Given a program, a set of names is derived first and initially each name is in an equivalence class by itself. For each pointer assignment $p = q$, the equivalence classes for $*p$ and $*q$ are unioned; furthermore, if the names, $**p$ and $**q$, exist, their equivalence classes are unioned; or if the names, $p{\rightarrow}f$ and $q{\rightarrow}f$, exist, their equivalence classes are unioned. Thus the algorithm calculates a reflexive, symmetrical, transitive and right-regular relation; any two names in an equivalence class are considered aliased. The algorithm uses the lower-level offset-size representation for names to handle features such as unions and type casting. If two names may be overlapped, they are considered aliased. They also handle function pointers and indirect calls.

Deutsch [Deu94] presented an aliasing analysis for recursive pointer data structures. He used regular expressions with integer variables to represent names (e.g., $x \rightarrow (tl \rightarrow)^i hd$, where i is 0, 1, ...). An alias consisted of two names and a constraint on the integer variables in the two names (e.g., $(x \rightarrow (tl \rightarrow)^i hd\ ,\ y \rightarrow (tl \rightarrow)^j hd)$, $j = i + 1$); this representation is very powerful; for instance, the above alias indicates i^{th} element of list x is aliased to the j^{th} element of list y. The constraints on integer variables can be expressed in a numerical lattice, which is a parameter of the analysis. The algorithm is flow-sensitive, context-sensitive, and is based on the idea of *abstract interpretation*; it can be potentially expensive. He did not have any empirical results to show the precision and cost of the analysis.

Altucher and Landi [AL95] presented an algorithm for calculating local must alias information for dynamically allocated locations. They named dynamic locations either by allocation sites in the case of calls to library allocation routines or by return nodes in the case of calls to user-defined allocation routines. Since each dynamic location name may represent more than one memory location, for example, when the allocation routine invoked is in a loop, they emphasized the *last* location created by an allocation routine and represented by a dynamic location name. The control flow graphs are broken down into segments so that it makes sense to talk about these last dynamic locations in each segment. An inexpensive and non-iterative analysis is applied to each segment to calculate the must aliases for dynamic location names. They empirically showed that these must aliases can be used to kill reaching definitions of dynamic locations and thus improve the quality of def-use information.

Neirynck, Panangaden, and Demers [NPD87] presented an algorithm for finding aliases in a higher order functional language with a few imperative constructs. They introduced the concept of variables which denote locations; two variables may refer the same location and thus are aliased through assignments or parameter passing. These variables are similar to single-level pointers. For each expression in a program, their algorithm finds a set of variables, each of which may be aliases to some variable in the expression; this set is called the *alias set* of the expression. The alias sets are then used in calculating the set of variables involved in evaluating each expression, which is called the *support set* of the expression. The support sets are used to determine if two expressions can be evaluated independently (i.e., it is used for dependence analysis). They used the approach of abstract interpretation [CC77].

3.4 Points-to Analysis

Points-to analysis uses the points-to representation, for example, (p,a) means variable p points to the location a. This is a *partial* representation of alias information as complete alias information needs to be derived from points-to pairs using transitive closure, with potential loss of precision [EGH94]. For example, if we have the points-to pairs, (p,a) and (q,a), we can say ($*p$,$*q$) is an alias, but only when the two points-to pairs hold along same execution paths. On the other hand, the points-to representation is more compact than the alias representation; in some cases, it can prevent extraneous aliases [EGH94].

Most points-to analyses determines *may* information, that is, a pointer may point to a location along *some* execution paths. There are few analyses that compute definite information, that is, a pointer must point to a location if it points to anything along *all* execution paths. Few points-to analyses handle language features such as type casting and unions in C. Unless mentioned specifically for an analysis, it is assumed that it does not consider these features.

Sagiv, Francez, Rodeh, and Wilhelm [SFRW90] presented an approach to calculating statically simple assertions that may or must hold at program points during execution. As examples, they applied the approach to three variants of the pointer equality problem in PASCAL-like languages, which is to determine if two pointers may or must point to the same location at a program point. Their analyses can handle simple boolean conditions involving pointers such as $p_1 = p_2$ or $p_1 \neq p_2$; they can also calculate may and must information simultaneously. But the analyses are intraprocedural only and they only allow pointers to simple types (i.e., single-level pointers only, no arrays, no records as in PASCAL).

Emami, Ghiya, and Hendren [EGH94] presented a flow-sensitive, context-sensitive algorithm for points-to analysis. The algorithm is mainly for points-to relations between stack locations as there is only one abstract location for all heap locations, to which all heap-directed pointers may point. For more calling context information, they use the invocation graph, which is the result of unfolding the call graph. In their analysis;

this unfolding is theoretically exponential in cost, but seems to be feasible in practice. The analysis computes both *possible* (i.e., may) and *definite* (i.e., must) points-to pairs, where the definite information can be used to improve the kill effect of statements. They handle function pointers naturally in the points-to analysis and construct the invocation graph incrementally.

Andersen [And94] developed a flow-insensitive and context-sensitive points-to analysis algorithm, which seems to be the only one of its kind. He first provided a specification of the points-to analysis as a non-standard type inference system, where, given a points-to solution, each expression is associated with a set of abstract locations it may denote. He then formulated the specification as a constraint system, in which each abstract location is assigned a type variable and constraints on these type variables are derived from the program. Finally, he solved the constraint system and instantiated the type variable for a location with a set of abstract locations, to which the location points. To identifying different contexts for procedures, he unfolded call graphs to get static call graphs, which is similar to the invocation graphs [EGH94]. If a procedure has n call contexts based on static call graphs, each of its formal parameters, location variables, allocation sites, etc., has n variants, one for each call context; each constraint derived from programs is over vectors of variants. His analysis considered the direct effects of type casting, but not the indirect ones such as locations being accessed inconsistently. No empirical results of the analysis were given.

Burke, Carini, Choi, and Hind [BCCH95] gave a flow-insensitive, context-insensitive algorithm for finding points-to information. One difference between this algorithm and others is that they employed two kinds of precomputed kill information to improve the precision of the flow-insensitive analysis. One kind summarizes aliases that will killed along all paths from the entry of a procedure to a call site, which may reduce number of aliases propagated to the procedure being called. Another kind summarizes aliases that will be killed along all paths from a call site to the exit of a procedure, which may reduce number of aliases true at the exit of the procedure. Nevertheless, the effectiveness of the kill information in practice is unknown as no empirical results were given. Their algorithm also does incremental analysis for function pointers and constructs call graphs incrementally.

Ruf [Ruf95] presented an empirical comparison of one context-sensitive and one context-insensitive points-to analyses. Both analyses are flow-sensitive, but one propagates along realizable paths only, and another propagates along all paths in a supergraph consisting of the control flow graphs of all procedures. For a set of test programs, he found the context-sensitive algorithm yields fewer points-to pairs than the context-insensitive one. For the Thru-deref MOD/REF analyses using the points-to results, the two analyses generate same precision, that is, the extra points-to pairs created by the context-insensitive analysis do not affect the Thru-deref MOD/REF analyses. Ruf offered some explanations in the paper such as the characteristics of the test programs, the simplicity of the Thru-deref MOD/REF problems, etc. It is not clear if this result about context-sensitivity holds in general.

Wilson and Lam [WL95] presented a flow-sensitive, context-sensitive algorithm for points-to analysis of C programs. Their idea is to use partial transfer functions (PTFs) to summarize the points-to effects of procedures and use them for context-sensitive analysis. They abstracted globals, heap names, locals of other procedures as extended parameters of a procedure, and used these in PTFs so that the number of PTFs can be reduced and chances for PTFs reuse can be increased. They represented locations by offsets and sizes; the analysis is conservative, for instance, any memory location can hold a pointer. Since explicit offsets and sizes are used, they can handle features such as type casting and unions. The analysis is exponential in the worst case where an exponential number of PTFs is required.

Steensgaard [Ste95, Ste96a, Ste96b] applied type inference techniques to points-to analysis of C programs. The idea is to associate with each variable a non-standard type, which describes the location and the possible run-time contents of the location if it is a pointer (only pointer values are of interest as contents of locations). The typing rules specify when two variables should have the same type (equality) or when one should have a more general type than another (inequality). If two variables have the same type, then the contents of their locations will have the same type, that is, they will point to the same set of locations if they are pointers. Initially, each variable in a program is assigned a type variable; a type inference algorithm will enforce the constraints derived from the program on these types and obtain the types for all variables. His type inference algorithm is monomorphic, that is, function calls are considered in context-insensitive manner. The result is a mapping from variables to the non-standard types, which describe the points-to relationship.

Steensgaard's first algorithm [Ste95] treats each assignment in a program as symmetric and only uses type equality constraints. The next algorithm [Ste96b] treats an assignment as asymmetric when the right hand side of the assignment is not of pointer types and thus uses both equality and inequality constraints. Neither algorithm handles structures or unions. In his latest extension to deal with structures and unions [Ste96a], locations are described by the following types:

- *blank* for all initial locations
- *simple* for locations accessed as one unit
- *struct* for locations accessed as structures
- *object* for locations accessed as unions

These types form a simple lattice with the following partial ordering: *blank* $\sqsubseteq$ *simple* $\sqsubseteq$ *object*, *blank* $\sqsubseteq$ *struct* $\sqsubseteq$ *object*. Initially, each location has type *blank*. When the type inference algorithm forces one location of type t_1 and another location of type t_2 to have the same type, the type above t_1 and t_2 in the lattice is the common type. For example, a *simple* and a *struct* become a *object*; two different *struct* become a *object*. Because pointer assignments are treated symmetrically, with type casting, it is possible to have many inconsistent uses of locations and thus to have many *objects*.

Tonella, Antoniol, Fiutem, and Merlo [TAFM97] extended Steensgaard's approach to deal with features of C++ language such as classes, polymorphism and virtual functions. It seems that they do not handle unions and type casting. They use the points-to results to slice C++ programs in a program understanding environment.

Shapiro and Horwitz [SH97b] presented two flow-insensitive, context-insensitive algorithms for points-to analysis. The first one is similar to Steensgaard's [Ste96b]. The difference is that variables in a program are partitioned into categories and nodes for two variables are unioned if they are in the same category. It is not known how to partition variables or how many categories are appropriate. The second one uses the first one for multiple runs with multiple categories; the points-to results of the multiple runs are then intersected to get a more precise points-to result. Neither algorithm distinguishes members of structures or unions; an assignment to any member is treated as an assignment to the structure or union.

3.5 Shape Analysis and Lifetime Analysis

Both shape analysis and lifetime analysis are concerned about dynamically allocated locations in heaps. The former seeks to find the shapes of dynamic data structures in programs; this information is useful for applications such as dependence analysis and parallelization. The latter tries to determine the lifetime of dynamic data structures; this information is useful for heap optimizations such as compile-time garbage collection, stack allocations instead of heap allocations. Usually, neither analysis considers aliasing among stack locations.

Jones and Muchnick [JM81] first worked on the shape analysis problem for first order LISP-like languages. Their analysis, which is intraprocedural only, calculates for each program point a set of graphs representing stores on one or more execution paths up to the program point. Nodes in the graphs represent dynamic locations; directed edges in the graphs represent pointers and are labeled with structure member names. A technique called *k-limiting* is used to make both the graphs and the number of them finite. Nodes beyond labeled paths of length k are partitioned into undirected connected components and all nodes in each component are collapsed into a *unknown* node. Unknown nodes may have outgoing edges and these edges are not labeled. The k-limiting notion is also used by other researchers [LR92, LH88]. Later, Jones and Muchnick [JM82] extended this work to interprocedural analysis and used the formal approach of abstract interpretation.

Deutsch [Deu90] gave an analysis that determine if a location can be accessed at a program point for higher order functional languages. Although the accessibility information can be used to derive other useful information such as liveness or isolation of data structures, it is really coarse-grain for other applications. He allowed closures and continuations, and used a very formal approach of abstract interpretation, but did not allow assignments (side effect) in the languages.

Ruggieri and Murtagh [RM88] had an algorithm that collected, at compile time, lifetime information of dynamic locations. They do not distinguish different locations allocated by the same statement and assume they will have the same lifetime. The algorithm has two passes, one for intraprocedural and another for interprocedural analysis. During the intraprocedural pass, formal parameters and returned values of procedures are place-holders, but very conservative alias assumptions are made about formal parameters and their subcomponents if they are of structure types; if two names have the same declared types, they are assumed to be aliased. During the interprocedural pass, summary information about actual parameters at calls and returned values of procedures collected after the intraprocedural pass is used to replace those place-holders. The interprocedural pass tracks the propagation of dynamic locations through procedures on call graphs. The compile-time lifetime information can be used for heap allocation optimization, for example, each allocation site can associated with a procedure such that all locations allocated at the site can be safely associated with an activation record of the procedure.

Chase, Wegman and Zadeck [CWZ90] associated one storage shape graph with each program point in their shape analysis. Their approach is different from Jones and Muchnick's [JM81, JM82] in that the k-limiting technique is not used. Nodes in storage shape graphs are merged only if they are allocated by the same statement and if other conditions are true, for example, they are not pointed to by a variable. Strong updates are used to improve analysis precision in certain circumstances. For implementation, they employ an efficient data structures for storage shape graphs.

Hendren [HN89, HN90] gave an analysis that focused on relationships among pointers. Specifically, she tried to answer questions such as whether or not there is a path from a node pointed by one pointer to a node pointed by another pointer, and if there is such a path, how to describe the path. She used *path matrices* to represent these relationships; each entry of a path matrix describes a relationship between two pointers. One path matrix is associated with each program point. The analysis is flow-sensitive and is able to detect list or tree structures, but gives up on cyclic data structures. A unique feature of the analysis is that may and must information is maintained simultaneously in path matrices.

Ghiya [Ghi95] proposed three analyses that focused on heap-directed pointers. Three kinds of matrices are calculated by the analyses, one by each analysis. All matrices are variations of path matrices [HN89, HN90], but only contain boolean values; they are designed to collect coarse path information efficiently. These matrices are:

- connection matrix

  This matrix provides information on whether or not two pointers point to the same data structure.

- direction matrix

  This matrix provides information on whether or not there is a path from a node pointed by one pointer to a node pointed by another pointer; this is mainly used for finding list or tree data structures.

- interference matrix

  This matrix provides information on whether or not two pointers can access the same location; this is mainly used for finding dag-like data structures.

All analyses are flow-sensitive and context-sensitive, similar to [EGH94].

Ghiya and Hendren [GH96] presented a flow-sensitive, context-sensitive algorithm of shape analysis for C programs. The analysis is performed after a points-to analysis [EGH94] and focuses on pointers found to point to heap by the points-to analysis. They use storeless abstractions (i.e., relationships between names) such as direction matrix and interface matrix, which are similar to path matrix [HN89, HN90]. These are very simple yet practical abstractions, but may not be powerful enough for programs with many structure changes. The invocation graph [EGH94] is used to make the analysis context-sensitive, but also makes the analysis costly that it may not scale to large programs.

Sagiv, Reps and Wilhelm [SRW96] presented an intraprocedural, flow-sensitive algorithm of shape analysis for LISP-like languages. They used the shape graphs to represent possible shapes at program points (i.e., store-based abstraction). Each node in a shape graph is named by the set of variables that must *all* point to the run-time location represented by the node along *some* execution paths. This naming scheme allows strong updates in their analysis; it also means some kinds of *definite* information is been calculated and thus may make the analysis costly. Each node also has information of whether it is the target of more than one pointer; this information makes it possible to *unsummarize* summary nodes in some cases. Their analysis gains better precision with the naming scheme and the sharing information, but it may be expensive as the number of nodes in a shape graph could be exponential in the number of variables in a program. The analysis is mainly intraprocedural; its cost may be a problem when extending to interprocedural analysis.

Region Inference [AFL95, BTV96, TT94] statically determines the lifetime of all run-time values including basic values, records, and function closures. Values of similar lifetime are grouped together to form a region, which are allocated and deallocated in a stack-like fashion. As the result of region inference, a source program can be translated into a target program with region annotations, which indicate the stack-like manner of allocations and deallocations of regions. The technique is aiming at achieving efficient use of memory resources and avoiding garbage collection conventionally used in implementations of applicative languages. The first region inference algorithm was given by Tofte and Taplin[TT94], which was later improved by Aiken, etc. [AFL95] and Birkedal, etc. [BTV96].

3.6 Call Graph Construction

Depending on the programming languages being considered, work in this research area can be classified into four groups. The first concerns *call graph construction* in the presence of procedure parameters as in FORTRAN77. The second considers function pointers and calls through them as in C. The other two deal with dynamically bound invocations as in object-oriented and first-class functions as in higher-order functional languages.

Call Graph Construction for FORTRAN Procedure parameters in FORTRAN are like single-level function pointers in C. Assignments for procedure parameters are not allowed; only a procedure name or another procedure parameter can be used as an actual argument for a procedure parameter. A procedure call through a procedure parameter may invoke any of procedures passed to that parameter.

Ryder [Ryd79] presented the first algorithm for call graph construction for FORTRAN programs with calls through procedure parameters. She used *procedure vectors* to represent possible values that procedure parameters of a procedure can take along an execution path; a procedure vector for a procedure consists of one procedure name for each procedure parameter of the procedure. The algorithm propagates procedure vectors on an incomplete call graph and extends the call graph, when calls through procedure parameters are resolved. Her algorithm is precise. Callahan, Carle, Hall, and Kennedy [CCHK90] extended Ryder's algorithm to handle recursion. Hall and Kennedy [HK93] simplified Ryder's algorithm by propagating *single pairs*, each made of a procedure parameter and a procedure name, rather than procedure vectors, on incomplete call graphs. Their algorithm is more efficient, but is not as precise. Lakhotia [Lak93] gave a more general algorithm which allows assignments for what he called *procedure variables*. The algorithm propagates single pairs on program dependence graphs. His algorithm treats procedure calls through these variables as arbitrary branches and thus is approximate.

Aliasing Analysis for Call Graph Construction Function pointers as in C can only point to functions or procedures. Like any other pointers, they can be in aggregates such as arrays and structures, can be in assignments, can be passed as arguments to or returned as values by procedures; that is, they are just a special kind of pointers. Therefore, most aliasing or points-to analysis algorithms can be adapted for function pointers to construct call graphs incrementally.

Weihl's aliasing algorithm [Wei80] can handle procedure variables and build call graphs incrementally as values for procedure variables are determined. Ghiya [Ghi92] extended Hendren's algorithm [HN89, HN90] to deal with function pointers. Emami, Ghiya, and Hendren [EGH94] handled function pointers in their points-to analysis. Altucher and Landi [AL94a] had an algorithm for constructing call graphs for programs with indirect calls through function pointers. The difference between their work and others is that they safely identified pointers related to function pointers and applied Landi and Ryder aliasing analysis [LR92] to those pointers only; others simply applied analyses to all pointers. The other analysis algorithms [BCCH95, Ste95, WL95] also handle function pointers and calls through them as a part of the analyses.

The idea of program decomposition for pointer aliasing analysis in this thesis was motivated by Altucher and Landi's work [AL94a].

Resolving Dynamically Bound Invocations Object-oriented languages provide the feature of dynamic bindings of method invocations at run time. In a statically typed language (e.g., C++), it is accomplished by the use of *virtual methods*. When a virtual method is invoked on an object, the actual method called is determined from the type of the object at run-time. In a dynamically typed language (e.g., SELF), all method invocations are dispatched at run time according to the types of receiving objects. To reduce the overhead associated with dynamic dispatches of method invocations, static analysis has been used to resolve method invocations at compile time. This problem is tantamount to type determination for objects in programs.

Chambers and Ungar [CU90] proposed an iterative type analysis for SELF. Pande and Ryder[PR94] developed a static type determination algorithm for C++ in the presence of pointers. Other analyses have also been given in the literature [APS93, BS96, DGC98, GDDC97, PC94]

First-class Functions in Higher-order Functional Languages In functional languages such as Scheme, functions are first class citizens. They are just data that can be created at run-time, stored into aggregates, passed as arguments to procedures, or returned as values from functions, etc.

Shiver [Shi88] proposed the first algorithm for resolving function invocations through variables in Scheme; he called the problem *control flow analysis*. He considered the problem as determining possible values for variables. At a program point, a set of lambda expressions (i.e., procedures) is associated with each interesting variable (i.e., a function pointer), which can take any of these lambda expressions as its value; these sets are propagated on an incomplete control flow graph until a fixed point is reached. Since functions can be created at run time in Scheme, the algorithm collapses all functions created at a program point to one static function. In essence, the algorithm is similar to these of Hall and Kennedy [HK93] and Lakhotia [Lak93]. It is not precise and may be very approximate because of the ubiquitous uses of first class functions in Scheme.

Other works in this area include [Ash96, Ash97, HM97, JW95, NN97, SZ94].

3.7 Applications Using Aliasing or Points-to Information

There are many analyses or optimizations using results of aliasing or points-to analyses. In this section, we briefly mention some of these works.

Chow and Rudmik [CR82] used the results of their aliasing analysis to compute direct MOD/REF information for statements, summary MOD/REF information for procedures, and interprocedural MOD information for call sites and procedures. In their calculation, the MOD information at one call site may include side effects caused by aliases introduced by other calls.

Landi, Ryder and Zhang [LRZ93] developed an algorithm for the modification side effect analysis for C using Landi and Ryder's aliasing algorithm [LR92]. The algorithm can also be used with other aliasing analyses [SRLZ98]. Empirical results of modification or references through pointer indirections were also reported in [EGH94, ZRL96].

Shapiro and Horwitz [SH97a] presented some empirical results of GMOD analysis, live variable analysis, and program slicing for C programs.

Pande, Ryder, and Landi [PRL91] worked on reaching definition and Def-Use problems for C. Altucher and Landi [AL95] improved the analyses using localized must-alias information.

Ghiya and Hendren [GH98] used points-to and heap analyses for optimizations such as loop-invariant removal, location-invariant removal, global common subexpression elimination, program understanding and debugging.

Debray, Muth, and Weippert [DMW98] presented a dependence analysis for low level representation of memory and used it in moving loop-invariant load operations out of loops.

Hendren and Nicolau [HN89, HN90] used a simple shape analysis for parallelizing programs with recursive data structures. There have been a lot of works in heap analysis for conflict analysis, dependence test, and parallelization [Gua88, Har89, HHN92a, HPR89, HHN92b, LH88, RRH92].

Other applications of aliasing or points-to information include register promotion [CL97] and data prefetching [LM96].

Chapter 4

Flow-insensitive Points-to Analysis with Unions and Type Casting

In this chapter, we present a flow-insensitive points-to analysis algorithm, which we also call the *PT analysis*. This analysis will be used by the program decomposition algorithm to resolve indirect calls through function pointers and to deal with type casting. In Section 4.1, we describe the inputs for the analysis. For Section 4.2 and 4.3, we assume we are dealing with programs without union types and type casting except for calls to dynamic allocation routines such as *malloc*(). In Section 4.2, we present the points-to analysis as a constraint-solving technique; in Section 4.3, we present it as a data flow analysis technique. We will discuss our approach to dealing with unions and type casting in the points-to analysis in Section 4.4. In Section 4.5, we present the points-to analysis in the presence of unions and type casting. Finally, we show empirical results of the points-to analysis in Section 4.6. In this chapter, we will refer to the *flow-insensitive* points-to analysis simply as the points-to analysis.

4.1 Inputs for Points-to Analysis

Given a program, the two input sets to the points-to analysis are:

- **O**: a set of object names

  These object names come from declarations and statements in the program. A subset of these names is a set of location names, which are object names representing locations (e.g., i, $s.f$, $a[\,]$). We assume **O** includes all location names declared in the program and all object names syntactically appearing in the program and their prefixes.

The following are four kinds of object names in **O**, where loc is a location name and $deref$ is an object name containing at least one dereference operator ($*$) (e.g., $*p$, $p{\rightarrow}f$).

- $\&loc$ (e.g., $\&p$)
- $\&(deref)$ (e.g., $\&(p{\rightarrow}f)$)
- loc (e.g., p)
- $deref$ (e.g., $*p$)

• **A**: a set of pointer assignments

These assignments are for object names of pointer types. They come from assignments in the program; formal-actual bindings in C can be considered as assignments. We assume assignments for structures or unions have been turned into assignments for structure or union members. For pointer aliasing analysis purposes, the program can be considered to consist of a sequence of assignments for names of pointer types.

Each pointer assignment is of the form $lhs = rhs$, where lhs is an object name of the form loc or $deref$, rhs is any object name, and both are of pointer types. Two or more instances of an assignment with same lhs and rhs in the program will be represented by just one instance in **A**. We assume that the object names used as left and right hand sides of pointer assignments are in **O**.

For each location name loc of pointer types in **O**, the points-to analysis calculates a set of locations ($pts(loc)$) that may be pointed to by loc. For each dereferenced name o in **O** of pointer type (e.g., $*p$), the points-to analysis calculates a set of locations ($pts(o)$) that may be pointed to by any location aliased to o. For example, if a variable i is aliased to $*p$, then $pts(*p)$ will contain the locations that i points to.

There are two reasons for keeping points-to sets for names like $*p$. The first reason is the convenience of the points-to analysis. For example, for an assignment $q = *p$, we can simply say $pts(q)$ should contain $pts(*p)$. If not available right away, $pts(*p)$ can be derived from the set $pts(p)$ and the points-to sets of the location names in $pts(p)$. The second reason is the necessity for deriving aliasing information for object names with more than one dereference. For instance, we might need alias information for names like $**p$; with the set $pts(*p)$, we can say $**p$ is aliased to i if i is in $pts(*p)$. In summary, keeping $pts(*p)$ is a trade-off of space for efficiency[1].

An important idea of our points-to analysis is that aliasing information between location names (e.g., i) and dereferenced names (e.g., $*p$) is derived from the points-to sets during the analysis and is used in the analysis to derive more points-to information. For example, if p is known to point to i by the analysis (i.e., $i \in pts(p)$), then $<*p\ , i>$ is an alias, which can then be used in the analysis at an assignment for $*p$, for instance, $*p = \&j$, to derive that i points to j (i.e., $j \in pts(i)$) . In the points-to analysis, we are not interested in aliases such as $< *l\ \ , *m >$.

The rules for deriving alias information from points-to information are as follows.

- If $l_1 \in pts(o)$, let o_1 be $apply(o, *\ )$ and $< o_1\ \ , l_1>$ is an alias found by the points-to analysis.

- Let $< o_1\ \ , l_1>$ be an alias found by the points-to analysis. If o_1 and l_1 are of a union type, for each union member m of the structure type, $< apply(o_1,m)$, $apply(l_1,m)>$ is an alias found by the points-to analysis.

- Let $< o_1\ \ , l_1>$ be an alias found by the points-to analysis. If o_1 and l_1 are of a structure type, for each member f of the structure type, $< apply(o_1,f)$, $apply(l_1,f)>$ is an alias found by the points-to analysis.

- Let $< o_1\ \ , l_1>$ be an alias found by the points-to analysis. If o_1 and l_1 are of an array type, $<apply(o_1,[\ ])$, $apply(l_1,[\ ])>$ is an alias found by the points-to analysis.

We will use an example in describing the points-to analysis in the next two sections; the declaration for variables, the set **O** and the set **A** are shown in Figure 4.1.

[1]If all object names in a program are normalized to have at most one dereference, then there is no need to keeping points-to sets for names like $*p$. Currently we allow names with more than one dereference.

$$int\ {**p},\ {**q},\ {*a},\ {*b},\ {*d},\ {*r},\ c,\ x,\ y;$$

$$\mathbf{O} = \{\ p,\ {*p},\ q,\ {*q},\ \&a,\ a,\ {*a},\ \&b,\ b,\ {*b},\ \&d,\ d,\ r,\ {*r},\ \&c,\ c,\ \&x,\ x,\ \&y,\ y\ \}$$

$$\mathbf{A} = \{\ p = \&a,\ p = \&b,\ {*p} = \&c,\ q = \&d,\ {*q} = {*p},\ a = \&x,\ b = \&y,\ r = {*q}\ \}$$

Figure 4.1: An Example

4.2 Points-to Analysis as Constraint-solving

In this section, we present the points-to analysis as solving a system of constraints on the points-to sets associated with object names. The constraints are derived from the pointer assignments in **A** and the aliasing information found during the points-to analysis. The constraints are dynamically changing as more aliasing information is found; in other words, the constraints are not *fixed* and more constraints may be added during the analysis. The result of the analysis is a *minimal* solution[2] for final system of constraints. The constraints for points-to analysis are as follows.

- If $\&l \in \mathbf{O}$, where l is a location name, $pts(\&l) = \{\ l\ \}$.
- Let o be a dereferenced name in **O**. If $<o\ ,\ l>$ is an alias found by the points-to analysis, where l is a location name, $pts(o) \supseteq pts(l)$; furthermore, if $\&o \in \mathbf{O}$, $pts(\&o) \supseteq \{\ l\ \}$.
- If $lhs = rhs$ is a pointer assignment in **A** and $<lhs\ ,\ l>$ is either a trivial alias[3] or an alias found by the points-to analysis, $pts(l) \supseteq pts(rhs)$.

We can also combine the constraints and formulate the system of constraints as follows.

- If $\&l \in \mathbf{O}$, where l is a location name, $pts(\&l) = \{\ l\ \}$.

[2] By minimal, we mean the set sizes of points-to sets are the smallest.

[3] That is, *lhs* is a location name and is aliased to itself.

- Let o be a dereferenced name in **O**. Assuming o is found by the points-to analysis to be aliased to any of the location names $\{ l_1, ..., l_k \}$, the constraint is:

$$pts(o) \supseteq \bigcup_{1 \leq j \leq k} pts(l_j)$$

$$pts(\&o) \supseteq \{l_1, ..., l_k\}, \quad \text{if } \&o \in \mathbf{O}$$

- Let $lhs = rhs$ be a pointer assignment in **A**. Assuming lhs is either trivially aliased to or found by the points-to analysis to be aliased to any of the locations in $\{ l_1, ..., l_n \}$, the constraints are :

$$pts(l_i) \supseteq pts(rhs), \text{ where } 1 \leq i \leq n$$

The system of constraints derived from a program can be solved by a constraint-solving algorithm that dynamically adds constraints to a given system. The constraint system can also be used in checking if a points-to solution is safe. Specifically, given a points-to solution, if all constraints are satisfied, then the points-to solution is safe; otherwise it is not safe.

For the example program, the system of constraints is given in Figure 4.2 and the solutions to the constraints, i.e., the final points-to sets associated with location names, are shown in Figure 4.3.

4.3 Points-to Analysis as Data Flow Analysis

In this section, we present the points-to analysis as solving a set of data flow equations on a graph called the *pointer value flow graph.* The graph is constructed from the set of object names **O** and the set of pointer assignments **A**. The pseudo code for the construction of the initial pointer value flow graph is given in Figure 4.4. For each object name in **O**, there is a node in the graph; for each pointer assignment $lhs = rhs$ in **A**, there is a $\longrightarrow$ edge from the node for rhs to the node for lhs; for each alias $<o, l>$ found during the points-to analysis, where o is a dereferenced name in **O**

$pts(\&a) = \{ a \}$

$pts(\&b) = \{ b \}$

$pts(\&c) = \{ c \}$

$pts(\&d) = \{ d \}$

$pts(\&x) = \{ x \}$

$pts(\&y) = \{ y \}$

for any $l \in pts(p)$, $pts(*p) \supseteq pts(l)$

for any $l \in pts(q)$, $pts(*q) \supseteq pts(l)$

$pts(p) \supseteq pts(\&a)$ $\quad (\{ p = \&a \} \subseteq \mathbf{A})$

$pts(p) \supseteq pts(\&b)$ $\quad (\{ p = \&b \} \subseteq \mathbf{A})$

for any $l \in pts(p)$, $pts(l) \supseteq pts(\&c)$ $\quad (\{ *p = \&c \} \subseteq \mathbf{A})$

$pts(q) \supseteq pts(\&d)$ $\quad (\{ q = \&d \} \subseteq \mathbf{A})$

for any $l \in pts(q)$, $pts(l) \supseteq pts(*p)$ $\quad (\{ *q = *p \} \subseteq \mathbf{A})$

$pts(a) \supseteq pts(\&x)$ $\quad (\{ a = \&x \} \subseteq \mathbf{A})$

$pts(b) \supseteq pts(\&y)$ $\quad (\{ b = \&y \} \subseteq \mathbf{A})$

$pts(r) \supseteq pts(*q)$ $\quad (\{ r = *q \} \subseteq \mathbf{A})$

Figure 4.2: The System of Constraints for the Example Program

$pts(\&a) = \{\ a\ \}$ $\qquad$ $pts(p) = \{\ a,\ b\ \}$

$pts(\&b) = \{\ b\ \}$ $\qquad$ $pts(q) = \{\ d\ \}$

$pts(\&c) = \{\ c\ \}$ $\qquad$ $pts(a) = \{\ x,\ c\ \}$

$pts(\&d) = \{\ d\ \}$ $\qquad$ $pts(b) = \{\ y,\ c\ \}$

$pts(\&x) = \{\ x\ \}$ $\qquad$ $pts(d) = \{\ x,\ y,\ c\ \}$

$pts(\&y) = \{\ y\ \}$ $\qquad$ $pts(r) = \{\ x,\ y,\ c\ \}$

$pts(*p) = \{\ x,\ y,\ c\ \}$ $\qquad$ $pts(c) = pts(x) = pts(y) = \{\ \}$

$pts(*q) = \{\ x,\ y,\ c\ \}$ $\qquad$ $pts(*a) = pts(*b) = pts(*r) = \{\ \}$

Figure 4.3: Solutions of the Constraints for the Example Program

```
graph-construction-wo-union-wo-casting()
{
    for each object name o in O
        create a node for o

    for each pointer assignment lhs = rhs in A
        if (there is no edge from the node for rhs to the node for lhs)
            create a ⟶ edge from the node for rhs to the node for lhs
}
```

Figure 4.4: graph-construction-wo-union-wo-casting()

and l is a location name in **O**, there is a $\longleftrightarrow$ edge from the node for o to the node for l. The initial pointer value flow graph does not have the $\longleftrightarrow$ edges, which are added during the points-to analysis. For the example program, the initial pointer value graph is shown in Figure 4.5.

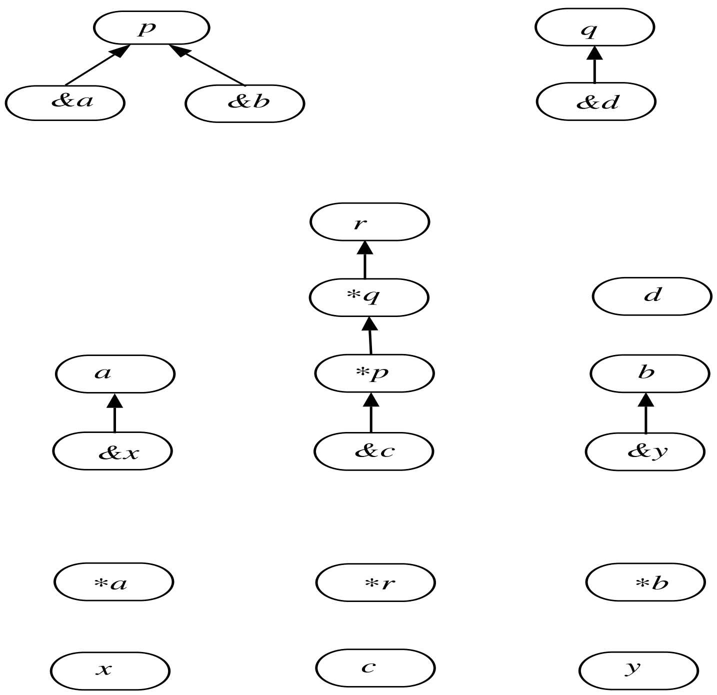


Figure 4.5: The Initial Pointer Value Flow Graph

We calculate a set of location names for each node (i.e., for each object name in **O**) in the pointer value flow graph. Since each node in the graph is for one object name, we will use the notation $pts(o)$ for the set associated with the node for the object name o in **O**. The data flow equations defined on the pointer value flow graph, are shown in Figure 4.6, where loc is a location name, $deref$ is a dereferenced name, obj_1 and obj_2 are any object names. The equations depend on the $\longleftrightarrow$ edges added during the analysis. The function ρ in the equations derives alias information from points-to pairs and adds $\longleftrightarrow$ edges to the graph if necessary; given a points-to set, the function will simply return the set after considering each points-to pair in the set. Intuitively,

$$pts(\ \&loc\ ) = \{\ loc\ \} \qquad (1)$$

$$pts(\ \&deref\ ) = \bigcup_{deref \longleftrightarrow loc} \{\ loc\ \} \qquad (2)$$

$$pts(\ loc\ ) = \rho \left(\begin{array}{ll} \displaystyle\bigcup_{obj_1 \longrightarrow loc} pts(\ obj_1\ ) \ \cup & (3a) \\ \displaystyle\bigcup_{loc \longleftrightarrow deref} \ \bigcup_{obj_2 \longrightarrow deref} pts(\ obj_2\ ) & (3b) \end{array} \right)$$

$$pts(\ deref\ ) = \rho \left(\bigcup_{deref \longleftrightarrow loc} pts(\ loc\ ) \right) \qquad (4)$$

Figure 4.6: Data Flow Equations for the Points-to Analysis without Unions and Casting

Equation (1) says $\&loc$ [4] is a pointer that points to one location loc and Equation (2) dictates $\&deref$ represents a pointer that points to any location that is aliased to $deref$. The equation for $pts(loc)$ is the most complicated. Let us explain each term on the right hand side of the equation. Term (3a) considers all pointer assignments with loc as left hand side; if $loc = obj_1$ is a pointer assignment, that is, there is a $\longrightarrow$ edge from obj_1 to loc in the graph, $pts(loc)$ contains $pts(obj_1)$, which is implied by the semantics of the assignment. Term (3b) considers all the aliases for loc found by the analysis; if loc is aliased to $deref$ (i.e., there is a $\longleftrightarrow$ edge between loc and $deref$) and $deref = obj_2$ is a pointer assignment (i.e., there is a $\longrightarrow$ edge from obj_2 to $deref$), then $pts(loc)$ contains $pts(obj_2)$, which is implied by the semantics of the assignment and the alias. Lastly, the points-to set for a dereferenced object name is the union of the points-to sets of locations aliased to the name (Equation (4)). Note that we do not have $pts(loc)$ contain $pts(deref)$ if $<deref\ , loc>$ is an alias found by the points-to analysis. Basically this is for better precision; $deref$ can be aliased to more than one location and thus $pts(deref)$ contains the points-to set of any of these locations by Equation

[4] A location name of an array type (e.g., a) is considered a pointer to the first array element (e.g., $\&(a[])$).

(4), but these locations are not aliased to each other.

The data flow equations can be solved by a worklist algorithm that adds $\longleftrightarrow$ edges to the initial pointer value flow graph and updates the equations. The pseudo code for such an algorithm is provided in Figure 4.7 and 4.8. In the worklist algorithm, there are two phases. In the first phase, the points-to set for each name in **O** is initialized to the empty set and the worklist is initialized. The second phase is an iterative one. In each iteration, one points-to pair is removed from the worklist and a location might be added to a points-to set. If a location is added, Term (3a) and (3b), and Equation (4) are affected because these equations define the relation between points-to sets. Figure 4.9 (I) shows what happens if a location l is added to the points-to set $pts(o)$ in worklist-algorithm-wo-union-wo-casting(). First, all $\longrightarrow$ from o are considered, that is, all pointer assignments with o as the right-hand-side are considered. For each such assignment $o_1 = o$, if o_1 is a location name, $pts(o_1)$ must contain l because of the pointer assignment; if o_1 is a dereferenced object name, all $\longleftrightarrow$ from o_1 are considered, that is, all locations found aliased to o_1 are considered, and for any of these locations, l_1, $pts(l_1)$ must contain l because of the assignment and the alias $<o_1 , l_1>$. Secondly, if o itself is a location name, all $\longleftrightarrow$ from o are considered, that is, all dereferenced names found aliased to o are considered, and for any of these names, o_2, $pts(o_2)$ must contain l because of the alias $<o , o_2>$. Note that all edges considered are in the *current* pointer value flow graph.

For each points-to pair (e.g., (o , l)), the routine find-an-alias-wo-union-wo-casting() (Figure 4.8) is called to deal with the new alias found $<apply(o, *) , l>$. This routine does what the function ρ is supposed to do. For the points-to analysis, we are only interested in aliases involving two names in **O**; aliases other than these will not affect the points-to analysis. If the alias involve two names in **O** and no $\longleftrightarrow$ exists between the nodes for the two names, an $\longleftrightarrow$ edge is added. Furthermore Term (3b) and Equation (4) are affected because these equations consider the $\longleftrightarrow$ edges. Figure 4.9 (II) shows the effect of a new alias $<deref , loc>$ and the new $\longleftrightarrow$ edge added between the node for $deref$ and the node for loc. First of all, if there is node for $pts(\&deref)$ in the graph, loc is added to $pts(\&deref)$ because of the alias. Secondly, all locations

```
worklist-algorithm-wo-union-wo-casting()
{
    /* initialization */
    for each object name o in O
    {
        pts(o) = { }
        if (o is &loc)
            add (&loc , loc) to the worklist   /* Equation (1) */
    }

    /* iteration */
    while (the worklist is not empty)
    {
        remove (o , l) from the worklist

        if (l is not in pts(o))
        {
            add l to pts(o)

            for each ⟶ edge from o
            {
                let the edge be from o to o1
                if (o1 is a location name)
                    add (o1 , l) to the worklist   /* Term (3a) */
                else /* o1 is a dereferenced object name. */
                    for each ⟷ edge from o1
                    {
                        let the edge be between o1 and a location name l1
                        add (l1 , l) to the worklist   /* Term (3b) */
                    }
            }

            if (o is a location name)
                for each ⟷ edge from o
                {
                    let the edge be between o and an object name o2
                    add (o2 , l) to the worklist   /* Equation (4) */
                }

            /* derive alias information from points-to information */
            if (o is neither &loc nor &deref)
                find-an-alias-wo-union-wo-casting(apply(o, * ) , l)
        }
    }
}
```

Figure 4.7: worklist-algorithm-wo-union-wo-casting()

```
find-an-alias-wo-union-wo-casting(deref , loc)
{
    if (deref ∉ O)   return   /* consider only names in O */

    if (there is already a ⟷ edge between the node for deref and the node for loc)
        return

    add a ⟷ edge between the node for deref and the node for loc

    if (&deref ∈ O)
        add (&deref , loc) to the worklist   /* Equation (2) */

    for each ⟶ edge to deref
    {
        let the edge be from obj1 to deref
        for each loc1 in pts(obj1)
            add (loc , loc1) to the worklist   /* Term (3b) */
    }

    for each loc2 in pts(loc)
        add (deref , loc2) to the worklist   /* Equation (4) */

    /* An alias involving a struct/array implies more aliases. */
    if (deref is of a structure type)
        for each member fld of the structure
            find-an-alias-wo-union-wo-casting(apply(deref,fld) , apply(loc,fld))
    else if (deref is of an array type)
            find-an-alias-wo-union-wo-casting(apply(deref,[ ]) , apply(loc,[ ]))
}
```

Figure 4.8: find-an-alias-wo-union-wo-casting()

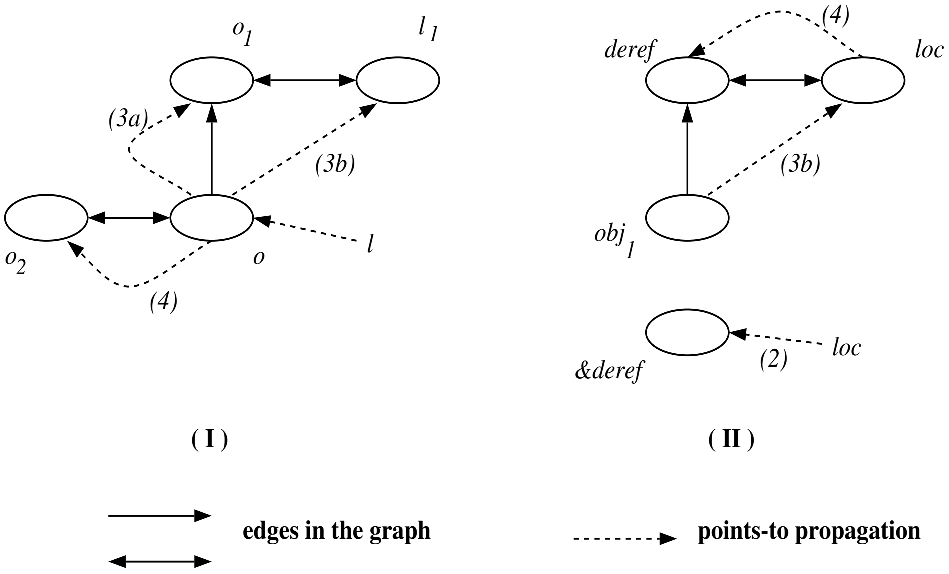


Figure 4.9: Propagation on the Pointer Value Flow Graph

```
points-to-analysis-wo-union-wo-casting()
{
    /* construction of the initial pointer value flow graph */
    graph-construction-wo-union-wo-casting()

    /* application of the worklist algorithm */
    worklist-algorithm-wo-union-wo-casting()
}
```

Figure 4.10: Points-to Analysis without Unions and Type Casting

in $pts(obj_1)$ are added to $pts(loc)$ because of the pointer assignment $deref = obj_1$ and the alias. Finally, all locations in $pts(loc)$ are added to $pts(deref)$ because of the alias. If the alias derived involves names of union types, structure types or array types, more aliases are found and the routine is called recursively.

The points-to analysis consists of two steps, the construction of the initial pointer value flow graph and the application of the worklist algorithm. The pseudo code for the analysis is provided in Figure 4.10.

In Figure 4.11, we show some intermediate pointer value flow graphs during application of the worklist algorithm to the example program in Figure 4.1. These graphs do not represent a step-by-step simulation of the algorithm; the purpose here is to explain the ideas of the algorithm through these graphs. Figure 4.11(a) is the graph where only the points-to sets for names such as $\&a$ and $\&b$ have been initialized and the points-to sets for other names are empty. The points-to pairs for names such as $\&a$ are then propagated along the $\longrightarrow$ edges in the graph to nodes for location names; we have the points-to set for p to be $\{\ a,\ b\ \}$, the points-to set for q to be $\{\ d\ \}$, etc., as shown in Figure 4.11(b). Once we find p points to a, we know that $*p$ is aliased to a. We will add a $\longleftrightarrow$ edge between the node for $*p$ and the node for a in the pointer value flow graph, as illustrated in Figure 4.11(c). Because of the edge, points-to pairs will be propagated from the node for a to the node for $*p$; points-to pairs will also be propagated from any node that has a $\longrightarrow$ edge to the node for $*p$ (e.g., the node for $\&c$) to the node for a. Thus we have the points-to sets for $*p$ and a to be $\{\ x,\ c\ \}$, as shown in Figure

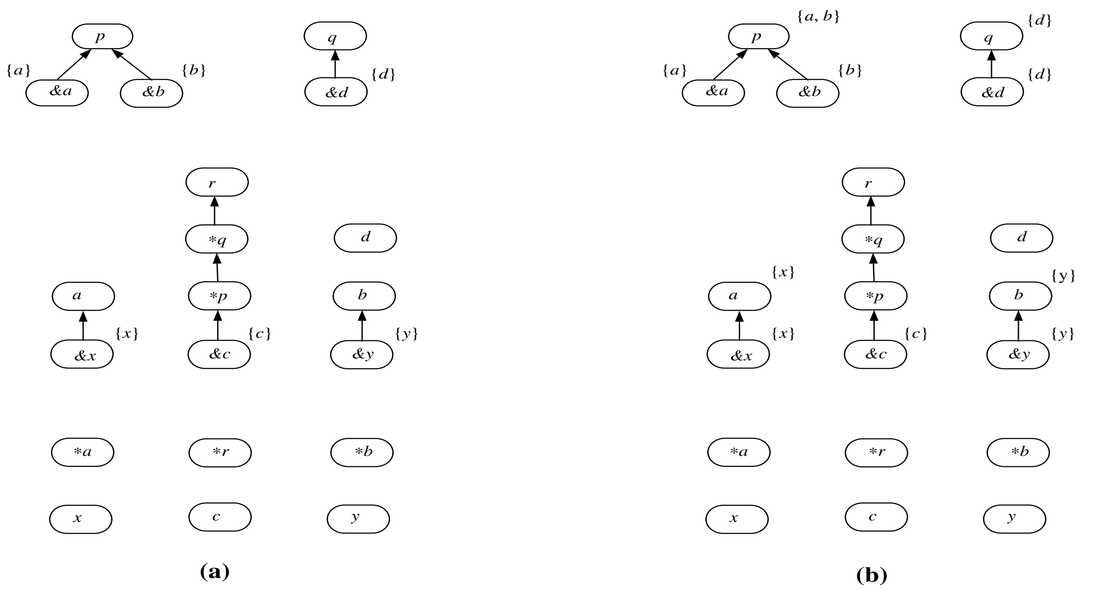


Figure 4.11: Intermediate Pointer Value Flow Graphs

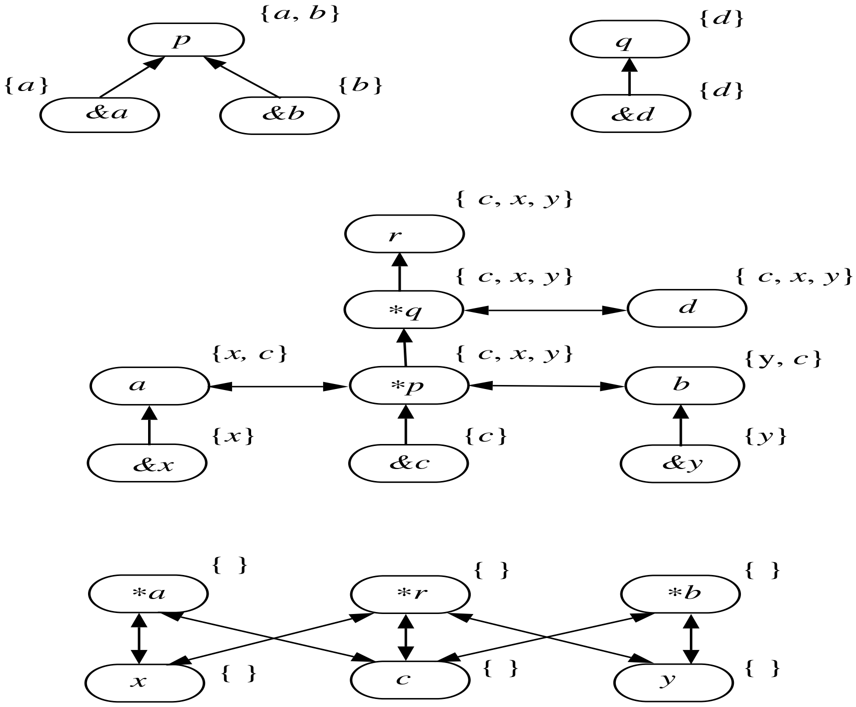


Figure 4.12: The Final Pointer Value Flow Graph

4.11(c). Similarly, we will add a $\longleftrightarrow$ edge between the node for $*p$ and the node for b because we find p points to b. Points-to pairs will be propagated from the node for b to the node for $*p$ and from the node for $\&c$ to the node for b. We have the points-to set for $*p$ to be $\{ c, x, y \}$ and the points-to set for b to be $\{ y, c \}$, as shown in Figure 4.11(d).

The final pointer value flow graph for the example program is given in Figure 4.12; the difference between the final and the initial pointer value flow graph (Figure 4.5) is the $\longleftrightarrow$ edges added. The points-to sets calculated from the equations for the example program are the same as those shown in Section 4.2. The non-trivial aliases derived from the points-to analysis, represented by the $\longleftrightarrow$ edges in Figure 4.12, are shown below as sets of object names.

$\{ a, *p \}$

$\{ x, *a, *r \}$

$\{ b, *p \}$

$\{ c, *a, *r, *b \}$

$\{ d, *q \}$

$\{ y, *b, *r \}$

Each set has exactly one location; all other names in the set are aliased to the location and thus aliased to each other.

Complexity Let N be the number of object names in $\mathbf{O}$, that is, $N = \mid \mathbf{O} \mid$. The number of pointer assignments in $\mathbf{A}$, that is, $\mid \mathbf{A} \mid$, will be bounded above by $\mathcal{O}(N^2)$.

In the routine graph-construction-wo-union-wo-casting(), creation of nodes takes at most $\mathcal{O}(N)$ time and creation of edges takes at most $\mathcal{O}(N^2)$ time. So the construction of the initial pointer value graph takes $\mathcal{O}(N^2)$ time.

The initialization phase of the routine worklist-algorithm-wo-union-wo-casting() takes $\mathcal{O}(N)$ time.

The maximum number of possible points-to pairs is $\mathcal{O}(N^2)$. Each points-to pair can be put into the worklist as most N times, one for each node in the pointer value flow graph. The while loop in the iteration phase of the routine worklist-algorithm-wo-union-wo-casting() has at most $\mathcal{O}(N^3)$ iterations.

Checking if a location is already in a points-to set (the if condition in worklist-algorithm-wo-union-wo-casting()) takes $\mathcal{O}(N)$ time.

Since each points-to set can have at most N locations, the *then* branch of the if condition will be executed at most $\mathcal{O}(N^2)$ times. Each time the *then* branch is executed, each of the two for loops considering $\longrightarrow$ and $\longleftrightarrow$ edges from o respectively takes at most $\mathcal{O}(N)$ time because at most N points-to pairs will be put into the worklist in the loop.

The number of calls to the routine find-an-alias-wo-union-wo-cast() in the iteration phase will be at most $\mathcal{O}(N^2)$ because there are at most $\mathcal{O}(N^2)$ points-to pairs. Even taking into account of the recursive calls to the routine, the number of calls to the routine is still $\mathcal{O}(N^2)$ because there are at most $\mathcal{O}(N^2)$ aliases involving only names in $\mathbf{O}$. In the routine, the loop for Equation (4) takes $\mathcal{O}(N)$ time and the loop for Equation (3) takes $\mathcal{O}(N^2)$ time; the remaining statements take $\mathcal{O}(N)$ time. So the complexity of the routine find-an-alias-wo-union-wo-cast() is $\mathcal{O}(N^2)$.

Putting all together, the complexity of the routine worklist-algorithm-wo-union-wo-cast() is $\mathcal{O}(N^4)$; the complexity of the points-to analysis is also $\mathcal{O}(N^4)$. This is the worst-case complexity of the analysis.

4.4 Unions and Type Casting

In this section, we consider new problems introduced by unions and type casting.

Without unions and type casting, each location can only be accessed in one way consistent with its declared type. A union, however, is a location that contains any one of the several members of various types at different times. Basically there are a number of ways to access the location; each union member corresponds to one. Type casting also allows a location to be accessed in different ways. For instance, we can have a pointer of type $t_1 *$ point to a location declared of type t_2 by casting and thus can access the location as if it is of type t_1. There is a difference between these two cases. In the former, we have the declared union type and know the ways to access the location; we call these unions *declared unions*. In the latter, only by points-to analysis will we know the different ways that a location can be accessed; we call these unions *effective unions*. Of course, a declared union can be accessed as a type other than any of its member types through type casting.

In summary, unions and type casting may cause a location to be accessed in more than one way. Therefore, a problem arises as whether or not two different ways of accessing a location can share some values, that is, whether or not a value written through one way of accessing can be read through another way of accessing. The C standard actually allows such sharing for declared unions; if a union contains several structures that have a common initial sequence, and if the union currently contains one of these structures (i.e., values in the union are stored as *the* structure), it is permitted to refer to the common initial part of any of the several structures and the values of any common initial part will be same as in *the* structure. For example, in the fragment in Figure 4.13, the three structures in the union have the first member *int type* in common. It is alright that the values of the union are last assigned using the member nf and the value $u.n.type$ is inspected using the member n. Furthermore, the values of $u.nf.type$ and $u.n.type$ are the same, that is, they are aliased.

In other words, different ways of accessing a location may potentially induce aliases. Any aliasing or points-to analysis has to find these aliases. Unfortunately, these aliases are closely related to memory size and alignment requirements for various types, which are implementation-dependent in general. By alignment requirements, we mean that locations of certain types must be allocated at certain addresses on real machines. Our approach to finding these aliases is to make reasonable, implementation-independent assumptions about memory size and alignment requirements based on type information (i.e., declared types) in source programs. With these assumptions, we determine aliases induced by different ways of accessing a location. We will find those aliases guaranteed by the C standard and maybe more, but we will not say two names are aliased if the alias is caused by implementation-dependent memory size and alignment requirements. We will trade off generality for practicality and portability, and require input programs

```
union {
        struct {
                int type;
        } n;
        struct {
                int type;
                int intnode;
        }ni;
        struct {
                int type;
                float floatnode;
        } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
        ... sin(u.nf.floatnode) ...
```

Figure 4.13: An Example of Unions

to satisfy a number of restrictions for our analysis. These restriction will be mentioned at the various places and we will summarize all of them in Section 4.4.4.

In Section 4.4.1, we will discuss types, accesses and access patterns. All assumptions we make about memory size and alignment requirements for various types will be presented in this section. In Section 4.4.2 and 4.4.3, we show how to determine aliases for declared and effective unions respectively. In Section 4.4.4, we summarize all restrictions required for our points-to analysis.

4.4.1 Basics

Object Names The first restriction we have is on the object names in input programs: we do not allow type casting in the middle of object names. The following is an example of this kind of casting.

$$((struct\ foo\ *\ )\ p) \rightarrow f$$

This restriction can be satisfied by a little bit program transformation, which we have manually performed for the test programs that violate this restriction. For the above object name, for example, we can use a new variable q of type $(struct\ foo\ *)$ and an assignment with casting:

$$q = (struct\ foo\ *)\ p$$

The name with casting in the middle can thus be replaced by the name $q{\rightarrow}f$.

Because of this restriction, each object name in the input program is used according to its declared type and we can talk about the type of an object name.

Types Our implicit assumption is that declared types provide implementation-independent information of memory sizes and alignment requirements.

Assumption 4.4.1 Different primitive types require different memory sizes and/or different alignment requirements.

□

The implication of this assumption for aliasing is as follows. Let t_1 and t_2 be two different primitive types. If a location can be accessed as either t_1 or t_2, by this assumption, values are *not* shared by the two ways of accessing the location; in other words, no alias is introduced.

Assumption 4.4.2 Different pointer types require same memory size and same alignment requirement.

□

The implication of this assumption is that if a location is accessed as either $t_1\,*$ or $t_2\,*$, values will be shared by the two ways of accessing the location.

To summarize the two assumptions, we define an equivalence relation for *primitive types* and *pointer types* called *AlignmentCompatible*. Let t_1 and t_2 be either primitive types or pointer types. We have the following:

- $AlignmentCompatible(t_1,t_2)$ holds if t_1 and t_2 are the *same* primitive type.

- $AlignmentCompatible(t_1,t_2)$ holds if both t_1 and t_2 are pointer types.

- for any other case, $AlignmentCompatible(t_1,t_2)$ does not hold.

Accesses By an *access*, we mean a location being read or written. We represent an access by a pair consisting of a location name and a type, where the location name indicates the address of and the type indicates the size of the memory being read or written. For instance, suppose variable t is of the following structure type:

$$struct\ foo\ \{\ int\ a;\ float\ b;\ char\ c;\ \}$$

$<t,\ struct\ foo>$ is an access, that is, t can be accessed as *struct foo*. Other possible accesses through t are:

$$<t.a,\ int>,\ <t.b,\ float>, \text{ and } <t.c,\ char>$$

Without type casting, the type in an access will be the declared type of the location name. With type casting, on the other hand, a location can be accessed as a type other than its declared type; thus the type in an access may not be the declared type of the location name. Suppose variable p is declared as follows.

$$struct\ bar\ \{\ float\ x;\ int\ y;\ \}\ *p;$$

With the assignment:

$$p = (struct\ bar\ *)\ \&t;$$

t can be accessed through p as if it is of type *struct bar*; so we have an access:

$$<t,\ struct\ bar>$$

Other possible accesses through p are:

$$<t.x,\ float> \text{ and } <t.y,\ int>$$

Access Patterns An *access pattern* for a location describes *one* way in which the location can be accessed. We will represent an access pattern by a pair; the first part is an access describing the way of accessing the location and the second part is an ordered sequence of accesses of primitive or pointer types made possible by the overall access. The reason for having accesses of primitive or pointer types is that values of primitive and pointer types are written to or read from locations in C and these types are of interest for points-to analyses.

For example, the location t can be accessed as $struct\ foo$ and the access pattern is:

$$(<t,\ struct\ foo>\ ,\ [\ <t.a,\ int>,\ <t.b,\ float>,\ <t.c,\ char>\ ])$$

If there are a number of ways to access a location, there are an equal number of access patterns for the location. For example, the location t can be accessed as either $struct\ foo$ or $struct\ bar$; thus there are two access patterns for t and they are:

$$(<t,\ struct\ foo>\ ,\ [\ <t.a,\ int>,\ <t.b,\ float>,\ <t.c,\ char>\ ])$$

$$(<t,\ struct\ bar>\ ,\ [\ <t.x,\ float>,\ <t.y,\ int>\ ])$$

As another example, we can have the following access patterns for a location f of declared type $float$ if f can be accessed as int through a pointer of type $int\ *$.

$$(<f,\ float>\ ,\ [\ <f,\ float>\ ])$$

$$(<f,\ int>\ ,\ [\ <f,\ int>\ ])$$

Now we define two functions related to access patterns. Given an access pattern ap, $NumberOfAccess(ap)$ is the number of accesses in the second part of the access pattern. For example,

$$NumberOfAccess(\ (<t,\ struct\ bar>\ ,\ [\ <t.x,\ float>,\ <t.y,\ int>\ ])\ ) = 2$$

Given a location name loc, $AccessPatternSet(loc)$ is the set of all access patterns for loc.

An access pattern represents the memory layout for a location accessed in a certain way. Access patterns for a location are used to determine aliases caused by multiple ways of accessing the location. The next two assumptions basically show how to do this.

Assumption 4.4.3 Given an access pattern for a location:

$$(<l, t> , [<l_1, t_1>, ..., <l_n, t_n>])$$

the offset of the access $<l_i, t_i>$, $1 \leq i \leq n$, relative to the start of the location l, depends on t_1, ..., t_{i-1}, t_i.
□

This is consistent with our assumption that the type of a location dictates the memory size and alignment requirement for the location. For an access in a sequence, its offset depends on the types of the accesses before it in the sequence and its own type.

Assumption 4.4.4 Given two access patterns for a location:

$$(<l^1, t^1> , [<l^1_1, t^1_1>, ..., <l^1_n, t^1_n>])$$

$$(<l^2, t^2> , [<l^2_1, t^2_1>, ..., <l^2_m, t^2_m>])$$

and assuming $AlignmentCompatible(t^1_k, t^2_k)$ holds for $1 \leq k < i$, $1 \leq i \leq n$ and $1 \leq i \leq m$, the following are true:

- If $AlignmentCompatible(t^1_i, t^2_i)$ holds, the two accesses, $<l^1_i, t^1_i>$ and $<l^2_i, t^2_i>$, are aliased, that is, a value written through one can be read through another and vice versa.
- If $AlignmentCompatible(t^1_i, t^2_i)$ does not hold, the two accesses, $<l^1_i, t^1_i>$ and $<l^2_i, t^2_i>$, may overlap, but are not aliased. Furthermore, any one access among $<l^1_i, t^1_i>$, ..., $<l^1_n, t^1_n>$ and any one access among $<l^2_i, t^2_i>$, ..., $<l^2_m, t^2_m>$ may overlap, but are not aliased.

□

If $AlignmentCompatible(t_k^1, t_k^2)$ holds for $1 \leq k < i$, $1 \leq i \leq n$ and $1 \leq i \leq m$, the pair of accesses, $<l_i^1,\ t_i^1>$ and $<l_i^2,\ t_i^2>$, require the same memory size and alignment; thus the offsets of the two accesses relative to the start of the location are the same and the two accesses will read from or write to the same location.

On the other hand, if i is the smallest such that $1 \leq i \leq n$, $1 \leq i \leq m$, and $AlignmentCompatible(t_i^1, t_i^2)$ does not hold, the two accesses, $<l_i^1,\ t_i^1>$ and $<l_i^2,\ t_i^2>$, require different memory size and/or alignment; thus the value read through one access will not be the same as the one written through another and vice versa. The accesses following $<l_i^1,\ t_i^1>$ and $<l_i^2,\ t_i^2>$ in the two sequences may be aliased, but this is implementation-dependent and not guaranteed by all implementations; we will assume that none of these accesses are aliased.

4.4.2 Declared Unions

In this section, we present an approach to handling locations declared of union types in the points-to analysis. Basically, a union type defines a number of ways to access a location; in other words, there are a number of access patterns for a location of a union type. These accesses may refer to same memory locations; therefore a union type may introduce aliases among the accesses it defines. For the points-to analysis, we need to determine these aliases.

To simplify the problem, we will only allow the following types for union members. Other types for union members can be handled in a similar fashion, but they will complicate our analysis. We believe the following types are usually used for union members in practice. Note this restriction does not apply to locations other than union members.

- primitive types
- pointer types
- array types

$union\ utag\ \{\ int\ alltypes;$
$\quad struct\ inode\ \{\ int\ type;\ int\ *ival;\ \}\ ni;$
$\quad struct\ rnode\ \{\ int\ type;\ float\ *rval;\ \}\ nr;\ \};$

Figure 4.14: Another Example of Unions

Only arrays of primitive or pointer types allowed.

- structure types

  Each structure member can be either a primitive type or a pointer type; it can also be any array of primitive or pointer types.

  No nested structures are allowed as union members. Bit-fields as structure members are not allowed.

Given a location l of a union type, each union member m defines an access pattern:

$$(<l.m,\, t>\ ,\ [\ <l_1,\, t_1>,\, ...,\, <l_n,\, t_n>\ ])$$

where t is the type for the union member. For instance, let u be a variable of the union type given in Figure 4.14. u has three access patterns, one for each union member:

$$(<u.alltypes,\, int>\ ,\ [\ <u.alltypes,\, int>\ ])$$

$$(<u.ni,\, struct\ inode>\ ,\ [\ <u.ni.type,\, int>,\, <u.ni.ival,\, int*>])$$

$$(<u.nr,\, struct\ rnode>\ ,\ [\ <u.nr.type,\, int>,\, <u.nr.rlval,\, float*>])$$

Note that $u.alltypes$, $u.ni$, and $u.nr$ all have the same offset as u by definition of unions.

We now define the *common initial sequence* for two access patterns of a location.

Definition 4.4.1 Given two access patterns for a location:

$$(<loc^1,\, typ^1>\ ,\ [\ <l_1^1,\, t_1^1>,\, ...,\, <l_n^1,\, t_n^1>\ ])$$

$$(<loc^2,\, typ^2>\ ,\ [\ <l_1^2,\, t_1^2>,\, ...,\, <l_m^2,\, t_m^2>\ ])$$

where t_i^1 and t_i^2 are either primitive types or pointer types, their common initial sequences are

$$\{ <l_1^1, t_1^1>, ..., <l_i^1, t_i^1> \} \text{ and } \{ <l_1^2, t_1^2>, ..., <l_i^2, t_i^2> \}$$

where i is the maximum such that $0 \leq i \leq n$, $0 \leq i \leq m$, and for all j, $1 \leq j \leq i$, $AlignmentCompatible(t_j^1, t_j^2)$ holds.

□

For example, for the following access patterns of u,

$$(<u.alltypes, int>\ , [\ <u.alltypes, int>\ ])$$

$$(<u.ni, struct\ inode>\ , [\ <u.ni.type, int>, <u.ni.ival, int\ *>\ ])$$

the common initial sequences are:

$$\{\ <u.alltypes, int>\ \}$$

$$\{\ <u.ni.type, int>\ \}$$

For the following access patterns of u,

$$(<u.ni, struct\ inode>\ , [\ <u.ni.type, int>, <u.ni.ival, int\ *>\ ])$$

$$(<u.nr, struct\ rnode>\ , [\ <u.nr.type, int>, <u.nr.rval, float\ *>,])$$

the common initial sequences are:

$$\{\ <u.ni.type, int>, <u.ni.ival, int\ *>\ \}$$

$$\{\ <u.nr.type, int>, <u.nr.rlval, float\ *>\ \}$$

By our assumptions, given the common initial sequences of any two access patterns of a location:

$$\{ <l_1^1, t_1^1>, ..., <l_i^1, t_i^1> \} \text{ and } \{ <l_1^2, t_1^2>, ..., <l_i^2, t_i^2> \}$$

```
calculate-access-patterns-for-union(l)
{
    AccessPatternSet(l) = { }
    let t be the declared type of l
    for each union member m of t
    {
        ap = create-an-access-pattern(apply(l,m) , member-type(t , m))
        add-an-access-pattern(l , ap)
    }
}
```

Figure 4.15: calculate-access-patterns-for-union()

the two accesses, $<l_j^1,\ t_j^1>$ and $<l_j^2,\ t_j^2>$, where $1 \leq j \leq i$, are aliased, that is, they refer to the same location. Because *AlignmentCompatible* is an equivalence relation on primitive and pointer types by definition, given all access patterns for a location of a union type, the common initial sequences of any two access patterns imply an equivalence relation on the set of accesses in all the access patterns.

For example, the common initial sequences for variable u imply an equivalence relation given below as two equivalence classes and all accesses in an equivalence relation refer to the same location.

$$\{\ <u.alltypes,\ int>,\ <u.ni.type,\ int>,\ <u.nr.type,\ int>\ \}$$

$$\{\ <u.ni.ival,\ int\ *>,\ <u.nr.rlval,\ float\ *>\ \}$$

In Figure 4.15, 4.16, and 4.17, we provide the routines that calculate access patterns and equivalence classes implied for locations of union types.

Given a location name and a type, the routine create-an-access-pattern() (Figure 4.16) simply returns an access pattern. The routine is simpler than it seems to be. In case the type for the location is an array type, possibly a multi-dimension array type, we need to break down the array into elements of either primitive or pointer types and enumerate these array elements in the access pattern[5]. For instance, for *loc* declared

[5] A possible optimization here is to keep one element and a counter in the access pattern, where the counter tells how many similar elements are in the access pattern.

```
create-an-access-pattern(loc , typ)
{
    if (typ is a primitive type or a pointer type)
    {
        return (<loc, typ> , [<loc, typ> ])
    }
    else if (typ is an array type)
    {
        if (typ is a one-dimension or multi-dimension array type
          of elements of a primitive or pointer type)
        {
            let elem be the location name for the array elements
            let etyp be the primitive or pointer type for array elements
            let n be the total number of the array elements
            return (<loc, typ> , [<elem, etyp>_1, ..., <elem, etyp>_n ])
        }
        else
            error      /* violates restrictions */
    }
    else if (typ is a structure type)
    {
        if (each member of typ is either a primitive type, a pointer type,
          or an array type of elements of a primitive or pointer type)
        {
            let m be the number of members of the structure type
            for the i^th member of the structure type, 1 <= i <= m
            {
                let fl_{i,1}, ..., fl_{i,k_i} be the sequence of locations of primitive or pointer types,
                which can be accessed through the member
                let ft_{i,1}, ..., ft_{i,k_i} be the types of these locations
            }
            return
              (<loc, typ> ,
              [<fl_{1,1}, ft_{1,1}>, ..., <fl_{1,k_1}, ft_{1,k_1}>, ..., <fl_{m,1}, ft_{m,1}>, ..., <fl_{m,k_m}, ft_{m,k_m}>])
        }
        else
            error      /* violates restrictions */
    }
    else
        error      /* violates restrictions */
}
```

Figure 4.16: create-an-access-pattern()

```
add-an-access-pattern(l , ap)
{
      for each access <l, t> of ap
            INIT-EQUIV-CLASS(<l, t>)

      for each access pattern ap_i in AccessPatternSet(l)
      {
            j = 1
            while ( j ≤ NumberOfAccess(ap) and j ≤ NumberOfAccess(ap_i))
            {
                  let <l^j, t^j> and <l_i^j, t_i^j> be the j^th access in ap and ap_i respectively
                  if (AlignmentCompatible(t^j,t_i^j))
                  {
                        UNION( <l^j, t^j> , <l_i^j, t_i^j> )
                        j = j + 1
                  }
                  else
                        break   /* exit the while loop */
            }
      }

      add ap to AccessPatternSet(l)
}
```

Figure 4.17: add-an-access-pattern()

int $loc[2][3]$, the access pattern will be as follows.

$$(<loc,\ int\ [\ ][\ ]>\ ,\ [\ <loc[\ ][\ ],\ int>_1,\ ...,\ <loc[\ ][\ ],\ int>_6\ ])$$

Similarly, if the type for the location is a structure type and one member is an array, we need to enumerate array elements of primitive or pointer types for this member.

Given a location name and a new access pattern for the location, the routine add-an-access-pattern() (Figure 4.17) first initializes each access in the new access pattern as an equivalence class and then compares the accesses in the new access pattern with the ones in each of the existent access patterns for the location. When comparing accesses in two access patterns, the accesses are considered in the order of their representation, that is, the first accesses in the two access patterns, the second accesses in the two access patterns, so on. If two accesses have $AlignmentCompatible$ types, they are aliased by our assumptions and their equivalence classes are unioned; the next accesses in the two access patterns will be considered. When two accesses do not have $AlignmentCompatible$ type, none of the remaining accesses in the two access patterns will be considered aliased by our assumptions; thus the next existent access pattern will be considered. Finally, the new access pattern is added to the set of access patterns for the location.

Given a location of a union type, the routine calculate-access-patterns-for-union() (Figure 4.15) creates an access pattern for each union member by calling create-an-access-pattern() and compares the newly created access pattern with the existent access patterns by calling add-an-access-pattern(). As the result of this calculation, there are a number of equivalence classes; each of them consists of accesses that are aliased to each other.

Take the variable u of *union utag* as an example. Initially, $AccessPatternSet(u)$ is an empty set. When the union member *alltypes* is first considered, the following access pattern is created by create-an-access-pattern():

$$(<u.alltypes,\ int>\ ,\ [\ <u.alltypes,\ int>\ ])$$

The routine add-an-access-pattern() will add the above access pattern to the set

$AccessPatternSet(u)$ and will make an equivalence class consisting of the access $<u.alltypes$, $int>$.

The union member ni is considered next. The following new access pattern is created:

$$(<u.ni,\ struct\ inode>\ ,\ [\ <u.ni.type,\ int>,\ <u.ni.ival,\ int\ *>\ ])$$

Two equivalence classes are created, one for $<u.ui.typ$, $int>$ and another for $<u.ui.ival$, $int*>$. Then the accesses of the access pattern in the set $AccessPatternSet(u)$ are compared to the accesses in the newly created access pattern, one by one. Since the first accesses in the two access patterns, $<u.alltypes$, $int>$ and $<u.ui.typ$, $int>$, have the same primitive type, the equivalence classes for the two accesses are unioned, that is, the two accesses are in the same equivalence class.

Finally the union member nr is considered. The new access pattern associated with this member

$$(<u.nr,\ struct\ rnode>\ ,\ [\ <u.nr.type,\ int>,\ <u.nr.rlval,\ float\ *>\ ])$$

will be examined against the two access patterns created for other two union members. When compared with the access pattern for member $alltypes$, the equivalence classes for $<u.alltypes$, $int>$ and $<u.nr.type$, $int>$ are unioned. When compared with the access pattern for member ni, the equivalence classes for $<u.ni.type$, $int>$ and $<u.nr.type$, $int>$ are unioned; the equivalence classes for $<u.ni.ival$, $int*>$ and $<u.nr.rval$, $float*>$ are unioned. So we obtain the two equivalence classes for the equivalence relation implied by the common initial sequences for variable u.

Complexity We assume that the maximum number of members in unions or structures is a small constant and the number of array elements for arrays either in union members or as union members is a small constant compared to the number of object names in $\mathbf{O}$.

Because of the assumption, the number of accesses in any access patterns is a small constant and the number of access patterns for any location is also a small constant.

So it is easy to see the complexity of the three routines for calculating access patterns and the equivalence relation implied in Figure 4.15, 4.16, and 4.17 is $\mathcal{O}(1)$.

4.4.3 Type Casting

In this section, we present our approach to handling type casting in the points-to analysis.

First of all, we will not allow arbitrary type casting; instead only casting between object names of pointer types is permitted. We use a simple type checking algorithm to find if an input program violates this restriction.

As a result of this restriction, a pointer of type t_1* can be found pointing to a location of type t_2 even if t_1 is not the same type as t_2 and thereafter the location can be accessed through the pointer as if it is of type t_1. In other words, the location is effectively being used as a union of two types, t_1 and t_2.

For example, suppose s is declared to be of the following structure type:

$$struct\ rnode\ \{\ int\ rtype;\ float\ *rval;\ \}$$

There is a pointer p of the following type that points to s:

$$struct\ inode\ \{\ int\ itype;\ int\ *ival;\ \}\ *p$$

Basically s can be accessed through p and itself as a location of any of the two types, that is, s is being used as a union. The locations that can be accessed through s are $<s.rtype,\ int>$ and $<s.rval,\ float*>$. Since p points to s, the dereferenced names $p{\rightarrow}itype$ and $p{\rightarrow}ival$ will access $s.itype$ and $s.ival$ respectively. So the locations that can be accessed through p are $<s.itype,\ int>$ and $<s.ival,\ int*>$.

Similar to declared unions, these effective unions caused by casting may induce aliases. We will use the same approach for declared unions to deal with aliases caused by effective unions. For the above effective union, our calculation will have the following:

- $<s.rtype,\ int>$ and $<s.itype,\ int>$ are aliased; the two accesses are in the same equivalence class.

- $<s.rval,\ float *>$ and $<s.ival,\ int *>$ are aliased; the two accesses are in the same equivalence class.

The same restrictions for declared unions apply to effective unions too. For example, the type restrictions for union members of declared unions will be enforced for effective unions.

4.4.4 Summary of Restrictions

Here we will summarize a number of restrictions that are mentioned at various places in the last two sections and how they are being enforced.

- Type casting in the middle of object names is not allowed.

  We eliminate these type casting by manual program transformation.

- The following are the restrictions on types for members of declared or effective unions.

  - Union members can not be of a union type.
  - Union members can not be of an array type of elements, which are neither of a primitive type nor of a pointer type.
  - Union members can not be of a structure type, which has one member of a structure type.
  - Union members can not be of a structure type, whose members are of an array type of elements, which are neither of a primitive type nor of a pointer type.

  We enforce this by examining the member types of declared unions before the points-to analysis and the member types of effective unions during the points-to analysis.

- Casting from a pointer type to a non-pointer type or vice versa is not allowed.

  We determine if there is such casting by a simple type checking algorithm.

```
graph-construction-w-union-w-casting()
{
    O_T = { }
    for each object name o in O
    {
        let t be the declared type of o
        add <o, t> to O_T
        create a node for <o, t>
    }

    for each pointer assignment lhs = rhs in A
    {
        let ltyp and rtyp be the declared types of lhs and rhs respectively
        if (there is no ⟶ edge from the node for <rhs, rtyp> to the node for <lhs, ltyp>)
            create a ⟶ edge from the node for <rhs, rtyp> to the node for <lhs, ltyp>
    }

    /* calculate access patterns for locations of union types */
    for each location name l in O
    {
        AccessPatternSet(l) = undefined
        if (l is of a union type)
            calculate-access-patterns-for-union(l)
    }

    /* add ⟷ edges due to declared unions */
    for each equivalence class implied by all access patterns
        for each pair of accesses <loc1, typ1> and <loc2, typ2> in the equivalence class
            create a ⟷ edge between the node for <loc1, typ1> and the node for <loc2, typ2>
}
```

Figure 4.18: graph-construction-w-union-w-casting()

4.5 Points-to Analysis With Unions and Type Casting

In this section, we present the points-to analysis algorithm for programs with unions and type casting. To deal with unions and type casting in the analysis, we will associate a type with an object name explicitly. An object name and a type together will now represent a name. Initially, each object name in $\mathbf{O}$ is associated with its declared type. We define a set $\mathbf{O}_\mathcal{T}$ to be initially the following set:

$$\{ <o, t> \mid o \in \mathbf{O} \text{ and } t \text{ is the declared type of } o \}$$

The initial pointer value flow graph has one node for each $<o, t>$. For each pointer assignment $lhs = rhs$ in $\mathbf{A}$, where the declared types of lhs and rhs are $ltyp$ and $rtyp$

respectively, the initial graph will also have a $\longrightarrow$ edge from the node for $<rhs,\ rtyp>$ to the node for $<lhs,\ ltyp>$. For each alias caused by declared unions, for instance, between $<loc_1,\ typ_1>$ and $<loc_2,\ typ_2>$, the initial graph has a $\longleftrightarrow$ edge between the node for $<loc_1,\ typ_1>$ and the node for $<loc_2,\ typ_2>$. The pseudo code for the construction of the initial pointer value flow graph is provided in Figure 4.18.

During the points-to analysis, new names will be created and added to the set $\mathbf{O}_{\mathcal{T}}$; aliases and effective unions will be found. For each new name added to $\mathbf{O}_{\mathcal{T}}$, a new node will be added to the pointer value flow graph. For an alias found, for instance, between $<deref,\ typ>$ and $<loc,\ typ>$, a $\longleftrightarrow$ edge between the node for $<deref,\ typ>$ and the node for $<loc,\ typ>$ will be added to the pointer value flow graph. For each alias caused by effective unions, for instance, between $<loc_1,\ typ_1>$ and $<loc_2,\ typ_2>$, the graph has a $\longleftrightarrow$ edge between the node for $<loc_1,\ typ_1>$ and the node for $<loc_2,\ typ_2>$.

In summary, the pointer value flow graph has one node for each name in $\mathbf{O}_{\mathcal{T}}$ and has the following three kinds of edges:

- $<rhs,\ rtyp> \longrightarrow <lhs,\ ltyp>$

  This edge represents a pointer assignment $lhs = rhs$ in $\mathbf{A}$, where the declared types of lhs and rhs are $ltyp$ and $rtyp$ respectively. Since type casting is possible, $rtyp$ and $ltyp$ may be different pointer types.

- $<loc_1,\ typ_1> \longleftrightarrow <loc_2,\ typ_2>$

  This edge is added either in the construction of the initial graph due to declared unions or during the points-to analysis due to effective unions. By our definition of common initial sequences, typ_1 and typ_2 can be either different pointer types or the same primitive type.

- $<deref,\ typ> \longleftrightarrow <loc,\ typ>$

  This edge is added during the points-to analysis because $<deref,\ typ>$ is found to be aliased to $<loc,\ typ>$.

We will be calculating points-to sets for nodes in the pointer value flow graph.

$$pts(\ <\&loc,\ typ\ *>\ )\ =\ \{\ <loc,\ typ>\ \} \tag{1}$$

$$pts(\ <\&(deref),\ typ\ *>\ )\ =\ \bigcup_{<deref,\ typ>\ \longleftrightarrow\ <loc,\ typ>} \{\ <loc,\ typ>\ \} \tag{2}$$

$$pts(\ <loc,\ typ>\ )\ =\ \rho_{typ} \left(\begin{array}{ll} \displaystyle\bigcup_{<obj_1,\ typ_1>\ \longrightarrow\ <loc,\ typ>} pts(\ <obj_1,\ typ_1>\ )\ \cup & (3a) \\[2ex] \displaystyle\bigcup_{\substack{<loc,\ typ> \\ \updownarrow \\ <deref,\ typ>}} \ \bigcup_{\substack{<obj_2,\ typ_2> \\ \downarrow \\ <deref,\ typ>}} pts(\ <obj_2,\ typ_2>\ ) & (3b) \\[2ex] \cup \displaystyle\bigcup_{<loc,\ typ>\ \longleftrightarrow\ <loc_3,\ typ_3>} pts(\ <loc_3,\ typ_3>\ ) & (3c) \end{array} \right)$$

$$pts(\ <deref,\ typ>\ )\ =\ \rho_{typ} \left(\bigcup_{<deref,\ typ>\ \longleftrightarrow\ <loc,\ typ>} pts(\ <loc,\ typ>\ ) \right) \tag{4}$$

Figure 4.19: Data Flow Equations for the Points-to Analysis with Unions and Casting

Since each node represents a name like $<o,\ t>$, we will be calculating the points-to sets such as $pts(<o,\ t>)$. The data flow equation is shown in Figure 4.19. Comparing these equations with the ones in Figure 4.6, there are three changes. The first change is the naming scheme; now each name consists of an object name and a type. The second one is Item (3c), which is for $\longleftrightarrow$ edges added for aliases caused by declared or effective unions. The third change is the function ρ_{typ} used in the equations, where typ is a pointer type. This function is an extension to the function ρ in Figure 4.6; besides deriving alias information from points-to pairs, this function does simple type conversions; if necessary, it adds new names to $\mathbf{O}_{\mathcal{T}}$, creates new nodes in the pointer value flow graph, calculates new access patterns for effective unions and adds $\longleftrightarrow$ edges for aliases caused by effective unions. The pseudo code of these functionalities applied to one points-to pair is given as a routine type-conversion() in Figure 4.20.

The assumption for using the routine type-conversion() is that the location loc of typ is found by the points-to analysis to be pointed to by a pointer of type typ_1*. The routine will return a location that can be accessed through the pointer. It first tries to convert the location of typ to a location of typ_1. Such conversions include:

- A pointer to a union can be converted to any member of the union and vice versa.

  If typ is a union type and the declared type of one member of the union is typ_1, then the pointer can be considered pointing to the member. This is consistent with the C standard.

  If $<loc,\ typ>$ refers to a union member and the declared type of the union is typ_1, then the pointer can be considered pointing to the union.

  If $<loc,\ typ>$ refers to a union member and the declared type of another union member is typ_1, then the pointer can be considered pointing to that union member.

- A pointer pointing to a structure can be converted to point to the first member of the structure and vice versa.

  If typ is a structure type and the declared type of the first member of the structure

```
type-conversion(typ1 * , <loc, typ>)
{
    if (typ1 is the same type as typ)   return <loc, typ1>

    if (typ is a union type)
        for each member mem of the union type
            if (typ1 is the same type as member-type(typ , mem))
                return <apply(loc,mem), typ1>
    else if (typ is a structure type)
    {
        let fld be the first member of the structure type
        if (typ1 is the same type as member-type(typ , fld))
            return <apply(loc,fld), typ1>
    }

    if (loc is the first member of a structure)
    {
        let sloc be the location name of the structure
        let styp be the declared type of the structure
        if (typ1 is the same type as styp)
            return <sloc, typ1>
    }
    else if (loc is a union member)
    {
        let uloc be the location name of the union
        let utyp be the declared type of the union
        if (typ1 is the same type as utyp)
            return <uloc, typ1>
        else
            for each member mem of the union type
                if (typ1 is the same type as member-type(utyp , mem))
                    return <apply(uloc,mem), typ1>
    }

    if (<loc, typ1> ∉ O_T)
        find-a-new-name(loc , typ1)

    return <loc, typ1>
}
```

Figure 4.20: Type Conversions in the Points-to Analysis

```
find-a-new-name(loc , typ)
{
    create-a-new-name(loc , typ)

    if (AccessPatternSet(loc) == undefined)
    {
        AccessPatternSet(loc) = { }
        if (loc is not a heap name)
        {
            /* create an access pattern for declared type of a non-heap-location */
            let t be the declared type of loc
            ap_0 = create-an-access-pattern(loc , t)
            add-an-access-pattern(loc , ap_0)
        }
    }

    ap = create-an-access-pattern(loc , typ)

    add-an-access-pattern(loc , ap)

    for each access <l, t> of ap
    {
        for each access <l_1, t_1> in FIND(<l, t>)
            if (<l, t> is not same as <l_1, t_1>)
                add a <--> edge between the node for <l, t> and the node for <l_1, t_1>
                if there is no such edge yet
                for each <o, t> in PointsTo<l_1, t_1>
                    add (<l, t> , <o, t>) to the worklist   /* Equation (3c) */
    }
}


create-a-new-name(loc , typ) /* recursively called */
{
    add <loc, typ> to O_T

    create a node for <loc, typ>

    pts(<loc, typ>) = { }

    if (typ is a union type)
        for each member mem of the union type
            create-a-new-name(apply(loc,mem) , member-type(typ , mem))
    else if (typ is a structure type)
        for each member fld of the structure type
            create-a-new-name(apply(loc,fld) , member-type(typ , fld))
    else if (typ is an array type)
            create-a-new-name(apply(loc,[ ]) , elem-type(typ))
}
```

Figure 4.21: Creation of New Names in the Points-to Analysis

is typ_1, then the pointer can be considered pointing to the first member. This is consistent with the C standard.

On the other hand, if $<loc,\ typ>$ refers to the first member of a structure of type typ_1, then the pointer can be considering pointing to the structure.

For example, let u be a variable of the union type given in Figure 4.14. If a pointer of type *struct inode* $*$ is found to point to u, the routine type-conversion() will say the pointer is actually pointing to $u.ni$. If a pointer of type *union utag* is found to point to $u.ni$, the routine type-conversion() will say the pointer is actually pointing to u.

If the above type conversions fail and the name $<loc,\ typ_1>$ is not in $\mathbf{O}_{\mathcal{T}}$, then a new name is just found and the routine find-a-new-name() (Figure 4.21) is called to deal with the new name. The purpose of the type conversions is to avoid creating unnecessary new names. In find-a-new-name(), the routine create-a-new-name() is first called, which will add the new name to the set $\mathbf{O}_{\mathcal{T}}$, create a node for the name in the pointer value flow graph and initialize the points-to set for the node. If the new name is either a union type, a structure type, or an array type, the routine create-a-new-name() will be called recursively for new names involving union members, structure members and array elements. This is for getting *all* new location names. After new names are created, the routine find-a-new-name() will add a new access pattern for the location and update the equivalence classes implied by all access patterns for the location. If any access in the new access pattern is found aliased to any access in the existent access patterns, a $\longleftrightarrow$ edge will be added to the pointer value flow graph between the nodes for the two accesses and the worklist may be updated because of the edge. If the new access pattern is the *first* one for the location, i.e., $AccessPatternSet(loc)$ is $undefined$, we need to add the default access pattern for the location and its declared type, that is, the location can be accessed through itself as the declared type. Note that heap names have to be accessed through pointers and thus there are no default access patterns for them.

For example, suppose s is declared to be of the following structure type:

$$struct\ rnode\ \{\ int\ rtype;\ float\ *rval;\ \}$$

and a pointer p of the following type is found to point to s:

$$struct\ inode\ \{\ int\ itype;\ int\ *ival;\ \}\ *p$$

None of the type conversion rules can be applied. Assume this is the first time that the name $<s,\ struct\ inode>$ is encountered, that is, the name is not in $\mathbf{O}_{\mathcal{T}}$ yet. Thus the routine create-a-new-name() will be called. The new name will be added to $\mathbf{O}_{\mathcal{T}}$ and a new node will be created in the pointer value flow graph for the new name. Since s is accessed through p as a structure, the routine create-a-new-name() will be called for $<s.itype,\ int>$ and $<s.ival,\ int*>$, that is names for the two structure members will be added and nodes for the names will be created. Furthermore, the following two access patterns will be created for s and put in the set $AccessPatternSet(s)$:

$$(<s,\ struct\ rnode>\ ,[\ <s.rtype,\ int>,\ <s.rval,\ float*>])$$

$$(<s,\ struct\ inode>\ ,[\ <s.itype,\ int>,\ <s.ival,\ int*>])$$

The equivalence classes implied by these two access patterns are calculated and they are as follows:

$$\{\ <s.rtype,\ int>\ ,\ <s.itype,\ int>\ \}$$

$$\{\ <s.rval,\ float*>\ ,\ <s.ival,\ int*>\ \}$$

Finally one $\longleftrightarrow$ edge is added between the node for $<s.rtype,\ int>$ and the node for $<s.itype,\ int>$; one $\longleftrightarrow$ edge is added between the node for $<s.rval,\ float*>$ and the node for $<s.ival,\ int*>$. Note that the nodes for these accesses have already been added in the pointer value flow graph by the routine create-a-new-name().

The worklist algorithm for points-to analysis with unions and type casting is shown in Figure 4.22 and 4.23. This algorithm is very similar to the one in Figure 4.7 and 4.8. There are two changes. One is the naming scheme, that is, a name now is an object name and a type. Another change is that the type conversion function is applied to each points-to pair removed from the worklist. The routine type-conversion() and find-an-alias-w-union-w-casting() do what the function ρ_{typ} in the equations is supposed to

```
worklist-algorithm-w-union-w-casting()
{
    /* initialization */
    for each object name <o, t> in O_T
    {
        pts(<o, t>) = { }
        if (<o, t> is <&loc, typ *>)
            add (<&loc, typ *> , <loc, typ>) to the worklist   /* Equation (1) */
    }

    /* iteration */
    while (the worklist is not empty)
    {
        remove (<o1, t1 *> , <l0, t0>) from the worklist

        <l1, t1> = type-conversion(t1 * , <l0, t0>)

        if (<l1, t1> is already in pts(<o1, t1 *>))   continue

        add <l1, t1> to pts(<o1, t1 *>)

        for each ⟶ edge from <o1, t1 *>
        {
            let the edge be from <o1, t1 *> to <o2, t2>
            if (o2 is a location name)
                add (<o2, t2> , <l1, t1>) to the worklist   /* Term (3a) */
            else
                for each ⟷ edge from <o2, t2>
                {
                    let the edge be between <o2, t2> and <o3, t3>
                    add (<o3, t3> , <l1, t1>) to the worklist   /* Term (3b) */
                }
        }

        if (o1 is a location name)
            for each ⟷ edge from <o1, t1 *>
            {
                let the edge be between <o1, t1 *> and <o4, t4>
                add (<o4, t4> , <l1, t1>) to the worklist   /* Term (3c) and Equation (4) */
            }

        /* derive alias information from points-to information */
        if (o1 is neither &loc nor &deref)
            find-an-alias-w-union-w-casting(apply(o1, *) , l1 , t1)
    }
}
```

Figure 4.22: worklist-algorithm-w-union-w-casting()

```
find-an-alias-w-union-w-casting(deref , loc , typ)
{
    if (<deref, typ> ∉ O_T)   return   /* consider only names in O_T */

    if (there is already a ⟷ edge between <deref, typ> and <loc, typ>)
        return

    add a ⟷ edge between <deref, typ> and <loc, typ>

    if (<&deref, typ * > ∈ O_T)
        add (<&deref, typ * > , <loc, typ>) to the worklist   /* Equation (2) */

    for each ⟶ edge to <deref, typ>
    {
        let the edge be from <obj_1, typ_1> to <deref, typ>
        for each <loc_2, typ_2> in pts(<obj_1, typ_1>)
            add (<loc, typ> , <loc_2, typ_2>) to the worklist   /* Term (3b) */
    }

    for each <loc_3, typ_3> in pts(<loc, typ>)
        add (<deref, typ> , <loc_3, typ_3>) to the worklist   /* Equation (4) */

    /* An alias involving union/struct/array implies more aliases. */
    if (typ is a union type)
        for each member mem of the union type
            find-an-alias-w-union-w-casting(apply(deref,mem), apply(loc,mem),
                                            member-type(typ , mem))
    else if (typ is a structure type)
        for each member fld of the structure type
            find-an-alias-w-union-w-casting(apply(deref,fld), apply(loc,fld),
                                            member-type(typ , fld))
    else if (typ is an array type)
            find-an-alias-w-union-w-casting(apply(deref,[ ]) , apply(loc,[ ]) , elem-type(typ))
}
```

Figure 4.23: find-an-alias-w-union-w-casting()

```
points-to-analysis-w-union-w-casting()
{
    /* construct the initial pointer value flow graph */
    graph-construction-w-union-w-casting()

    /* apply the worklist algorithm */
    worklist-algorithm-w-union-w-casting()
}
```

Figure 4.24: Points-to Analysis with Unions and Type Casting

accomplish. The points-to analysis algorithm with unions and type casting is given in Figure 4.24.

We assume that the maximum number of members in unions (declared or effective) or structures is a small constant and the number of array elements for arrays either in union members or as union members is a small constant.

Let N be the number of object names in $\mathbf{O}$, that is, $N = \mid \mathbf{O} \mid$. Initially, $\mathbf{O}_{\mathcal{T}}$ has N names. During the points-to analysis, more names may be added to the set $\mathbf{O}_{\mathcal{T}}$; this happens if a location is being accessed through pointers as a type other than its declared type. We assume the number of locations being accessed in this way and the types that they are being accessed as are both small constants compared to the number of object names in $\mathbf{O}$. Therefore, the number of names in $\mathbf{O}_{\mathcal{T}}$ during the points-to analysis is $\mathcal{O}(N)$.

In the routine graph-construction-w-union-w-casting(), creation of nodes takes at most $\mathcal{O}(N)$ time and creation of $\longrightarrow$ edges takes at most$\mathcal{O}(N^2)$ time. Because the complexity of the routine calculate-access-patterns-for-union() is $\mathcal{O}(1)$, the cost of calculating access patterns for all locations of union types is $\mathcal{O}(N)$. The cost of creating $\longleftrightarrow$ edges is $\mathcal{O}(N)$ as the number of equivalence classes is $\mathcal{O}(N)$ and the number of accesses in each class is $\mathcal{O}(1)$. So the construction of the initial pointer value graph takes at most $\mathcal{O}(N^2)$ time.

Besides the naming scheme, this worklist algorithm is different from the one in Section 4.3 in that the routine type-conversion() is called for each points-to pair removed from the worklist. The complexity of the routine type-conversion() in Figure 4.20, find-a-new-name() and create-a-new-name() in Figure 4.21, is $\mathcal{O}(1)$ because of our assumptions about the number of members in unions or structures and the number of new names added during the points-to analysis. By using $\mathbf{O}_{\mathcal{T}}$ instead of $\mathbf{O}$ and following the same argument as in Section 4.3, we claim the complexity of this worklist algorithm is at most $\mathcal{O}(N^4)$.

The worst case complexity of the points-to analysis with unions and type casting is $\mathcal{O}(N^4)$.

4.6 Empirical Results of the Points-to Analysis

We have implemented a prototype of the points-to analysis algorithm presented in Section 4.5 as a step after program decomposition (Chapter 6). The program decomposition technique uses the points-to analysis on parts of a program to resolve indirect calls through function pointers in the program. So our prototype implementation of the points-to analysis assumes indirect calls have been already resolved.

Our implementation handles pointer arithmetic in a simple way. First, all pointer names as left hand sides of pointer assignments with pointer arithmetic are identified. In our intermediate representation (Figure 6.1), these assignments are of the following form:

$$\mathrm{PtrName} = \mathrm{PtrName}_1 \ +/- \ \mathrm{IntName}$$

For example, $p = q + i$, where p, q are names of pointer type and i is a name of integer type, is such an assignment; we identify p as a pointer involving in pointer arithmetic and consider the assignment as having the same effect as $p = q$. Secondly, we will treat pointers involving in pointer arithmetic (such as p) differently from other pointers in the points-to analysis. Assume the pointer p is found to point to an array element (e.g., $a[\,]$) by the points-to analysis. In this case, the pointer arithmetic is trivially handled because the name for the array element ($a[\,]$) basically can be any element of the array. Assume the pointer p is found to point to a structure member (e.g., $s.f_k$, where f_k is the k^{th} member of the structure type, which has n members: f_1, ..., f_n). Assume the type of the member f_k is t; then p is of type $t\ *$. In this case, p may point to members before or after f_k in the structure because of the pointer arithmetic. For members before f_k, if any, the member f_{k-1} is examined first. If it is of type t or of type t [ ], the points-to analysis will say that p points to $s.f_{k-1}$ or $s.f_{k-1}[\ ]$ and the member before it will be examined; if it is not of the above types, the points-to analysis will not examine any members before it. For members after f_k, if any, the member f_{k+1} is examined first. If it is of type t or of type t [ ], the points-to analysis will say that p points to $s.f_{k+1}$ or $s.f_{k+1}[\ ]$ and the member right before it will be examined; if it is not of the above types, no members before it will be examined.

In this section, we present some empirical results of our prototype implementation of the points-to analysis.

Test Programs In Table 4.1, we present the test C programs used. The programs are ordered by the number of nodes in their internal representation (ICFG). Also shown in the table are the number of lines of code, the number of indirect calls through function pointers, the number of locations of union type, and the number of pointer-related assignments with type casting in each program. Some of the type casting is in assignments for dynamic allocations (i.e., calls to *malloc*() or *calloc*()).

Thru-deref MOD/REF Results In Figures 4.25 and 4.26, we present the Thru-deref MOD/REF results of the PT analysis for the test programs. For comparison, we also show the Thru-deref MOD/REF results of an extended version of the Landi/Ryder flow- and context-sensitive aliasing analysis (the FS analysis) [LR92] for 8 of the test programs; for other test programs, the FS analysis runs out of memory after a while. The extended Landi/Ryder's algorithm handles unions and type casting in a different way from ours. It uses offsets and sizes to represent location names; if two names are overlapped, they are considered aliased.

Overall, the Thru-deref MOD/REF results of the PT analysis are quite good. On average, 4 programs (*smail*, *bc*, 008.*espresso*, *T-W-MC*) have more than 2 locations modified per Thru-deref MOD site and 6 programs (*assembler*, *smail*, *bc*, *bison*, 008.*espresso*, *T-W-MC*) have more than 2 locations referenced per Thru-deref REF site. The worst case is for the program 008.*espresso*. For the 8 programs that go

program name	ICFG nodes	lines of codes	indirect calls	unions	ptr assgn w/ casting
chomp	745	448	0	0	8
loader	1564	1219	0	0	3
stanford	1772	887	0	0	2
pokerd	1896	1241	0	2	3
sim	3034	1439	0	0	18
dineroIII	3477	2961	0	0	2
assembler	3602	2673	0	0	3
smail	3697	3270	0	0	5
archie-client	3856	4680	1	3	29
023.eqntott	4748	3548	9	3	26
rolo	5169	4860	1	0	9
simulator	5575	3733	0	0	3
flex	7377	6970	0	0	75
agrep	8568	3801	0	0	17
bc	8602	7760	19	3	41
zip	9290	7462	1	0	37
bison	9569	7420	0	0	101
022.li	10687	7443	3	0	5
larn	21185	9550	1	0	22
008.espresso	30416	13619	15	0	191
T-W-MC	51628	23788	18	0	273

Table 4.1: Test Programs

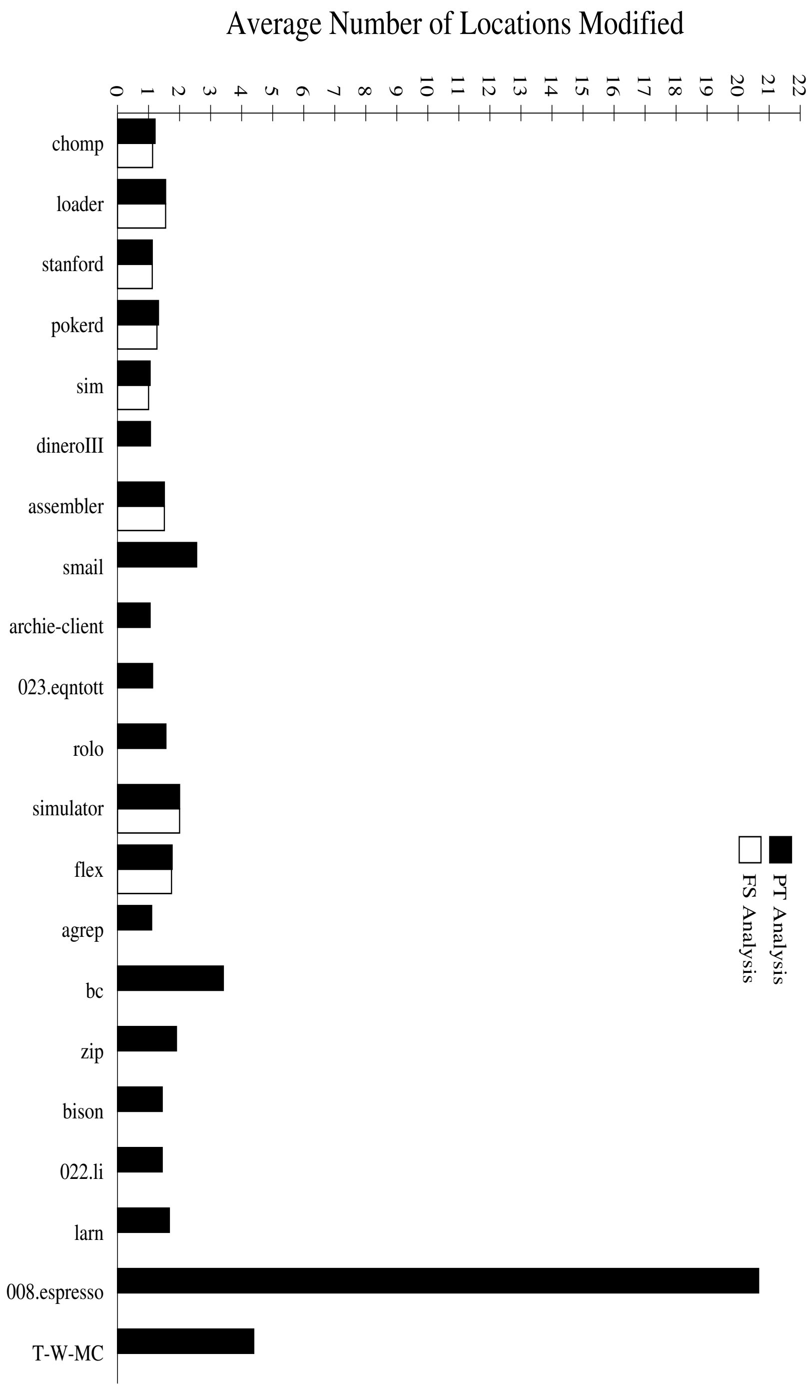


Figure 4.25: Thru-deref MOD Results for the PT Analysis

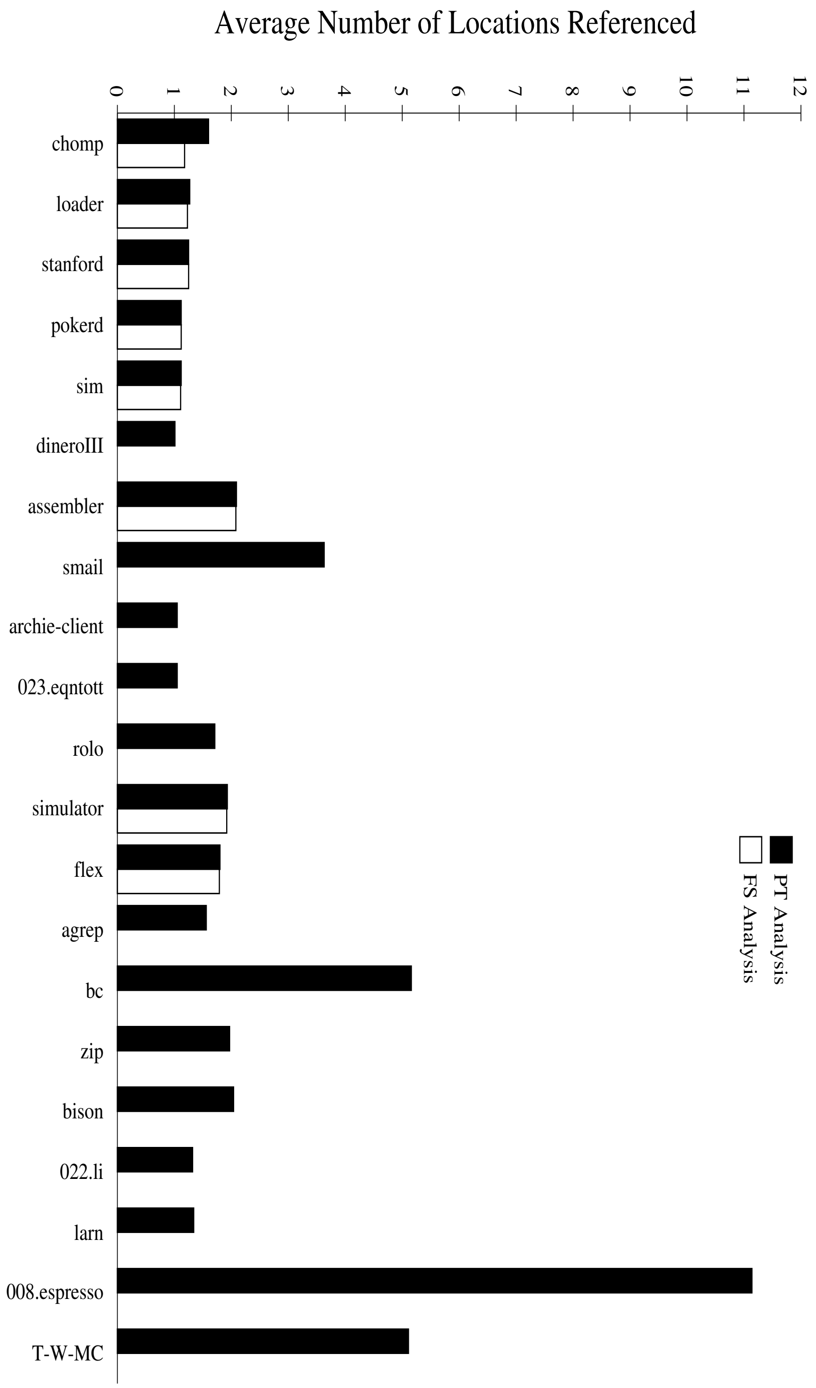


Figure 4.26: Thru-deref REF Results for the PT Analysis

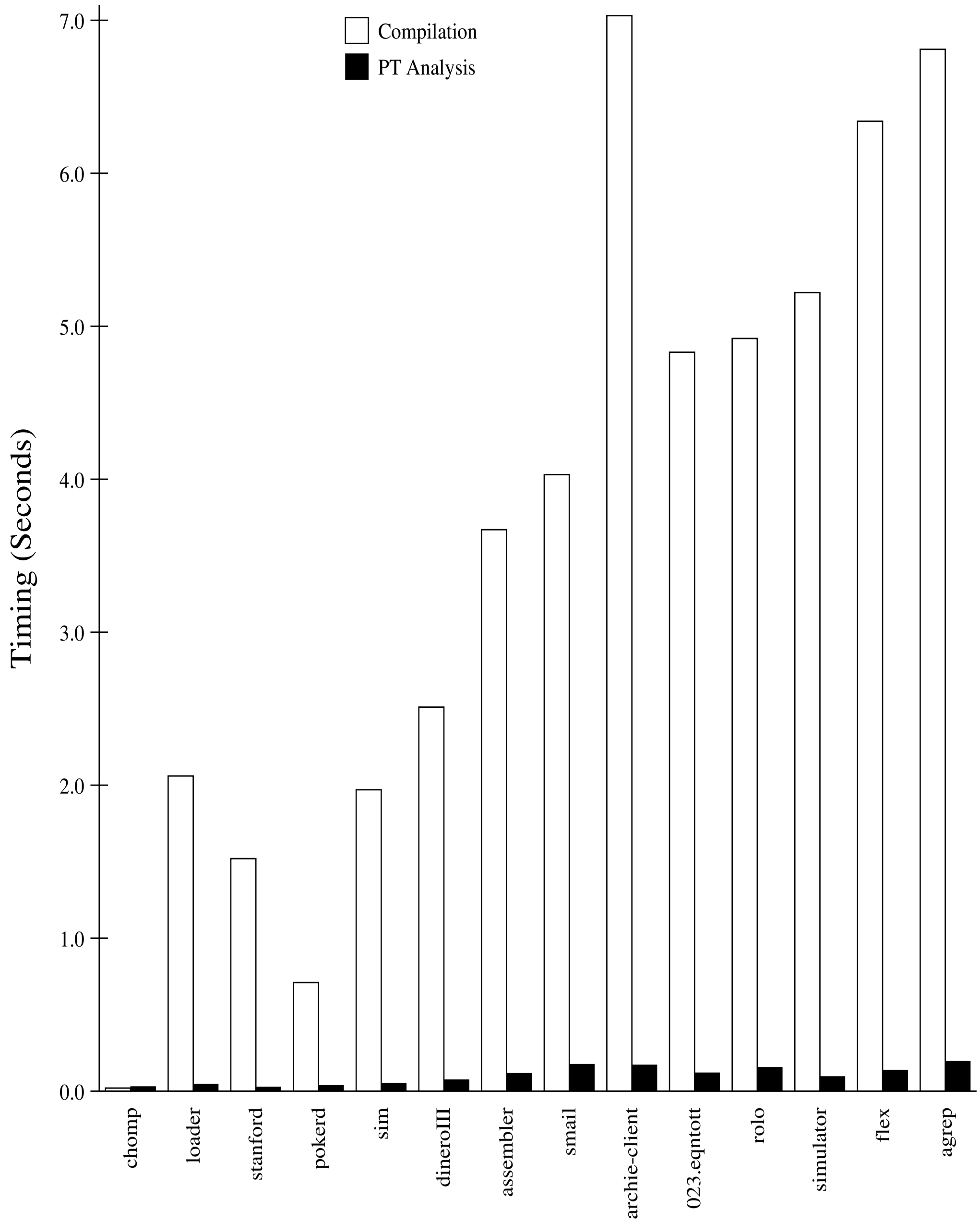


Figure 4.27: Timings for the PT Analysis (Part 1)

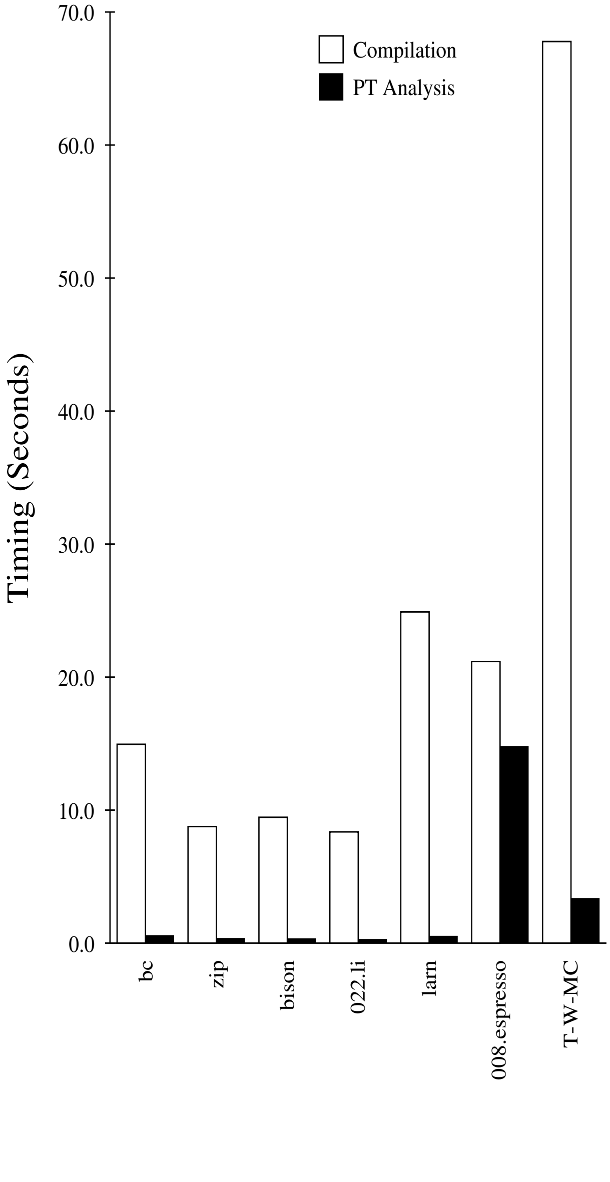


Figure 4.28: Timings for the PT Analysis (Part 2)

through the FS analysis, the PT analysis is almost as precise as the FS analysis. This is very encouraging, but we need to analyze more programs.

Timing Results In Figures 4.27 and 4.28, we present the time used by the PT analysis on each of the test programs; for comparison, we also show the time of a simple compilation using *gcc* without any optimization for each program. By implementation choice, the PT analysis does not resolve indirect calls through function pointers; the time of the PT analysis does not include the time for doing that. The timing results are collected by averaging over ten runs of the PT analysis or the compilation on a Sun SPARCstation 20 running Solaris 2.5.1 with 100M byte physical memory and 350M byte swap space. From the figures, we can see that the PT analysis takes much less time than the compilation for most programs. For all programs but two (008.*espresso* and *T-W-MC*), the PT analysis uses less than 0.6 seconds; it takes about 14.9 seconds for 008.*espresso* and about 3.4 seconds for *T-W-MC* because there are many cases of type casting in the two programs.

4.7 An Example for the Points-to Analysis

In this section, we show the points-to result on a small program. The program is given in Figure 4.29.

There is a pointer assignment with casting in the program, which basically makes the location s of type *struct bar* to be pointed to by a pointer of type $struct\ foo\ *$. Because of this, the points-to analysis will create a new name representing a location of type $struct\ foo$:

$$<s,\ struct\ foo>$$

The name $*p$ will be aliased to this location. Since it is of a structure type, the following names representing the structure members will also be created.

$$<s.a,\ int>,\ <s.b,\ float\ *>, \text{ and } <s.c,\ float\ *>$$

Now the location s is effectively being accessed as a union and it has two access patterns:

```
struct foo
{
    int a;
    float *b;
    float *c;
} *p;

struct bar
{
    int d;
    int *e;
    float f;
    float *g;
} s, *r;

float x,y,z,**f;
```

```
main()
{
    r = &s;
    r→d = 1;
    r→g = &x;

    p = (struct foo *) r;
    p→a = 2;

    p→b = &y;
    *(p→b) = 1.0;

    p→c = &z;
    *(p→c) = 2.0;

    *(r→e) = 1;

    f = &(r→g);
    *f = p→c;
    **f = 1.0;
}
```

Figure 4.29: An Example Program for Points-to Analysis

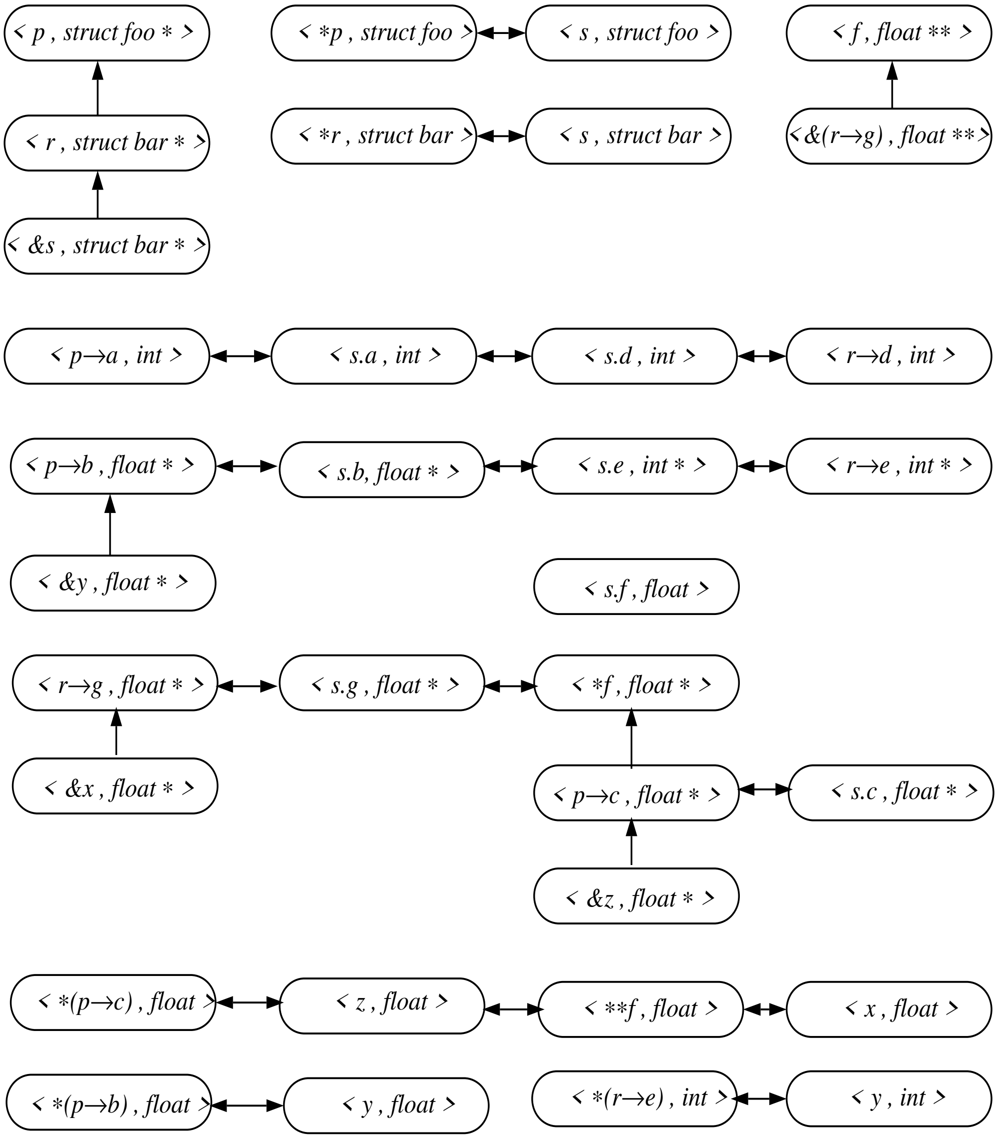


Figure 4.30: The Pointer Value Flow Graph

$pts(\ <\&s,\ struct\ bar\ *>\ ) = \{\ <s,\ struct\ bar>\ \}$
$pts(\ <r,\ struct\ bar\ *>\ ) = \{\ <\mathrm{s},\ struct\ bar>\ \}$
$pts(\ <p,\ struct\ foo\ *>\ ) = \{\ <s,\ struct\ foo>\ \}$
$pts(\ <\&(r\to g),\ float\ **>\ ) = \{\ <s.g,\ float\ *>\ \}$
$pts(\ <f,\ float\ **>\ ) = \{\ <s.g,\ float\ *>\ \}$
$pts(\ <\&y,\ float\ *>\ ) = \{\ <y,\ float>\ \}$
$pts(\ <p\to b,\ float\ *>\ ) = \{\ <y,\ float>\ \}$
$pts(\ <s.b,\ float\ *>\ ) = \{\ <y,\ float>\ \}$
$pts(\ <s.e,\ int\ *>\ ) = \{\ <y,\ int>\ \}$
$pts(\ <r\to e,\ int\ *>\ ) = \{\ <y,\ int>\ \}$
$pts(\ <\&x,\ float\ *>\ ) = \{\ <x,\ float>\ \}$
$pts(\ <\&z,\ float\ *>\ ) = \{\ <z,\ float>\ \}$
$pts(\ <p\to c,\ float\ *>\ ) = \{\ <z,\ float>\ \}$
$pts(\ <s.c,\ float\ *>\ ) = \{\ <z,\ float>\ \}$
$pts(\ <r\to g,\ float\ *>\ ) = \{\ <x,\ float>,\ <z,\ float>\ \}$
$pts(\ <s.g,\ float\ *>\ ) = \{\ <x,\ float>,\ <z,\ float>\ \}$
$pts(\ <*f,\ float\ *>\ ) = \{\ <x,\ float>,\ <z,\ float>\ \}$

Figure 4.31: Points-to Solution for the Example Program

$\{\ <s,\ struct\ bar>,\ <*r,\ struct\ bar>\ \}$
$\{\ <s,\ struct\ foo>,\ <*p,\ struct\ foo>\ \}$
$\{\ <s.a,\ int>,\ <s.d,\ int>,\ <p\to a,\ int>,\ <r\to d,\ int>\ \}$
$\{\ <s.b,\ float\ *>,\ <s.e,\ int\ *>,\ <p\to b,\ float\ *>,\ <r\to e,\ int\ *>\ \}$
$\{\ <s.g,\ float\ *>,\ <r\to g,\ float\ *>,\ <*f,\ float\ *>\ \}$
$\{\ <s.c,\ float\ *>,\ <p\to c,\ float\ *>\ \}$
$\{\ <z,\ float>,\ <*(p\to c),\ float>,\ <**f,\ float>\ \}$
$\{\ <x,\ float>,\ <**f,\ float>\}$
$\{\ <y,\ float>,\ <*(p\to b),\ float>\ \}$
$\{\ <y,\ int>,\ <*(r\to e),\ int>\ \}$

Figure 4.32: Alias Solution Derived from the Points-to Analysis

---

$$(<s,\ struct\ bar>\ ,\ [\ <s.d,\ int>,\ <s.e,\ int\ *>,\ <s.f,\ float>,\ <s.g,\ float\ *>\ ])$$

$$(<s,\ struct\ foo>\ ,\ [\ <s.a,\ int>,\ <s.b,\ float\ *>,\ <s.c,\ float\ *>\ ])$$

The equivalence classes implied by the above access patterns, are as follows.

$$\{\ <s.a,\ int>,\ <s.d,\ int>\ \}$$

$$\{\ <s.b,\ float\ *>,\ <s.e,\ int\ *>\ \}$$

In other words, $<s.a,\ int>$ and $<s.d,\ int>$ refer to the same location; $<s.b,\ float\ *>$ and $<s.e,\ int\ *>$ refer to the same location.

The points-to analysis will find $<s.b,\ float\ *>$ points to $<y,\ float>$. Because of the above aliases, $<s.e,\ int\ *>$ points to $<y,\ float>$. Thus the location y is accessed as a union. The following name will be created by the points-to analysis:

$$<y,\ int>$$

The location y will have the following two access patterns, which do not imply any equivalence class.

$$(<y,\ float>\ ,\ [<y,\ float>])$$

$$(<y,\ int>\ ,\ [<y,\ int>])$$

The final pointer value flow graph is shown in Figure 4.30. Each $\longrightarrow$ edge represents a pointer assignment in the program and each $\longleftrightarrow$ edge represents an alias between two names. The points-to solution for the program is given in Figure 4.31.

The alias solution for the program can derived in the following way. Each name representing a location in the final pointer value flow graph will be examined; all names having a $\longleftrightarrow$ edge to the location name are aliased to each other and to the location name, which means we can represent these aliases by an equivalence class consisting of all names involved. For instance, we will have the following equivalence class representing three aliases among the three names:

$$\{ <z,\ float>,\ <*(p{\rightarrow}c),\ float>,\ <**f,\ float> \}$$

If two location names are aliased due to unions (e.g., $<s.a,\ int>$ and $<s.d,\ int>$), then all names having a $\longleftrightarrow$ edge to any of the location names are aliased to each other and to any of the location names. For instance, we would have the following equivalence class representing six aliases among the four names:

$$\{ <s.d,\ int>,\ <s.a,\ int>,\ <r{\rightarrow}d,\ int>,\ <p{\rightarrow}a,\ int> \}$$

The alias solution for the program is given in Figure 4.32. Trivial equivalence classes with only one name are not shown.

Chapter 5

Program Decomposition and Flow-insensitive Aliasing Analysis without Unions and Type Casting

For flow-insensitive pointer aliasing analysis, a program can be considered as a sequence of assignments having effects on pointer aliasing. In this chapter, we present a technique that partitions these assignments into independent sets with respect to their aliasing effects. We call this program decomposition for pointer aliasing analysis; one potential application of the decomposition is using different analysis methods on these independent sets of assignments to achieve specific goals of analysis cost and precision. We also present a flow-insensitive aliasing algorithm based on the program decomposition called *FA analysis*. In this chapter, we consider programs without indirect calls through function pointers, unions, or type casting (except casting in calls to system-defined memory allocation routines). In Chapter 6, we will extend the ideas in this chapter to programs with the above features. This chapter is organized as follows. In Section 5.1, we discuss the program representation used in this chapter. In Section 5.2, we define interesting equivalence relations on names in a program. In Section 5.3 and 5.4, we present, respectively, the definition and calculation of the equivalence relation for program decomposition and the equivalence relation for the FA analysis.

5.1 Program Representation

We consider C programs that do not have type casting, except in calls to system-defined memory allocation routines such as *malloc*() and *calloc*(). Call-by-value parameter passing is assumed. We represent a program in an intermediate form, whose syntax is given in Figure 5.1. Basically a program consists of a number of procedures. One of these procedures is *main*() and another is a special procedure *_init_*(), which initializes all global variables and calls *main*(). A procedure is a sequence of statements; the first one has to be the entry and the last the exit. Call and return statements are used to represent procedure calls. There are a number of kinds of assignment statements including non-pointer assignment with a basic arithmetic or relational operation, three kinds of pointer assignments, structure assignment with both sides being of structure types. Because of our assumption that there is no type casting, both sides of a pointer assignment or a structure assignment have the same type. Other statements allowed are: heap allocation, heap deallocation, conditional goto and goto. Statements have IDs associated with them, where an ID is in the range of 1..(# of statements). Conditional and goto statements use statement IDs for their destinations. This is a quadruple representation and it can be depicted in a graphical form such as the ICFG[LR92]. Object names are used extensively in the representation.

5.2 Relations on Sets of Object Names

For easy presentation, we will not consider accessor [ ] and names with [ ] in them in this chapter. We will consider those names in the next chapter.

We now define sets of object names, and relations on sets of object names, in which we are interested.

Definition 5.2.1 A set of object names, S, is *closed with respect to prefixes* if the following are true:

- If $o \in$ S and o is of the form $\&o_1$, then $o_1 \in$ S.
- If $o \in$ S and o is of the form $o_1.member$, then $o_1 \in$ S.
- If $o \in$ S and o is of the form $*o_1$, then $o_1 \in$ S.

□

For example, the set { $p{\rightarrow}g$, $\&x$ } is not closed with respect to prefixes because $*p$ is not in the set. The set { p, $*p$, $p{\rightarrow}g$, $\&x$, x } is closed.

Program ::=	$(\text{Procedure})^{+}$	
Procedure ::=	$(\text{L: Statement})^{+}$	(labeled statements)
Statement ::=	entry of P(FmlName_1,...,FmlName_m)	(procedure entry)
	| exit of P	(procedure exit)
	| call P(ArgName_1,...,ArgName_m)	(call statement)
	| return from P	(return statement)
	| PrmName = Exp	(non pointer assignment)
	| PtrName = NULL	(pointer assignment)
	| PtrName = &Name	(pointer assignment)
	| PtrName = PtrName_1	(other pointer assignment)
	| StrtName = StrtName_1	(structure assignment)
	| HeapAlloc(PtrName)	(heap allocation)
	| HeapDealloc(PtrName)	(heap deallocation)
	| if (PrmName) (goto L_1) (goto L_2)	(conditional goto)
	| goto L	(goto statement)

Exp ::=	Constant	(constant)
	| PrmName	(object name of primitive types)
	| Exp_1 Op Exp_2	(arithmetic or relational operation)

FmlName_1, ..., FmlName_m: formal variable names
PrmName, PrmName_1, PrmName_2: object names or constants of primitive types
PtrName, PtrName_1: object names of pointer types
StrtName, StrtName_1: object names of structure types
Name, ArgName_1, ..., ArgName_m: object names or constants of any type
L, L_1, L_2: IDs for statement
Op: primitive operators (e.g., arithmetic, relational)
NULL: nil pointer

Figure 5.1: Intermediate Representation

Definition 5.2.2 A relation R on a set of object names is said to be *type consistent* if for any $(o_1,o_2) \in$ R, o_1 and o_2 have the same type.

□

We will be interested in type consistent relations on sets of object names closed with respect to prefixes.

Definition 5.2.3 Let S be a set of object names closed with respect to prefixes and R be a relation on S. R is a *weakly right-regular* relation on S if the following are true:

- If $(o_1,o_2) \in$ R, both $o_1^{'} = apply(o_1, *)$ and $o_2^{'} = apply(o_2, *)$ are in S, then $(o_1^{'},o_2^{'}) \in$ R.
- If $(o_1,o_2) \in$ R, both $o_1^{'} = apply(o_1, member)$ and $o_2^{'} = apply(o_2, member)$ are in S, then $(o_1^{'},o_2^{'}) \in$ R.

□

For example, let S be the set $\{\ p,\ *p,\ p{\rightarrow}g,\ \&x,\ x,\ x.g\ \}$. The relation $Z = \{$ (p,$\&x$) $\}$ on S is not weakly right-regular because $(*p,x)$ and $(p{\rightarrow}g,x.g)$ are not in Z. The relation $\{\ (p,\&x),\ (*p,x),\ (p{\rightarrow}g,x.g)\ \}$ is a weakly right-regular relation on S.

Lemma 5.2.1 Let S be a set of object names closed with respect to prefixes and R be a weakly right-regular relation on S. If there are a tuple $(o_1,o_2) \in$ R and a sequence of accessors $A = a_1a_2...a_n$ $(n \geq 1)$ such that both $o_1^{'} = apply^*(o_1,A)$ and $o_2^{'} = apply^*(o_2,A)$, are in S, then $(o_1^{'},o_2^{'}) \in$ R.

□

We give a brief proof here.

Consider any prefix of A, $a_1a_2...a_j$, where $1 \leq j \leq n$. Because both $o_1^{'}$ and $o_2^{'}$ are in S, and S is closed with respect to prefixes, it can be proved by induction on j that the object names, $apply^*(o_1,a_1a_2...a_j)$ and $apply^*(o_2,a_1a_2...a_j)$, are in S. Because R is a weakly right-regular relation on S, it can be proved by induction on j that $(apply^*(o_1,a_1a_2...a_j),apply^*(o_2,a_1a_2...a_j)) \in$ R.

□

Definition 5.2.4 Let S be a set of object names closed with respect to prefixes and R be a relation on S. R^e is the smallest[1] equivalence relation on S containing R.

□

In another words, R^e is the reflexive, symmetric and transitive closure of R.

Definition 5.2.5 Let S be a set of object names closed with respect to prefixes and R be a relation on S. R^{wr} is the smallest weakly right-regular equivalence relation on S containing R.

□

If R is type consistent, both R^e and R^{wr} are type consistent.

Since both R^e and R^{wr} are equivalence relations, they can be represented as sets of equivalence classes. For example, let S be the set $\{\ p,\ *p,\ p{\rightarrow}g,\ \&x,\ x,\ x.g\ \}$ and R = $\{\ (p,\&x)\ \}$ be a relation on S. R^e consists of the following five equivalence classes:

$$\{\ p,\ \&x\ \}\quad \{\ *p\ \}\quad \{\ p{\rightarrow}g\ \}\quad \{\ x\ \}\quad \{\ x.g\ \}$$

And R^{wr} consists of the following three equivalence classes:

$$\{\ p,\ \&x\ \}\quad \{\ *p,\ x\ \}\quad \{\ p{\rightarrow}g,\ x.g\ \}$$

R^e and R^{wr} can also be represented as graphs, where nodes represent equivalence classes and edges represent *direct* prefix relation (either $*$ or *member*) between object names in equivalence classes.

5.3 The PE Relation

5.3.1 Definition of the PE Relation

First, we define the concept of *pointer-related assignment* in a program; a statement will be of interest for compile-time aliasing analysis if it contains one or more pointer-related assignments.

[1]By *smallest*, we mean the least number of tuples.

```
struct st {                    main()
    int *f;                    {
    int g;                         int z, u, w, i, *r, *y, **q;
} x, *p, *tt;                      z = 0;
                                   p = &x;
                                   p→f = &z;
                                   p→g = 0;
                                   tt = p;

                                   q = &y;
                                   *q = &w;
                                   r = &u;
                                   *r = 1;
                                   *q = r;
                                   i = *(p→f) + **q;
                               }
```

Figure 5.2: An Example Program

Definition 5.3.1.1 A pointer-related assignment in a program is one of the following:

- a pointer assignment
- a structure assignment such that the structure type contains pointer members
- a formal-actual pair at a call statement such that the two are either of pointer type or of structure type with pointer members
- a heap allocation, HeapAlloc(p), which has the same effect as the pointer assignment: $p = \&heap_{id}$, where id is the statement ID of the heap allocation.
- a heap deallocation, HeapDealloc(p), which, besides the deallocation, has the same effect as the pointer assignment: $p =$ NULL.

□

Throughout this section, we will use the program in Figure 5.2 as an example. The following are the pointer-related assignments in the program:

$p = \&x$

$p \rightarrow f = \&z$

$tt = p$

$q = \&y$

$*q = \&w$

$r = \&u$

$*q = r$

For the purpose of aliasing analysis, a program can be considered as a sequence of pointer-related assignments of the form: $lhs = rhs$, where lhs is an object name and rhs is either an object name or NULL.

Definition 5.3.1.2 The set of object names used in a program, B_0, is defined inductively below:

- If an object name o syntactically appears anywhere in the program[2], then $o \in B_0$.
- If $o \in B_0$ is of a structure type, for each member *member* of the structure, $apply(o{,}member) \in B_0$.
- If $o \in B_0$ and o is of the form $\&o_1$, then $o_1 \in B_0$.
- If $o \in B_0$ and o is of the form $o_1.member$, then $o_1 \in B_0$.
- If $o \in B_0$ and o is of the form $*o_1$, then $o_1 \in B_0$.

$\square$

By definition, B_0 is closed with respect to prefixes.

For the example program,

$$B_0 = \left\{ \begin{array}{c} p,\ *p,\ p{\to}f,\ *(p{\to}f),\ p{\to}g,\ \&x,\ x,\ x.f,\ x.g,\ tt, \\ q,\ *q,\ **q,\ \&y,\ y,\ r,\ *r,\ \&z,\ z,\ \&w,\ w,\ \&u,\ u \end{array} \right\}$$

Definition 5.3.1.3 Given a program and the set B_0 for the program, let R_0 be a relation on B_0 defined below:

---

[2]Heap names are considered appearing in dynamic allocation statements.

$$\mathrm{R}_0 = \left\{ (lhs\ ,rhs) \;\middle|\; \begin{array}{l} lhs = rhs \text{ is a pointer-related assignment} \\ \text{in the program and } rhs \text{ is not NULL} \end{array} \right\}$$

We call ${\mathrm{R}_0}^{wr}$ the PE (Pointer-related-assignment-induced-Equality) relation.

□

Intuitively, a tuple (o_1,o_2) is in R_0 if the two object names have the same value at a program point due to one pointer-related assignment. A tuple (o_1,o_2) is in the PE relation if the two object names have the same value due to a sequence of pointer-related assignments; the PE relation also contains other tuples as pointer-related assignments are considered symmetric[3] and the control flow in the program is not taken into account.

The PE relation is a value-equality relation. Suppose two object names have the same value. If both are of pointer type, their dereferences will contain the same value; if both are of structure type, their corresponding members have the same values. This is why the PE relation needs to be weakly right-regular.

Because we only consider programs without type casting, the relation R_0 in the above definition is type consistent. So the PE relation is also type consistent.

For the example program, the PE relation has eight equivalence classes, shown in Figure 5.3. For each pointer-related assignment in the program, $lhs = rhs$, lhs and rhs (if it is not NULL) are in the same equivalence class of the PE relation. So each pointer-related assignment can be considered associated with a unique equivalence class of the PE relation. In Figure 5.3, we show the set of pointer-related assignments in the example program for each equivalence class of the PE relation.

5.3.2 Calculation of the PE Relation

We assume the following routines are available for initializing and maintaining equivalence classes:

[3]This is same as saying that for each pointer-related assignment, $lhs = rhs$, there is also an assignment $rhs = lhs$.

equivalence classes:	pointer-related assignments:
$\{\ p,\ tt,\ \&x\ \}$	$\{\ p = \&x,\ tt = p\ \}$
$\{\ *p,\ x\ \}$	$\{\ \}$
$\{\ p{\rightarrow}f,\ x.f,\ \&z\ \}$	$\{\ p{\rightarrow}f = \&z\ \}$
$\{\ p{\rightarrow}g,\ x.g\ \}$	$\{\ \}$
$\{\ *(p{\rightarrow}f),\ z\ \}$	$\{\ \}$
$\{\ q,\ \&y\ \}$	$\{\ q = \&y\ \}$
$\{\ *q,\ y,\ r,\ \&w,\ \&u\ \}$	$\{\ *q = \&w,\ r = \&u,\ *q = r\ \}$
$\{\ w,\ u\ \}$	$\{\ \}$

Figure 5.3: The PE Relation for the Example Program

- INIT-EQUIV-CLASS(o), where o is an object name, initializes an equivalence class with one object name, o.
- FIND(o), where o is an object name, returns the equivalence class for o.
- UNION(e_1,e_2), where e_1 and e_2 are two equivalence classes, returns an equivalence class that consists of the object names in both e_1 and e_2.

We further assume that the cost of each call to INIT-EQUIV-CLASS() is a constant time and the cost of each call to FIND() or UNION() depends on the data structure chosen to represent equivalence classes.

Assuming the set B_0 is available, the algorithm calculating the PE relation is given in Figure 5.4. The algorithm has two phases. In Phase 1, an equivalence class is created for each object name in B_0. Each class maintains a prefix relation between its object names and those in other classes. The initial PREFIX set for an equivalence class e with one object name, o, is one of the following cases:

- $\{\ (f_1,apply(o,f_1)),\ ...,\ (f_i,apply(o,f_i))\ \}$, if o is of a structure type with members, f_1, ..., f_i.
- $\{\ (*,apply(o,*))\ \}$, if $apply(o,*) \in B_0$.
- $\{\ \}$, otherwise.

Here are some of the initial PREFIX sets for the example program in Figure 5.3:

$$\text{PREFIX}(\text{FIND}(*p)) = \{\ (g, p \rightarrow g),\ (f, p \rightarrow f)\ \}$$

$$\text{PREFIX}(\text{FIND}(p)) = \{\ (\ *\ , *p)\ \}$$

$$\text{PREFIX}(\text{FIND}(tt)) = \{\ \}$$

Intuitively, if there is a tuple $(a\ ,\ o) \in \text{PREFIX}(e)$, where a is an accessor, o is an object name and e is an equivalence class, then there are an object name o_1 in e and an object name o_2 in FIND(o) such that $o_2 = apply(o_1,a)$. If equivalence classes are represented by nodes in a graph, a tuple $(a\ ,\ o) \in \text{PREFIX}(e)$ represents an edge from the node for e to the node for FIND(o).

In Phase 2, we go through all pointer-related assignments in the program and union equivalence classes. When two classes are unioned, their prefix sets are examined; any two equivalence classes having the same prefix relation with the two classes will be unioned; this is done by the recursive calls to the MERGE() routine. This makes sure the weakly right-regular property is satisfied.

As an example, suppose the assignment $p = \&x$ of the program in Figure 5.3 is the *first* being considered. The equivalence classes, FIND(p) and FIND($\&x$), are merged. Their PREFIX sets are:

$$\text{PREFIX}(\text{FIND}(p)) = \{\ (\ *\ , *p)\ \}$$

$$\text{PREFIX}(\text{FIND}(\&x)) = \{\ (\ *\ , x)\ \}$$

Thus the equivalence classes, FIND($*p$) and FIND(x), are merged. These two classes have the following PREFIX sets:

$$\text{PREFIX}(\text{FIND}(*p)) = \{\ (g, p \rightarrow g),\ (f, p \rightarrow f)\ \}$$

$$\text{PREFIX}(\text{FIND}(x)) = \{\ (g, x.g),\ (f, x.f)\ \}$$

Therefore, FIND($p \rightarrow g$) and FIND($x.g$) will be merged; so will FIND($p \rightarrow f$) and FIND($x.f$).

The algorithm can be thought as a graph algorithm, in which equivalence classes are nodes of a graph and tuples in the PREFIX relations of equivalence classes are edges of the graph. When two equivalence classes are unioned, their nodes are replaced by

```
calculate-PE-relation()
{
  /* Phase 1 */
  for each o ∈ B_0
  {
     INIT-EQUIV-CLASS(o)
     PREFIX(FIND(o)) = { }
  }
  for each o ∈ B_0
  {
     if (o == &o_1)
       add ( * , o_1) to PREFIX(FIND(o))
     else if (o == *o_1)
       add ( * , o) to PREFIX(FIND(o_1))
     else if (o == o_1.member)
       add (member , o) to PREFIX(FIND(o_1))
  }
  /* Phase 2 */
  for each pointer-related assignment, lhs = rhs, where rhs ≠ NULL
     if (FIND(lhs) ≠ FIND(rhs))
       MERGE(FIND(lhs) , FIND(rhs))
}

MERGE(e_1,e_2)
{
     e = UNION(e_1 , e_2) /* union the two classes */
     /* calculate the new prefix relation */
     new-prefix = PREFIX(e_1)
     for each (a,o) ∈ PREFIX(e_2)
        if there is (a_1,o_1) ∈ new-prefix such that a == a_1 †
        {
          if (FIND(o) ≠ FIND(o_1))
            MERGE(FIND(o) , FIND(o_1))
        }
        else
          new-prefix = new-prefix ∪ { (a,o) }
     PREFIX(e) = new-prefix /* set the prefix relation */
}
```

† That is, the two accessors are either $*$ or the same member name.
We assume each structure member has a unique name.

Figure 5.4: Calculation of the PE Relation

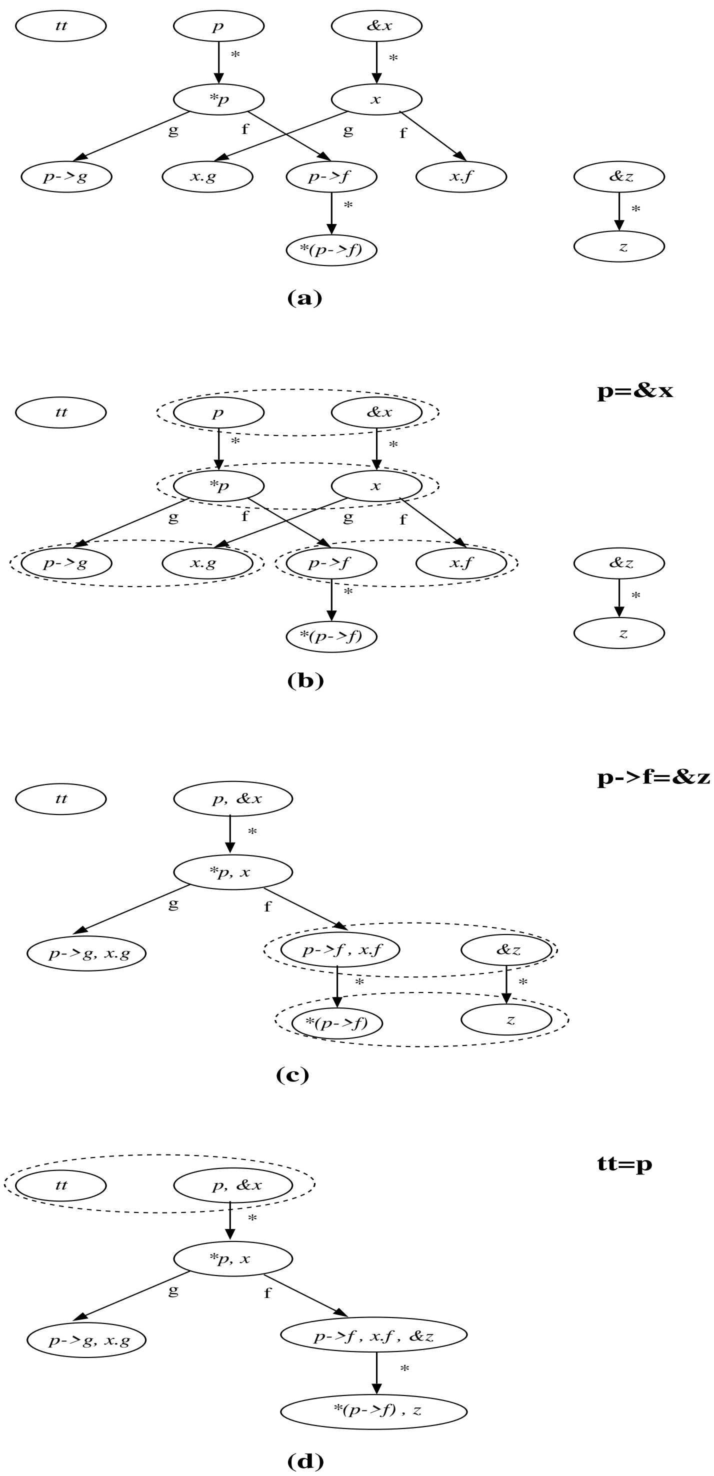


Figure 5.5: Intermediate Graphs in Calculating PE Relation

a new node; all incoming edges to the two nodes and all outgoing edges from the two nodes become incoming edges and outgoing edges of the new node respectively. To illustrate the idea, in Figure 5.5, we show the intermediate graphs during calculation of the PE relation for the example program in Figure 5.2. To make it simple, we only consider the following names:

$$tt,\ p,\ *p,\ p{\rightarrow}f,\ *(p{\rightarrow}f),\ p{\rightarrow}g,\ \&x,\ x,\ x.f,\ x.g,\ \&z,\ \mathrm{z}$$

and the following pointer assignments:

$$p{=}\&x,\ p{\rightarrow}f{=}\&z,\ tt{=}p$$

in the program. Figure 5.5(a) is the initial graph, where each node represents an equivalence class consisting of one name and each edge represents the prefix relation between names (e.g., there is an edge from the node for p to the node for $*p$, which is annotated with $*$). In Figure 5.5(b), we consider the pointer assignment: $p{=}\&x$. The dotted circles in the graph indicate unions of equivalence classes. For example, the equivalence classes for p and $\&x$ are unioned; since both classes have a outgoing edge annotated with $*$, the target classes of the two edges (the equivalence classes for $*p$ and x) are unioned too. Similarly, the classes for $p{\rightarrow}f$ and $x.f$ are unioned; the classes for $p{\rightarrow}g$ and $x.g$ are unioned. In Figure 5.5(c), the pointer assignment: $p{\rightarrow}f{=}\&z$ is considered; in Figure 5.5(d), the assignment: $tt{=}p$ is considered.

5.3.3 Complexity

We assume the program is well-typed (i.e., there is no casting except in calls to memory allocation routines). So all object names in an equivalence class will have the same type. We also assume that the maximum number of members for any structure type is a small constant compared to the number of object names in the program. By this assumption, the size of the PREFIX relation for any equivalence class is a small constant.

Let N_0 be number of object names used in the program, that is, $N_0 = \mid B_0 \mid$.

Given a program, the set B_0 is computed by going through each statement in the intermediate representation of the program and examining object names appearing in

$struct\ s_1\ \{\ int\ i\ ,\ *p;\ \};$

$struct\ s_2\ \{\ struct\ s_1\quad f_{21}\ ,\ f_{22}\ \};$

$struct\ s_3\ \{\ struct\ s_2\quad f_{31}\ ,\ f_{32}\ \};$

...

$struct\ s_n\ \{\ struct\ s_{n-1}\quad f_{n,1}\ ,\ f_{n,2}\ \}$ x;

Figure 5.6: Exponential Number of Object Names

the statement. This takes time linear in the size of the intermediate representation and the number of object names in B_0.

Phase 1 of the calculation of the PE relation will take $\mathcal{O}(N_0)$ time. The cost of Phase 2 is dominated by calls to routine MERGE(). Below we will estimate the number of calls to UNION() and FIND() in Phase 2. Besides the recursive calls to itself, each call of MERGE() will incur one call to UNION(), a constant number of calls to FIND() by our assumption that the number of members of any structure type is a small constant, and a constant cost for other operations in MERGE(). Since each call to UNION() will reduce the number of equivalence classes by one and there are initially N_0 classes, there are no more than N_0 calls to UNION() in Phase 2. Therefore there are no more than N_0 calls to MERGE() and the number of calls to FIND() will be $\mathcal{O}(N_0)$. If we use a fast union/find algorithm such as the one in [Tar83], the cost of all calls to UNION() and FIND() in Phase 2 is $\mathcal{O}(N_0 \times \alpha(N_0,N_0))$, where α is the inverse of the Ackermann's function; the cost of calls to MERGE() in Phase 2 is $(\mathcal{O}(N_0 \times \alpha(N_0,N_0)) + \mathcal{O}(N_0))$. The complexity of the algorithm is $\mathcal{O}(N_0 \times \alpha(N_0,N_0))$.

Note that the number of object names used in a program could be exponential in the maximum nesting depth of structures in the program. For instance, the program segment in Figure 5.6 will cause exponential number of object names to be created. We think this worst case rarely, if ever, happens in real programs.

5.3.4 Program Decomposition

The result of the algorithm in Figure 5.4 can be thought as a labeled, directed multi-graph, where nodes are equivalence classes and edges are represented by tuples in the PREFIX relations for equivalence classes; specifically, if there is a tuple (a,o) in PREFIX(e), where a is an accessor, o is an object name and e is an equivalence class, then there is an edge from the node for e to the node for FIND(o), labeled with the accessor a. We call the graph G_{PE}. In Figure 5.7, we show the G_{PE} for the example program, where each node is annotated with the set of object names in the equivalence class that it represents.

Since the PE relation is type consistent, object names in each equivalence class of the relation have the same type. In G_{PE}, a node for an equivalence class with object names of a pointer type may [4] have an outgoing edge annotated with $*$; a node for an equivalence class with object names of a structure type has a number of outgoing edges, each of which is annotated with a member name of the structure type.

G_{PE} can be decomposed into weakly connected components. For instance, the G_{PE} for the example program shown in Figure 5.7 has two weakly connected components. There are two sets that are unique to each component:

- a set of pointer-related assignments, which is the union of the assignments for the nodes in the component, and
- a set of object names, which is the union of the object names for the nodes in the component.

The pointer-related assignments for each weakly connected component will only create run-time aliases that involve variable names and heap names in the set of object names affiliated with the component. In another words, the sets of pointer-related assignments for weakly connected components in G_{PE}, are independent of each other in terms of their aliasing effects. Therefore, they constitute a program decomposition for

[4] There may not be such an edge. For example, assume that a variable name v in a program is of pointer type and is in an equivalence class by itself. If the name $*v$ is never used in the program and $*v$ is not aliased to any location name, then there is no outgoing from the node for v, annotated with $*$.

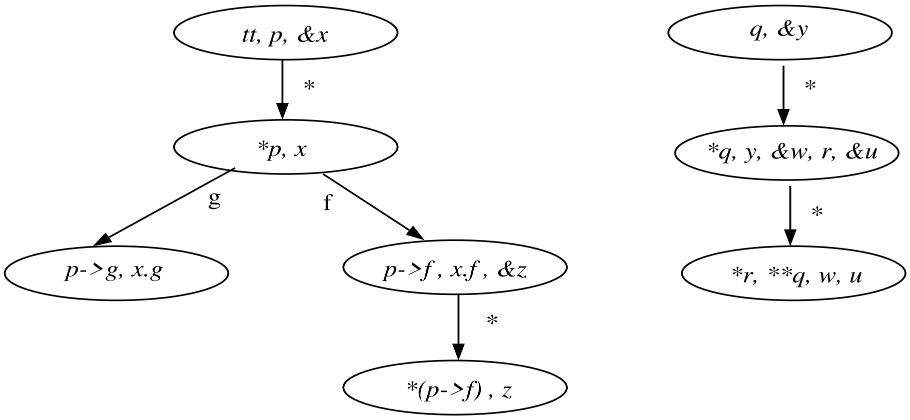


Figure 5.7: G_{PE} for the Example Program

pointer aliasing analysis. To formally show that these weakly connected components are a decomposition for pointer aliasing analysis, we will define a new relation called FA in next section. The relation can be calculated separately for each weakly connected component of G_{PE}, and can be proved to be a safe estimate of the run-time aliases.

Within a weakly connected component, if there is a path from a node n_1 to a node n_2, then the assignment for n_1 may affect aliasing of names for n_2; if there is no path between the two nodes, then assignments for the two do not affect each other in term of pointer aliasing. For example, in Figure 5.7, to determine aliases for the name $p \rightarrow g$, we only need to consider two assignments: $p = \&x$ and $tt = p$. This allows the analysis of some (instead of *all*) of the pointer-related assignments associated with a weakly connected component, which may be useful for some applications (e.g., call graph construction for programs with indirect calls through function pointers).

5.4 The FA Relation

5.4.1 Definition of the FA Relation

The following are the notations that will be used in this section.

- B_1 is the subset of B_0 that excludes any object name with &, that is,

$$B_1 = B_0 - \{\ o \mid o \in B_0 \text{ and } o \text{ is of the form } \&o_1\ \}$$

  B_1 is closed with respect to prefixes.

- $B_{\mathrm{PE}}(n)$ is the set of object names associated with a node n in G_{PE}.

  We have $B_{\mathrm{PE}}(n) \subseteq B_0$. Any object name in B_1 is in some $B_{\mathrm{PE}}(n)$, where n is a node in G_{PE}.

- $(B_{\mathrm{PE}}(n) \cap B_1)$ is the is the set of object names associated with a node n in G_{PE}, which do not contain &.

We assume that a path in G_{PE} consisting of only one node is annotated with an empty sequence of accessors, ϵ. First, We define a set of object names based on paths in G_{PE}.

Definition 5.4.1.1 Given the G_{PE} for a program, B is the set of object names defined below:

$$B = \left\{ \ o \ \middle| \ \begin{array}{l} \text{there is a path from a node } n \text{ to a node } m \text{ in } G_{\mathrm{PE}} \text{ such that the} \\ \text{path is annotated with a sequence of accessors } a_1a_2...a_j \ (j \geq 0) \\ \text{and } o = \mathit{apply}^*(o_1, a_1a_2...a_j), \text{ where } o_1 \in (B_{\mathrm{PE}}(n) \cap B_1) \end{array} \right\}$$

□

By definition, B does not contain any object names with & and B is closed with respect to prefixes. Since paths consisting of one node are annotated with ϵ, any object name in $(B_{\mathrm{PE}}(n) \cap B_1)$, where n is a node in G_{PE}, is in B. In another words, $B_1 \subseteq B$.

For the example program in Figure 5.2, we have:

$$B = \left\{ \begin{array}{l} p,\ *p,\ p{\rightarrow}f,\ *(p{\rightarrow}f),\ p{\rightarrow}g,\ tt,\ *tt,\ tt{\rightarrow}f,\ *(tt{\rightarrow}f),\ tt{\rightarrow}g, \\ x,\ x.f,\ *(x.f),\ x.g,\ q,\ *q,\ **q,\ y,\ *y,\ r,\ *r,\ z,\ w,\ u \end{array} \right\}$$

Note that some of the names (e.g., $*(tt{\rightarrow}f)$, $*y$) in B above are not in the set B_0 for the example program.

By definition, for each object name $o \in B$, there is a path from a node n to a node m in G_{PE} such that the path is annotated with a sequence of accessors A and o = $apply^{*}(o_1,A)$, where $o_1 \in (B_{\mathrm{PE}}(n) \cap B_1)$. There may be more than one such path in G_{PE}. It is easy to show that all these paths for o will end at the node m in G_{PE}. Here is a sketch of the proof. Suppose there is a path from n_1 annotated with A_1 such that o = $apply^{*}(o_1,A_1)$, where $o_1 \in (B_{\mathrm{PE}}(n_1) \cap B_1)$ and there is another path from n_2 annotated with A_2 such that $o = apply^{*}(o_2,A_2)$, where $o_2 \in (B_{\mathrm{PE}}(n_2) \cap B_1)$. Since $apply^{*}(o_1,A_1)$ = $apply^{*}(o_2,A_2)$, without loss of generality, we assume $A_1 = A_2^{'}.A_2$; in another words, $o_2 = apply^{*}(o_1,A_2^{'})$. By the algorithm in Figure 5.4, there is a path in G_{PE} from n_1 to n_2 such that the path is annotated with $A_2^{'}$. Therefore, the two paths for o will end at the same node.

Next we define a relation on B.

Definition 5.4.1.2 Given the graph G_{PE} for a program, let R_1 be a relation on B defined below:

$$\mathrm{R}_1 = \left\{ (o_1^{'},o_2^{'}) \;\middle|\; \begin{array}{l} \text{there is a path from a node } n \text{ to a node } m \text{ in } G_{\mathrm{PE}} \text{ such that} \\ \text{the path is annotated with a sequence of accessors } a_1a_2...a_j, \\ \text{NumOfDerefs}(a_1a_2...a_j) \geq 1,\ o_1^{'} = apply^{*}(o_1,a_1a_2...a_j) \text{ and} \\ o_2^{'} = apply^{*}(o_2,a_1a_2...a_j), \text{ where } o_1 \in B_{\mathrm{PE}}(n) \text{ and } o_2 \in B_{\mathrm{PE}}(n) \end{array} \right\}$$

We call $\mathrm{R}_1{}^{e}$ the FA (Flow-insensitive Alias) relation.

□

By definition, for each $(o_1^{'},o_2^{'}) \in \mathrm{R}_1$, there is a path from a node n in G_{PE} such that the path is annotated with a sequence of accessors A, NumOfDerefs$(A) \geq 1$, $o_1^{'}$ = $apply^{*}(o_1,A)$ and $o_2^{'} = apply^{*}(o_2,A)$, where $o_1 \in B_{\mathrm{PE}}(n)$ and $o_2 \in B_{\mathrm{PE}}(n)$. If o_1 does not contain &, then $o_1^{'} \in B$. If o_1 does contain &, by the algorithm in Figure 5.4, node n has an outgoing edge labeled with $*$ in G_{PE}; since NumOfDerefs$(A) \geq 1$, $o_1^{'} \in B$. Similarly, $o_2^{'} \in B$. Therefore, R_1 defines a relation on B.

Intuitively, a tuple $(o_1^{'},o_2^{'})$ is in the FA relation if the two object names are aliased, assuming pointer-related assignments are considered symmetric and the control flow in the program is not taken into account.

{ p }
{ tt }
{ $*p$, $*tt$, x }
{ $p{\rightarrow}f$, $tt{\rightarrow}f$, $x.f$ }
{ $p{\rightarrow}g$, $tt{\rightarrow}g$, $x.g$ }
{ $*(p{\rightarrow}f)$, $*(tt{\rightarrow}f)$, $*(x.f)$, z }

{ q }
{ $*q$, y }
{ r }
{ $**q$, $*y$, $*r$, w, u }

Figure 5.8: The Equivalence Classes of the FA relation for the Example Program

For the example program in Figure 5.2, the FA relation has ten equivalence classes shown in Figure 5.8.

Lemma 5.4.1.1 R_1 is a weakly right-regular relation on B.

□

Let $(o_1^{'},o_2^{'})$ be a tuple in R_1. By definition, there is a path from a node n to a node m in G_{PE} such that the path is annotated with a sequence of accessors A, NumOfDerefs(A) ≥ 1, $o_1^{'} = apply^*(o_1,A)$ and $o_2^{'} = apply^*(o_2,A)$, where $o_1 \in B_{\mathrm{PE}}(n)$ and $o_2 \in B_{\mathrm{PE}}(n)$.

Suppose the object name $apply(o_1^{'},a) \in B$, where a is an accessor. By definition, there is a path for G_{PE} for $apply(o_1^{'},a)$. Since all paths for $o_1^{'}$ in G_{PE} end at node m, there must be an outgoing edge from node m such that the edge is annotated with a. Therefore, $apply(o_2^{'},a) \in B$; by definition, $(apply(o_1^{'},a),apply(o_2^{'},a)) \in R_1$.

By the above argument, R_1 is a weakly right-regular relation on B.

□

Lemma 5.4.1.2 The FA relation is a weakly right-regular equivalence relation on B.

□

Proof by induction on the calculation of the reflexive, symmetric and transitive closure of R_1.

□

By definition, the FA relation can be partitioned according to weakly connected components of G_{PE}, that is, there is a subrelation of the FA relation for each component, which is independent of other components in G_{PE}. For instance, for the example program in Figure 5.2, the FA relation can be partitioned into two subrelations.

We prove in Appendix B that the FA relation for a program contains the run-time aliases at any program point on an execution path of the program. The subrelation of the FA relation for each weakly connected component of G_{PE}, is a safe estimate of the run-time aliases that can be induced by the pointer-related assignments associated with the component. Therefore, sets of pointer-related assignments affiliated with weakly connected components in G_{PE} form a program decomposition for pointer aliasing analysis.

5.4.2 Calculation of the FA Relation

Since the number of object names in B may be infinite if there is a cycle in G_{PE}, we can not always directly calculate the set B. However, by definition of B, names in B may be represented as paths in graphs, and graphs with cycles can represent infinite number of object names. As a matter of fact, G_{PE} is such a graph, where each name in B is represented by one or more paths in the graph.

So the idea is to use a directed graph to represent the FA relation. The graph will have a structure similar to G_{PE}, that is, nodes are annotated with a set of object names; edges of the graph are annotated with either $*$ or a member name of a structure type. We would like the following to be true:

(1) Each name in B is represented by one or more paths in the graph.

(2) Each path in the graph represents a set of object names in B.

(3) $(o_1,o_2) \in$ FA *if and only if* the path for o_1 and the path for o_2 end at the same node in the graph.

Given G_{PE}, the calculation of the FA relation, shown in Figure 5.9, is tantamount

```
calculate-FA-relation()
{
   for each o ∈ B_1
   {
      INIT-EQUIV-CLASS(o)
      PREFIX(FIND(o)) = { }
   }
   /* Phase 1 */
   for each edge in G_PE from n to m annotated with accessor member
   {
      for each o ∈ B_PE(n)
         add (member, apply(o,member)) to PREFIX(FIND(o))
   }
   /* Phase 2 */
   for each edge in G_PE from n to m annotated with accessor *
   {
      obj-set = { o_1 | o_1 ∈ B_PE(m) and o_1 = apply(o, * ), where o ∈ B_PE(n) }
      let o_1 be an arbitrary object name in obj-set
      for each o_2 ∈ obj-set
         if (FIND(o_1) ≠ FIND(o_2))
           MERGE(FIND(o_1),FIND(o_2))
      for each o ∈ (B_PE(n) ∩ B_1)
         if there is not a ( * , o_2) in PREFIX(FIND(o)) such that FIND(o_2) == FIND(o_1)
           add ( * , o_1) to PREFIX(FIND(o))
   }
}
```

Figure 5.9: Calculation of the FA Relation

to the construction of such a graph. The algorithm represents nodes by equivalence classes and edges by tuples in the PREFIX relations of equivalence classes.

Initially, there is one node for each object name in B_1 and there is no edge.

In Phase 1, each edge in G_{PE} annotated with a member name *member*, is examined. Assume the edge is coming out of node n in G_{PE}. For each object name $o \in (B_{\mathrm{PE}}(n) \cap B_1)$ (i.e., excluding object names with &), an edge is added from the node for FIND(o) to the node for FIND(*apply*(o,*member*)). Note that by the definition of the set B_0, for any object name o of structure type and any member *member* of the type, if $o \in B_0$, then *apply*(o,*member*) $\in B_0$.

As an example, consider G_{PE} in Figure 5.7. After the edges labeled with f and g are examined, we have:

PREFIX(FIND($*p$)) = { (g,$p \rightarrow g$), (f,$p \rightarrow f$) }
PREFIX(FIND(x)) = { (g,$x.g$), (f,$x.f$) }

In other words, there are two outgoing edges from the node for FIND($*p$), one to the node for FIND($p \rightarrow g$) and another to the node for FIND($p \rightarrow f$). Similarly there are two edges from the node for FIND(x).

In Phase 2, each edge in G_{PE} annotated with $*$, is examined. Assume the edge is from node n to node m in G_{PE}. First, all the object names $o_1 \in B_{\mathrm{PE}}(m)$ such that o_1 = *apply*(o, $*$) for some $o \in B_{\mathrm{PE}}(n)$, are collected; because of the presence of the edge in G_{PE}, there is at least one such object name. Then, nodes for equivalence classes containing these object names are unioned, that is, these names will be in the same equivalence class. Other nodes may also be unioned if they can be reached through the same sequence of accessors from the nodes for these names; the MERGE() routine takes care of all these. After the necessary unions, there is a node representing the equivalence class containing all the object names collected; let us call it node x. For each object name $o \in (B_{\mathrm{PE}}(n) \cap B_1)$, an edge is added from the node for FIND(o) to the node x.

Again, consider G_{PE} in Figure 5.7. When examining the edge from the equivalence class with object name, p, tt and &x, two names, $*p$ and x, will be collected. The equivalence classes, FIND($*p$) and FIND(x), are merged. As a result of the merge, the equivalence classes, FIND($p \rightarrow g$) and FIND($x.g$), are merged; so are FIND($p \rightarrow f$) and FIND($x.f$). Finally, edges labeled with $*$ are added for the equivalence classes, FIND(p) and FIND(tt); their PREFIX sets are:

PREFIX(FIND(p)) = { ($*$,$*p$) }
PREFIX(FIND(tt)) = { ($*$,$*p$) }

The calculation of the FA relation can be thought as a graph algorithm, in which equivalence classes are nodes of a graph and tuples in the PREFIX relations of equivalence classes are edges of the graph. In Figure 5.10, we show the intermediate graphs during calculation of the FA relation from G_{PE} in Figure 5.7. To make it simple, we only consider the following names:

$$tt,\ p,\ *p,\ p \rightarrow f,\ *(p \rightarrow f),\ p \rightarrow g,\ \&x,\ x,\ x.f,\ x.g,\ \&z,\ \mathrm{z}$$

and the edges related to these names in G_{PE}. Figure 5.10(a) is the initial graph, where each node represents an equivalene class consisting of one name and there is no edge. Note that names with & are not included in the graph. In Figure 5.10(b), 5.10(c), and 5.10(d), we examine edges in G_{PE} related to the names being considered here; the edges in the dotted squares in each figure are from G_{PE}. We first examine the two edges annotated with the structure members g and f. Basically, we add two edges annotated with g and two annotated with f to the graph, as shown in Figure 5.10(b). For example, one is added from the node for $*p$ to the node for $p \rightarrow f$ and one is added from the node for $*p$ to the node for $p \rightarrow g$. We then examine the edges annotated with $*$. In Figure 5.10(c), we consider the edge from the equivalence class

$$\{\ p,\ tt,\ \&x\ \}$$

to the equivalence class

$$\{\ *p,\ x\ \}$$

Since $*p$ can be derived from p and x can be derived from $\&x$ by applying $*$ respectively, the equivalence classes for $*p$ and x are unioned. As both classes have a outgoing edge

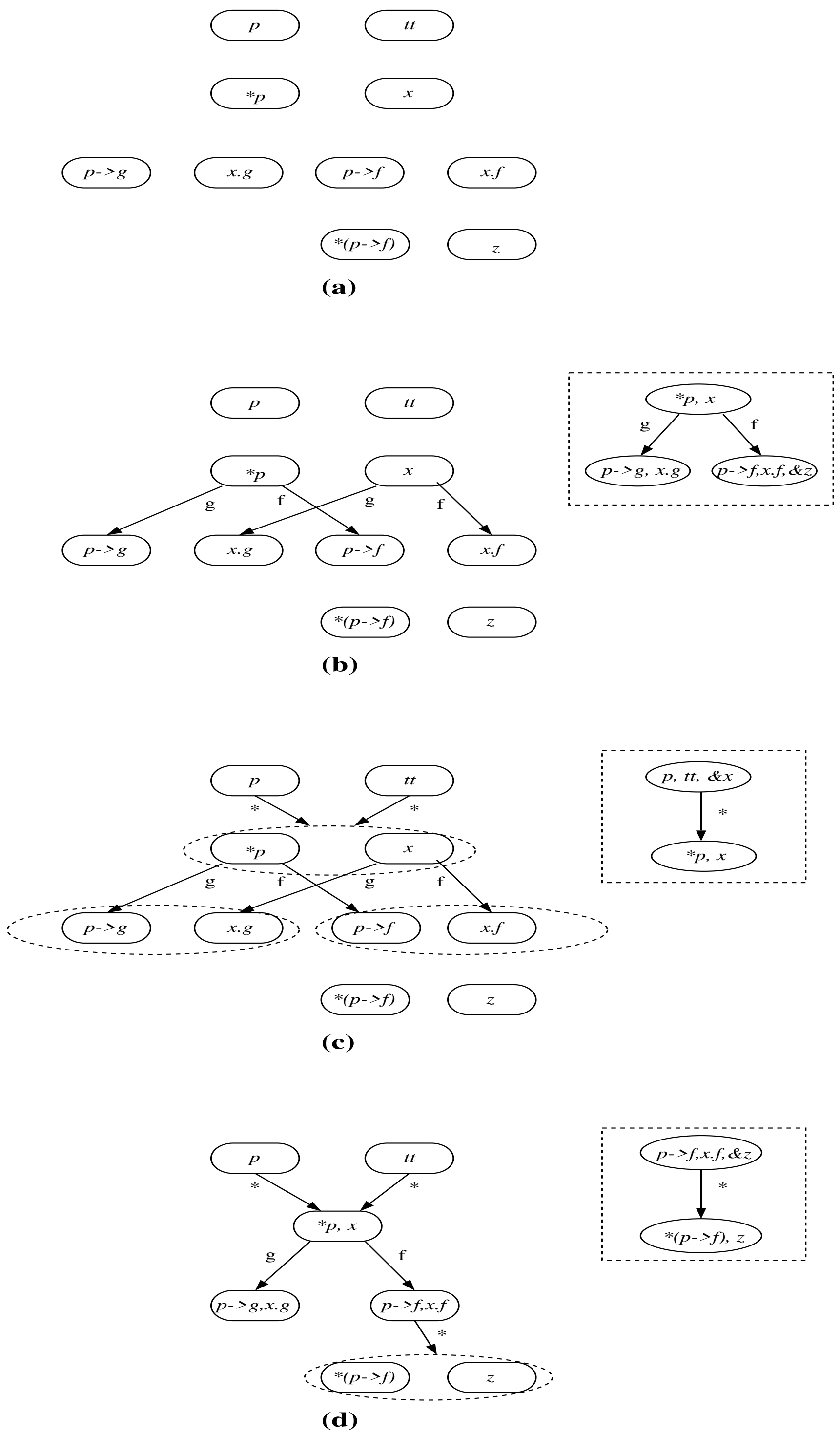


Figure 5.10: Intermediate Graphs in Calculating FA Relation

annotated with g, the target classes of the two edges (the equivalence class for $p{\rightarrow}g$ and $x.g$) are unioned too. Similarly, the classes for $p{\rightarrow}f$ and $x.f$ are unioned. The dotted circles in the graph indicate unions of equivalence classes. In Figure 5.10(d), we examine the edge from equivalence class

$$\{\ p{\rightarrow}f,\ x.f,\ \&z\ \}$$

to the equivalence class

$$\{\ *(p{\rightarrow}f),\ z\ \}$$

The equivalence classes for $*(p{\rightarrow}f)$ and z are unioned.

The result of the calculation, is a labeled, directed multi-graph such that

- Each node in the graph represents an equivalence class; the node is annotated with a set of object names (a subset of B_1) that are in the equivalence class.

  We will use $B_{\mathrm{FA}}(x)$ for the set of object names for a node x in G_{FA}; $B_{\mathrm{FA}}(x) \subseteq B_1$.

- Each edge in the graph from the node for equivalence class e_1 to the node for equivalence class e_2, corresponds to a tuple (a,o) in PREFIX(e_1), where a is an accessor, o is an object name and e_2 = FIND(o); the edge is annotated with the accessor a.

We call the graph G_{FA}. In Figure 5.11, we show the G_{FA} for the example program in Figure 5.2. The next few lemmas show that G_{FA} is the graph we want. We provide their proofs in Appendix A.

Lemma 5.4.1 For any object name o_1 in B, there is a path from a node x in G_{FA} such that the path is annotated with a sequence of accessors $A = a_1a_2...a_j$ $(j \geq 0)$ and $o_1 =$ *apply*$^*(o,A)$, where $o \in B_{\mathrm{FA}}(x)$.

□

This lemma corresponds to item (1) on page 107.

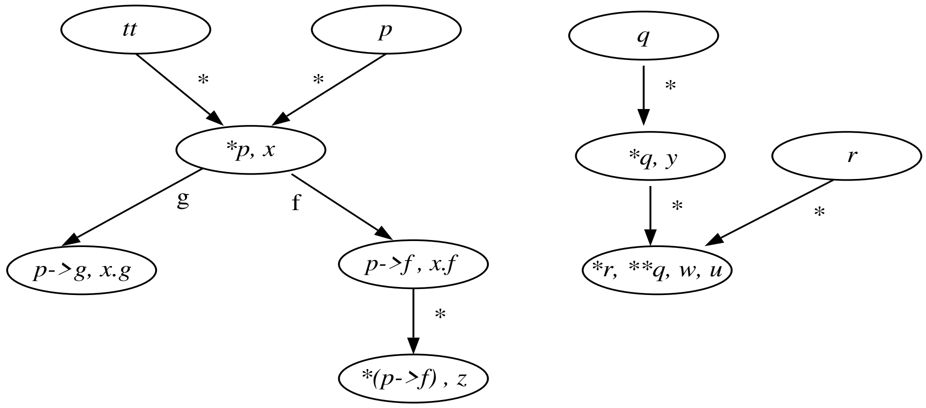


Figure 5.11: G_{FA} for the Example Program

Lemma 5.4.2 For each path from a node x in G_{FA} such that the path is annotated with a sequence of accessors $A = a_1a_2...a_j$ $(j \geq 0)$, and for any object name $o \in B_{\mathrm{FA}}(x)$, the object name $apply^*(o,A)$ is in B.

□

This lemma corresponds to item (2) on page 107.

Lemma 5.4.3 For each node x in G_{FA}, we define the following set of object names:

$$O_{\mathrm{FA}}(x) = \left\{ o \;\middle|\; \begin{array}{l} \text{there is a path from a node } y \text{ to a node } x \text{ in } G_{\mathrm{FA}} \text{ such that the} \\ \text{path is annotated with a sequence of accessors } a_1a_2...a_j \ (j \geq 0) \\ \text{and } o = apply^*(o_1,a_1a_2...a_j), \text{ where } o_1 \in B_{\mathrm{FA}}(y) \end{array} \right\}$$

These sets constitute a partition of the set B and the equivalence relation induced by these sets is the FA relation.

□

Object names in $O_{\mathrm{FA}}(x)$, where x is a node in G_{FA}, are represented by paths in G_{FA} ending at node x. By this lemma, $(o_1,o_2) \in \mathrm{FA}$ *if and only if* $(o_1,o_2) \in O_{\mathrm{FA}}(x)$, where x is a node in G_{FA}. So this lemma corresponds to item (3) on page 107.

$\{\ p\ \}$
$\{\ tt\ \}$
$\{\ *p,\ x\ \}$
$\{\ p{\rightarrow}f,\ x.f\ \}$
$\{\ p{\rightarrow}g,\ x.g\ \}$
$\{\ *(p{\rightarrow}f),\ z\ \}$

$\{\ q\ \}$
$\{\ *q,\ y\ \}$
$\{\ r\ \}$
$\{\ **q,\ *r,\ w,\ u\ \}$

Figure 5.12: Equivalence Classes of the Partial FA Relation for the Example Program

We assume a path in G_{FA} consisting of one node is annotated with ϵ; so by definition of $O_{\mathrm{FA}}(x)$, $B_{\mathrm{FA}}(x) \subseteq O_{\mathrm{FA}}(x)$, for any node x in G_{FA}.

Definition 5.4.2.1 For each node x in G_{FA}, $B_{\mathrm{FA}}(x)$ is the set of object names associated with x. These sets constitute a partition of the set B_1. We call the equivalence relation induced by these sets *the partial FA relation.*

□

The partial FA relation is the projection of the FA relation on the set B_1. Since the FA relation for a program is a safe estimate of the run-time aliases for the program, we can use the partial FA relation as safe alias relation involving *only* object names in B_1. Since we have all location names of the program in B_1, the partial FA relation contains enough information about the location names to which any object name in B_1 with pointer dereferences may be aliased. We will make use of this in solving the Thru-deref MOD/REF problems. The equivalence classes of the partial FA relation for the example program are shown in Figure 5.12.

It is worth noting that in the full FA relation for the program shown in Section 5.4.1, we have an equivalence class $\{\ *(p{\rightarrow}f),\ *(tt{\rightarrow}f),\ *(x.f),\ z\ \}$. Here the corresponding equivalence class is $\{\ *(p{\rightarrow}f),\ \mathrm{z}\ \}$. This is because the object names, $*(tt{\rightarrow}f)$ and $*(x.f)$, are not used explicitly in the program. Aliases involving any of these names may be necessary for some applications (e.g., shape analysis) and may not be necessary for others (e.g., Thru-deref MOD/REF problems).

The partial FA relation can be partitioned according to weakly connected components of G_{PE}. The subrelation of the partial FA relation for a weakly connected component in G_{PE}, can be calculated by an algorithm similar to the one shown in Figure 5.9, which starts with *only* object names for that component (excluding names with &) and examines *only* edges in the component; this subrelation can be used as a safe alias relation for object names associated with the component. The partial FA relation for the example program, can be partitioned into two subrelations.

We call the calculation of the G_{FA} and the partial FA relation for aliasing information the FA analysis.

5.4.3 Complexity

By a similar argument as the one in Section 5.3.3, the complexity of the algorithm in Figure 5.9 is $\mathcal{O}(N_1 \times \alpha(N_1, N_1))$, where N_1 is the number of object names in B_1, that is, $N_1 = \mid B_1 \mid$.

Chapter 6

Program Decomposition and Flow-insensitive Aliasing Analysis with Unions and Type Casting

In this chapter, we extend the ideas in Chapter 5 to programs with indirect calls through function pointers, unions, and type casting. This chapter is organized as follows. In Section 6.1, we give the intermediate representation for programs with indirect calls and type casting. In Section 6.2, we extend the definitions in Section 5.2 for object names and define interesting equivalence relations on typed object names. In Section 6.3, we present our approach to dealing with indirect calls, unions, and type casting in partitioning assignments in a program. In Section 6.4, we present the FA analysis based on the result of program decomposition. In Section 6.5, we show some empirical results. Finally, we give the results of the program decomposition and the FA analysis for an example program in Section 6.6.

6.1 Program Representation

We represent a program in an intermediate form, whose syntax is given in Figure 6.1. Basically a program consists of a number of procedures. One of these procedures is *main*() and another is a special procedure *_init_*(), which initializes all global variables and calls *main*(). A procedure is a sequence of statements; the first one has to be the entry and the last the exit. Call and return statements are used to represent procedure calls; indirect calls are through function pointer variables. Special variables (e.g., *malloc*()) are used to represent values returned by functions (e.g., *malloc*). There are a number of kinds of assignment statements including non-pointer assignment with a basic arithmetic or relational operation, several kinds of pointer assignments, structure or union assignment with both sides being of the same structure or union types. Type casting is allowed for actual parameters in calls or pointers as the right hand side of pointer assignments; pointer arithmetic is allowed in pointer assignments. Other statements allowed are conditional goto and goto. Statements have IDs associated with them, where an ID is in the range of 1..(# of statements). Conditional and goto statements use statement IDs for their destinations. This is a quadruple representation and it can be depicted in a graphical form such as the ICFG[LR92].

We will have the same restrictions on input programs as those mentioned in Section 4.4.4. Some of the restrictions are reflected in the intermediate representation; for example, a C program with type casting between a pointer and an integer, can not be translated into the representation.

Definition 6.1.1 A procedure is said to be *reachable* if there is a chain of direct or indirect calls from *main* to it.

□

Definition 6.1.2 A pointer-related assignment in a reachable procedure is one of the following:

- a pointer assignment
- a structure or union assignment such that the structure or union type contains pointer members
- an actual at a direct call statement and the corresponding formal of the procedure being called such that the two are either of pointer type or of structure or union type with pointer members
- an actual at an indirect call statement and the corresponding formal of the procedure *found* to be called such that the two are either of pointer type or of structure or union type with pointer members

□

Program ::=	(Procedure)$^+$	
Procedure ::=	(L: Statement)$^+$	(labeled statements)
PossibleCast ::=	ϵ	
	(PointerType)	(type casting)
Statement ::=	entry of P(FmlName$_1$,...,FmlName$_m$)	(procedure entry)
	| exit of P	(procedure exit)
	| call P(PossibleCast ArgName$_1$,..., PossibleCast ArgName$_m$)	(direct call)
	| call fp(PossibleCast ArgName$_1$,..., PossibleCast ArgName$_m$)	(call through a function pointer)
	| return	(return)
	| PrmName = Exp	(non pointer assignment)
	| PtrName = NULL	(pointer assignment)
	| PtrName = PossibleCast &Name	(pointer assignment)
	| PtrName = PossibleCast PtrName$_1$	(pointer assignment)
	| PtrName = PtrName$_1$ PlusOrMinusOp IntName	(pointer arithmetic)
	| StrtName = StrtName$_1$	(structure or union assignment)
	| if (PrmName) (goto L$_1$) (goto L$_2$)	(conditional goto)
	| goto L	(goto statement)

Exp ::=	Constant	(constant)
	| PrmName	(object name of primitive types)
	| Exp$_1$ Op Exp$_2$	(arithmetic or relational operation)

FmlName$_1$, ..., FmlName$_m$: formal variable names
IntName: object names or constants of integer type
PrmName, PrmName$_1$, PrmName$_2$: object names or constants of primitive types
PtrName, PtrName$_1$: object names of pointer types
StrtName, StrtName$_1$: object names of structure or union types
Name, ArgName$_1$, ..., ArgName$_m$: object names or constants of any type
L, L$_1$, L$_2$: IDs for statement
PlusOrMinusOp: addition or subtraction operator
Op: primitive operators (e.g., arithmetic, relational)
NULL: nil pointer

Figure 6.1: Intermediate Representation

6.2 Relations on Sets of Typed Object Names

We define sets of typed object names and relations on these sets.

Definition 6.2.1 A set of typed object names, S, is *closed with respect to prefixes* if the following are true:

- If $<\&o,\ t\ *> \in$ S, then $<o,\ t> \in$ S.
- If $<o.m,\ t> \in$ S and the type of the structure or union containing m is t_1, then $<o,\ t_1> \in$ S.
- If $<*o,\ t> \in$ S, then $<o,\ t\ *> \in$ S.
- If $<o[\ ],\ t> \in$ S, then $<o,\ t\ [\ ]> \in$ S.

□

Intuitively, if S is closed with respect to prefixes, then for any name in S, all its prefixes should also be in S.

Assume we have the following declaration:

$$struct\ foo\ \{\ int\ a;\ float\ *b;\ float\ *c;\ \};$$

The set $\{\ <p{\rightarrow}c,\ float\ *>,\ <\&y,\ float\ *>\ \}$ is not closed with respect to prefixes because prefixes of $p{\rightarrow}c$ such as p and $*p$ are not in the set. The following set is closed with respect to prefixes.

$$\left\{\begin{array}{l} <p,\ struct\ foo\ *>,\ <*p,\ struct\ foo>,\ <p{\rightarrow}c,\ float\ *>, \\ <\&y,\ float\ *>,\ <y,\ float> \end{array}\right\}$$

Definition 6.2.2 Given a set of reachable procedures in a program, P^0, a set of typed object names B^0, which is closed with respect to prefixes, can be defined for P^0 inductively as follows:

- If an object name o syntactically appears anywhere in a reachable procedure in P^0, then $<o,\ t> \in B^0$, where t is the declared type of o.

- If $<l,\ t> \in B^0$, where l is a location name and t is either a structure or a union type, then $<l.m,\ member\text{-}type(t\ ,\ m)> \in B^0$, where m is any member of the structure or union.
- [1]If $<l,\ t\ [\ ]> \in B^0$, where l is a location name, then $<l[\ ],\ t> \in B^0$.
- If $<\&o,\ t\ *> \in B^0$, then $<o,\ t> \in B^0$.
- If $<o.m,\ member\text{-}type(t\ ,\ m)> \in B^0$, then $<o,\ t> \in B^0$.
- If $<o[\ ],\ elem\text{-}type(t)> \in B^0$, then $<o,\ t> \in B^0$.
- If $<*o,\ t> \in B^0$, then $<o,\ t\ *> \in B^0$.

□

Basically, the set B^0 contains all names in reachable procedures in P^0 and their prefixes; these names are used according to their declared types. It also has all location names in reachable procedures in P^0 that can be accessed as their declared types (except heap names).

Definition 6.2.3 A relation R on a set of typed object names is said to be *type consistent* if one of the following is true for any $(<o_1,\ t_1>,<o_2,\ t_2>) \in$ R.

- t_1 and t_2 are the same type.
- t_1 and t_2 are both pointer types.
- t_1 and t_2 are both array types.
- t_1 is a pointer type and t_2 is an array type.
- t_1 is an array type and t_2 is a pointer type.

□

---

[1]A heap name $heap_{id}$ has a declared type $char\ [\ ]$. We do not create a name $<heap_{id}[\ ],\ char>$ right away and instead create a name $<heap_{id}[\ ],\ t>$ if there is a pointer of type $t\ *$ pointing to $heap_{id}$.

A object name of an array type implicitly represents a pointer name. Thus for each tuple of two names in a type consistent relation, either the two are of the same type or the two are both pointer types.

We will be interested in type consistent relations on sets of typed object names closed with respect to prefixes.

Definition 6.2.4 Let S be a set of typed object names closed with respect to prefixes and R be a type consistent relation on S. R is a *weakly right-regular* relation on S if the following are true:

- If $(<o_1, t>,<o_2, t>) \in \mathrm{R}$, both $<o_1^{'}, t^{'}> = apply_{\mathrm{T}}(<o_1, t>\ ,<m, t>)$ and $<o_2^{'}, t^{'}> = apply_{\mathrm{T}}(<o_2, t>\ ,<m, t>)$ are in S, then $(<o_1^{'}, t^{'}>,<o_2^{'}, t^{'}>) \in \mathrm{R}$.
- If $(<o_1, t\,[\,]>,<o_2, t\,[\,]>) \in \mathrm{R}$, both $<o_1^{'}, t> = apply_{\mathrm{T}}(<o_1, t>\ ,<[\,], t\,[\,]>)$ and $<o_2^{'}, t> = apply_{\mathrm{T}}(<o_2, t>\ ,<[\,], t\,[\,]>)$ are in S, then $(<o_1^{'}, t>,<o_2^{'}, t>) \in \mathrm{R}$.
- If $(<o_1, t_1>,<o_2, t_2>)^2 \in \mathrm{R}$, both $<o_1^{'}, t> = apply_{\mathrm{T}}(<o_1, t_1>\ ,<*, t\ *>)$ and $<o_2^{'}, t> = apply_{\mathrm{T}}(<o_2, t_2>\ ,<*, t\ *>)$ are in S, then $(<o_1^{'}, t^{'}>,<o_2^{'}, t^{'}>) \in \mathrm{R}$.

□

Let S be the following set:

$$\left\{\begin{array}{l} <p,\ struct\ foo\ *>,\ <*p,\ struct\ foo>, \\ <p{\rightarrow}a,\ int>,\ <p{\rightarrow}b,\ float\ *>,\ <p{\rightarrow}c,\ float\ *>, \\ <\&s,\ struct\ foo\ *>,\ <s,\ struct\ foo>, \\ <s.a,\ int>,\ <s.b,\ float\ *>,\ <s.c,\ float\ *> \end{array}\right\}$$

The relation $\{\ (<p,\ struct\ foo\ *>,<\&s,\ struct\ foo\ *>)\ \}$ on S is type-consistent, but not weakly right-regular. The following relation is type-consistent and weakly right-regular.

[2] It is possible that one of $\{\ t_1,\ t_2\ \}$ is a pointer type and another is an array type.

$$\left\{\begin{array}{l} (<p,\ struct\ foo\ *>,<\&s,\ struct\ foo\ *>), \\ (<*p,\ struct\ foo>,<s,\ struct\ foo>), \\ (<p{\rightarrow}a,\ int>,<s.a,\ int>), \\ (<p{\rightarrow}b,\ float\ *>,<s.b,\ float\ *>), \\ (<p{\rightarrow}c,\ float\ *>,<s.c,\ float\ *>) \end{array}\right\}$$

Lemma 6.2.1 Let S be a set of typed object names closed with respect to prefix. Let R be a type-consistent and weakly right-regular relation on S. If there are a tuple $(tobj_1,tobj_2) \in$ R and a sequence of accessors A = $<a_1,\ t_1>$ $<a_2,\ t_2>$ ... $<a_n,\ t_n>$ ($n \geq 1$) such that both $tobj_1^{'} = apply^*_{\mathrm{T}}(tobj_1\ ,\ A)$ and $tobj_2^{'} = apply^*_{\mathrm{T}}(tobj_2\ ,\ A)$ are in S, then $(tobj_1^{'},tobj_2^{'}) \in$ R.

□

Proof by induction on the number of accessors n.

Definition 6.2.5 Let S be a set of typed object names closed with respect to prefixes and R be a type-consistent relation on S. R^e is the smallest[3] equivalence relation on S containing R.

□

In another words, R^e is the reflexive, symmetric and transitive closure of R.

Definition 6.2.6 Let S be a set of typed object names closed with respect to prefixes and R be a type-consistent relation on S. R^{wr} is the smallest weakly right-regular equivalence relation on S containing R.

□

If R is type consistent, both R^e and R^{wr} are type consistent.

Since both R^e and R^{wr} are equivalence relations, they can be represented as sets of equivalence classes.

[3]By *smallest*, we mean the least number of tuples.

Let R = { ($<p$, $struct\ foo\ *>$,$<\&s$, $struct\ foo\ *>$) } be a type-consistent relation on S. R^e consists of the following equivalence classes:

{ $<p$, $struct\ foo\ *>$, $<\&s$, $struct\ foo\ *>$ }

{ $<*p$, $struct\ foo>$ }

{ $<p\rightarrow a$, $int>$ }

{ $<p\rightarrow b$, $float\ *>$ }

{ $<p\rightarrow c$, $float\ *>$ }

{ $<s$, $struct\ foo\ *>$ }

{ $<s.a$, $int>$ }

{ $<s.b$, $float\ *>$ }

{ $<s.c$, $float\ *>$ }

And R^{wr} consists of the following equivalence classes:

{ $<p$, $struct\ foo\ *>$, $<\&s$, $struct\ foo\ *>$ }

{ $<*p$, $struct\ foo>$, $<s$, $struct\ foo\ *>$ }

{ $<p\rightarrow a$, $int>$, $<s.a$, $int>$ }

{ $<p\rightarrow b$, $float\ *>$, $<s.b$, $float\ *>$ }

{ $<p\rightarrow c$, $float\ *>$, $<s.c$, $float\ *>$ }

R^e and R^{wr} can also be represented as graphs, where nodes represents equivalence classes and edges are annotated with a direct prefix relation between typed object names in equivalence classes (e.g. $<*$, $struct\ foo\ *>$ and $<a$, $struct\ foo>$).

6.3 Program Decomposition with Unions and Type Casting

In Figure 6.2, we show the overview of our implementation of program decomposition for programs with indirect calls through function pointers, unions, and type casting. As the first step, we calculate an initial PE relation by considering only procedures reachable through direct calls. We then apply the points-to analysis algorithm in Chapter 4 to

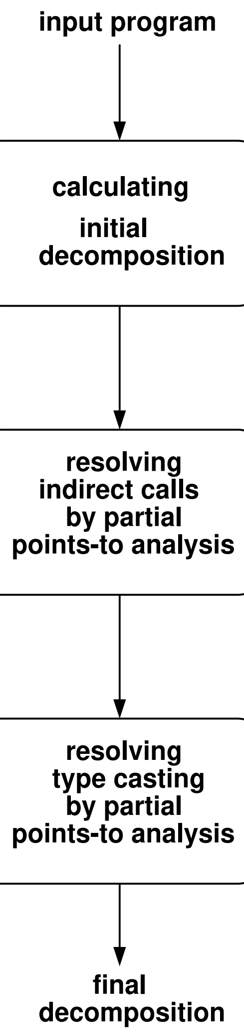


Figure 6.2: Program Decomposition with Indirect Calls, Unions, and Type Casting

program segments that may affect function pointer aliasing and resolve indirect calls through function pointers. Finally we again use the points-to analysis algorithm to determine locations being accessed by pointers involving assignments with either type casting or pointer arithmetic. The initial PE relation is incrementally updated in the two steps; we obtain a final PE relation, which implies a program decomposition and provides a basis for the flow-insensitive aliasing analysis.

6.3.1 The Initial Program Decomposition

Definition of Initial PE Relation For the initial program decomposition, we will only consider procedures that are reachable because of direct calls in a program. A simple flow analysis will be performed to determine these reachable procedures.

Let P^1 be the set of these reachable procedures. Let B^1 be the set of typed object names defined for P^1 according to Definition 6.2.2. Let R^1 be a relation on B^1 defined below:

$$\left\{ (<lhs,\, t_{lhs}>, <rhs,\, t_{rhs}>) \;\middle|\; \begin{array}{l} lhs = rhs \text{ is a pointer-related assignment in a} \\ \text{reachable procedure in } P^1,\ t_{lhs} \text{ is the declared} \\ \text{type of } lhs, \text{ and } t_{rhs} \text{ is the declared type of } rhs \end{array} \right\}$$

Let E^1 be the equivalence relation implied by access patterns of *declared* unions. The initial PE relation is $(R^1 \cup E^1)^{wr}$. The initial G_{PE} is the graph representation of the initial PE relation; it provides a decomposition of the program consisting of only reachable procedures through direct calls.

Calculation of Initial PE Relation We assume the following routines are available for initializing and maintaining equivalence classes:

- INIT-EQUIV-CLASS($tobj$), where $tobj$ is a typed object name, initializes an equivalence class with the typed object name.
- FIND($tobj$), where $tobj$ is a typed object name, returns the equivalence class for the name.
- UNION(e_1,e_2), where e_1 and e_2 are two equivalence classes, returns an equivalence class that consists of the typed object names in both e_1 and e_2.

```
add-a-typed-obj-name(<o, t>)
{
    if (the name <o, t> already exists)
        return

    INIT-EQUIV-CLASS(<o, t>)
    PREFIX(FIND(<o, t>) = { }

    if (o == &o1)
    { /* case 1 */
        let t be t1 *
        add-a-typed-obj-name(<o1, t1>
        add (< * , t1 * > , <o1, t1>) to PREFIX(FIND(<o, t>))
    }
    else if (o == *o1)
    { /* case 2 */
        add-a-typed-obj-name(<o1, t * >)
        let e1 be FIND(<o1, t * >)
        if there is (< * , t * >, tobj) ∈ PREFIX(e1)
            merge-two-equiv-classes(FIND(<o, t>) , FIND(tobj))
        else if there is (<[ ], t [ ]>, tobj) ∈ PREFIX(e1)
        {
            PREFIX(e1) = (PREFIX(e1) - { (<[ ], t [ ]>, tobj) }) ∪ { < * , t * > }
            merge-two-equiv-classes(FIND(<o, t>) , FIND(tobj))
        }
        else
            add (< * , t * > , <o, t>) to PREFIX(e1)
    }
    else if (o == o1[ ])
    { /* case 3 */
        add-a-typed-obj-name(<o1, t [ ]>)
        let e1 be FIND(<o1, t [ ]>)
        if there is (<[ ], t [ ]>, tobj) ∈ PREFIX(e1)
            merge-two-equiv-classes(FIND(<o, t>) , FIND(tobj))
        else if there is (< * , t * >, tobj) ∈ PREFIX(e1)
            merge-two-equiv-classes(FIND(<o, t>) , FIND(tobj))
        else
            add (<[ ], t [ ]> , <o, t>) to PREFIX(e1)
    }
    else if (o == o1.m)
    { /* case 4 */
        let the structure or union type containing m be t1
        add-a-typed-obj-name(<o1, t1>)
        if there is (<m, t1> , tobj) ∈ PREFIX(FIND(<o1, t1>))
            merge-two-equiv-classes(FIND(<o, t>) , FIND(tobj))
        else
            add (<m, t1> , <o1.m, t>) to PREFIX(FIND(<o1, t1>))
    }
}
```

Figure 6.3: add-a-typed-obj-name()

```
merge-two-equiv-classes(e_1,e_2)
{
    if (e_1 == e_2) return

    e = UNION(e_1 , e_2) /* union the two classes */

    /* calculate the new prefix set */
    new-prefix = PREFIX(e_1)
    for each (<a, t>,tobj) ∈ PREFIX(e_2)
        if there is (<a, t>,tobj_1) ∈ new-prefix
            merge-two-equiv-classes(FIND(tobj) , FIND(tobj_1))
        else if <a, t> == <[ ], t_1 [ ]> and there is (< ∗ , t_1 ∗ >,tobj_1) ∈ new-prefix
        {
            merge-two-equiv-classes(FIND(tobj) , FIND(tobj_1))
        }
        else if <a, t> == < ∗ , t_1 ∗ > and there is (<[ ], t_1 [ ]>,tobj_1) ∈ new-prefix
        {
            merge-two-equiv-classes(FIND(tobj) , FIND(tobj_1))
            new-prefix = (new-prefix - { (<[ ], t_1 [ ]>,tobj_1) } ) ∪ { (< ∗ , t_1 ∗ >,tobj) }
        }
        else
            new-prefix = new-prefix ∪ { (<a, t>,tobj) }

    PREFIX(e) = new-prefix /* set the prefix set */
}
```

Figure 6.4: merge-two-equiv-classes()

Each equivalence class e maintains a prefix set, PREFIX(e), which indicates relations between names in this class and other classes. If there is a tuple ($<a,\ t>$,tobj) $\in$PREFIX(e), where a is an accessor, t is a type, and $tobj$ is a typed object name, then there are a name $<o_1, t_1>$ in e and a name $<o_2,\ t_2>$ in FIND($tobj$) such that $<o_2,\ t_2>$ $=$ $apply_{\mathrm{T}}(<o_1,\ t_1>\ ,\ <a,\ t>)$. For example, the following are some prefix sets:

Let e_1 be an equivalence class with two names:

$$<p,\ struct\ foo\ *> \text{ and } <\&s,\ struct\ foo\ *>$$

Let e_2 be an equivalence class with two names:

$$<*p,\ struct\ foo> \text{ and } <s,\ struct\ foo\ *>$$

PREFIX(e_1) $=$ { ($<*\ ,\ struct\ foo\ *>$,$<s,\ struct\ foo\ *>$) }

PREFIX(e_2) $=$ { ($<a,\ struct\ foo>$,$<s.a,\ int>$), ($<b,\ struct\ foo>$,$<s.b,\ float\ *>$),
($<c,\ struct\ foo>$,$<s.c,\ float\ *>$) }

Equivalence classes are represented by nodes in G_{PE}; tuples in a prefix set of an equivalence class are represented by edges in G_{PE}. There are two basic operations in calculating the PE relation (constructing G_{PE}). One is to add a typed object name and another is to merge two equivalence classes.

In Figure 6.3, we give the routine for adding a typed object name. When a name is added, we first check if the name already exists; if not, an equivalence class for the name is created and the prefix set for the class is initialized to be empty.

If the added name is $<\&o_1, t_1\ *>$ (case 1), the name $<o_1, t_1>$ is added by recursively invoking add-a-typed-obj-name() and an edge is then added from the node for $<\&o_1,$ $t_1\ *>$ to the node for $<o_1,\ t_1>$.

If the added name is $<*o_1,\ t>$ (case 2), the name $<o_1,\ t\ *>$ is added by recursively invoking add-a-typed-obj-name(). The outgoing edges from the node for $<o_1,\ t\ *>$ (i.e., tuples in PREFIX(FIND($<o_1,\ t\ *>$))) are examined. There are three cases.

- If there is already an edge annotated with $<*\ ,\ t\ *>$, the target node of the edge is merged with the node for $<*o_1,\ t>$.

- If there is already an edge annotated with $<[\,], t\,[\,]>$, the annotation for the edge is changed into $< *\,, t\ \ * >$ and the target node of the edge is merged with the node for $<*o_1, t>$.
- If there is no outgoing edge annotated with either $< *\,, t\ \ * >$ or $<[\,], t'\ [\,]>$, an edge is added from the node for $<o_1, t\ \ * >$ to the node for $<*o_1, t>$.

If the added name is $<o_1[\,], t>$ (case 3), the name $<o_1, t\,[\,]>$ is added by recursively invoking add-a-typed-obj-name(). The outgoing edges from the node for $<o_1, t\ \ * >$ are examined. The three cases are similar to the ones for added name $< * \, o_1, t>$.

Lastly, if the added name is $<o_1.m, t>$ and the structure or union type containing member m is t_1 (case 4), the name $<o_1, t_1>$ is added by recursively invoking add-a-typed-obj-name(). The outgoing edges from the node for $<o_1, t_1>$ are examined. If there is already an edge annotated with $<m, t_1>$, the target node of the edge is merged with the node for $<o_1.m, t>$; otherwise an edge is added from the node for $<o_1, t_1>$ to the node for $<o_1.m, t>$.

In Figure 6.4, we give the routine for merging two equivalence classes. When two classes are merged, they are first unioned and the outgoing edges from the two (i.e., their PREFIX sets) are examined to determine the edges from the new class as the result of the union. If each class has an edge with same annotation, the new class has one edge with the annotation and the targeted equivalence classes of the edges are merged. If one class has an edge annotated with $<*, t\ *>$ and another has an edge annotated with $<[\,], t\,[\,]>$, the new class has an edge annotated with $< *\,, t\ \ * >$ only and the targeted classes of the edges are merged. If one class has an edge with an annotation that another does not have, the new class will have the edge. Intuitively, merge-two-equiv-classes() is used to maintain weakly right-regular equivalence relations.

In both operations, an edge annotated with $< *\,, t\ \ * >$ is preferred over an edge with $<[\,], t\,[\,]>$.

The calculation of the initial PE relation involves the following steps:

1. add each name in B^1
2. for each $(<lhs, t_{lhs}>, <rhs, t_{rhs}>)$ in R^1, merge the two equivalence classes:

```
done = FALSE
do {
    equiv-class-set = find-PE-equiv-classes-for-indirect-calls()

    typed-obj-name-set = find-related-typed-object-names(equiv-class-set)

    lhs-rhs-set = find-related-lhs-rhs(equiv-class-set)

    done = incremental-points-to-analysis(typed-obj-name-set, lhs-rhs-set)

    if (done == FALSE)
    {
        resolve-indirect-calls()

        update-PE-graph()
    }

} until (done == TRUE)
```

Figure 6.5: Resolving Indirect Calls by Points-to Analysis

$$\text{FIND}(<lhs,\, t_{lxkhs}>) \text{ and } \text{FIND}(<rhs,\, t_{rhs}>)$$

3. for each $(<l_1, t_1>,<l_2, t_2>)$ in E^1, merge the two equivalence classes:

$$\text{FIND}(<l_1,\, t_1>) \text{ and } \text{FIND}(<l_2,\, t_2>)$$

6.3.2 Function Pointers and Indirect Calls

If there is an indirect call in any reachable procedure, we need to know what procedures may be invoked by the call. This can be achieved by an aliasing or a points-to analysis algorithm. We will use the points-to analysis presented in Chapter 4. Furthermore, we will use the program decomposition provided by G_{PE} and analyze only program segments related to function pointers used in indirect calls, that is, we will do a partial points-to analysis. G_{PE} may be changed by the results of the partial points-to analysis, for instance, when a procedure is found to be reachable because it is invoked by an indirect call, names and pointer-related assignments in the procedure will be added to G_{PE}. Therefore an incremental approach is necessary.

Calculation of PE Relation After Resolving Indirect Calls In Figure 6.5, we show the high-level description of an iterative algorithm for resolving indirect calls. Below we explain the steps in each iteration one by one.

find-PE-equiv-classes-for-indirect-calls()

For each indirect call $fp(...)$ in a reachable procedure, the PE equivalence class for fp is of interest for resolving indirect calls. This routine returns the set of PE equivalence classes for function pointers used in indirect calls in reachable procedures. Each of these classes corresponds to a node in G_{PE}. Let us call the set of these nodes N_1. For example, in the connected component shown in Figure 6.6 (a), the dashed node is in N_1.

Names and pointer-related assignments in connected components in G_{PE} without nodes in N_1 are not of interest for resolving indirect calls and thus are not considered in the partial points-to analysis. Even within a connected component with nodes in N_1, it is not necessary to consider all names or all pointer-related assignments in the points-to analysis for resolving indirect calls. The next two routines will find the related names and assignments.

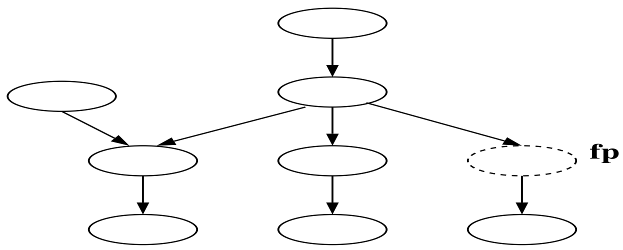
find-related-lhs-rhs()

Given the set of nodes N_1, this routine finds all nodes in G_{PE} such that there exists a directed path from each of them to a node in N_1. Let us call the set of these nodes N_2. Note that $N_1 \subseteq N_2$. For example, in Figure 6.6 (b), the dashed nodes are in N_2.

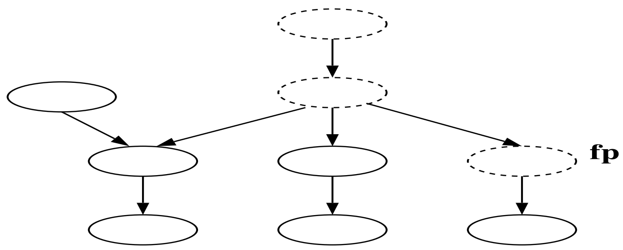
The pointer-related assignments ($lhs = rhs$) involving names associated with nodes in N_2 are of interest for resolving indirect calls and will be considered in the partial points-to analysis. Intuitively, assignments involving names associated with nodes that are *not* in N_2, can not create aliases related to names associated with nodes in N_1. Therefore they do not have to be considered for resolving indirect calls.

find-related-typed-object-names()

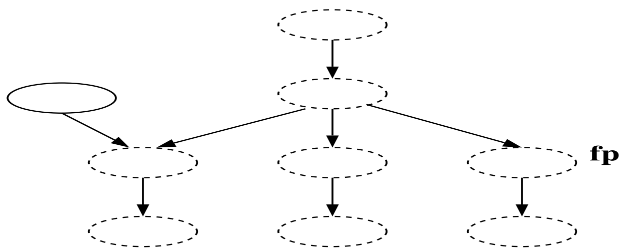
Given the set of nodes N_2, this routine finds all the nodes such that there exists a directed path from a node in N_2 to each of the nodes. Let us call the set of these nodes N_3. Note that $N_2 \subseteq N_3$. For example, in Figure 6.6 (c), the dashed nodes are in N_3.



(a) find-PE-equiv-classes-for-indirect-calls()



(b) find-related-lhs-rhs()



(c) find-related-typed-object-names()

Figure 6.6: Partial Points-to Analysis for Indirect Calls within One Connected Component of G_{PE}

The typed object names associated with nodes in N_3 are of interest for resolving indirect calls and will be considered in the partial points-to analysis. Intuitively, the pointer-related assignments being considered for the points-to analysis may create aliases involving names associated with nodes in N_3.

incremental-points-to-analysis()

This is a modified version of the points-to analysis algorithm in Chapter 4. It takes a set of typed object names and a set of pointer-related assignments as arguments. It maintains a pointer value flow graph, which is initially empty, incrementally creates one node for each typed object name if there does not exist one for the name already, and creates one $\longrightarrow$ edge for each pointer-related assignment if there does not exist one for the assignment already. When a node or a $\longrightarrow$ edge is created, a worklist will be updated with points-to pairs to be propagated in the pointer value flow graph. If the worklist is empty after considering all typed object names and all pointer-related assignments, this routine returns TRUE, which means no iteration is required. If the worklist is not empty, the routine will propagate points-to pairs in the worklist and then returns FALSE, which means another iteration is necessary.

If any pointer-related assignment with type casting is considered in the points-to analysis, locations may be found to be accessed as types other than their declared types. Thus the points-to analysis may create new typed object names and may find new access patterns caused by *effective* unions. For example, a location l may be accessed as type t, which is not its declared type, because a pointer of type $t\ *$ points to it. In this case, a new access pattern for the location is found and the name $<l, t>$ is created; for the purpose of program decomposition, we assume the name $<\&l, t\ *>$ is also created. Furthermore, if t is either a structure type, or a union type, or an array type, more location names are created by the points-to analysis (see Figure 4.21 in Chapter 4).

resolve-indirect-calls()

Let $fp(...)$ be an indirect call in a reachable procedure and t_{fp} is the declared type of fp. For each $<proc, t_{fp}>$ in $pts(<fp, t_{fp}>)$, the procedure $proc$ is considered being invoked at the indirect call.

When a procedure *proc*, which is not reachable, is found to be called through a function pointer, any procedure such that there is a chain of direct or indirect calls from *proc* to it, will be considered reachable if it is not already considered so.

G_{PE} is updated for procedures found reachable due to indirect calls. Given such a procedure, for each of the typed object names and their prefixes in the procedure, $<o,\ t>$, a node is added to G_{PE} if there does not exist one already; for each of the pointer-related assignments in the procedure, $lhs = rhs$, where t_{lhs} and t_{rhs} are the declared types of lhs and rhs respectively, the node for $<lhs,\ t_{lhs}>$ and the node for $<rhs,\ t_{rhs}>$ are merged in G_{PE}. If any new location name in the procedure is of union type, access patterns and common initial sequences for the location will be determined.

update-PE-graph()

This routine updates G_{PE} with the results of the partial points-to analysis. Specifically, it does the following:

1. update G_{PE} with new typed object names created in the points-to analysis

   For each name created in the points-to analysis, a node is added to G_{PE}.

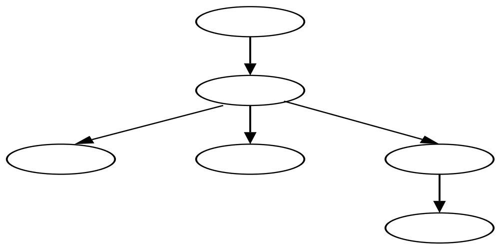
2. update G_{PE} with the points-to information

   Let $<p,\ t\ *>$ be a pointer name. For each $<l,\ t>$ in $pts(<p,\ t\ *>)$, one of the following must be true:

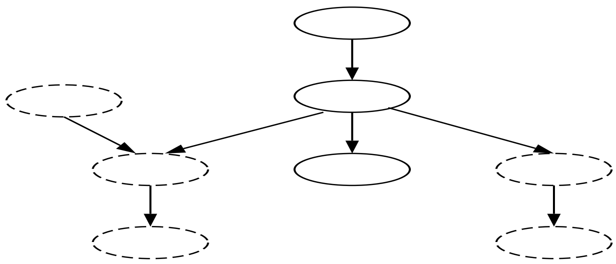
   - l is of the form $l_1[\ ]$ and $<l_1,\ t\ [\ ]>$ is used in a reachable procedure.
   - $<\&l,\ t\ *>$ is used in a reachable procedure.
   - l is accessed as type t because the pointer $<p,\ t\ *>$ points to it. In this case, the name $<\&l,\ t\ *>$ is created by the points-to analysis.

   In the first case, the node for $<p,\ t\ *>$ and the node for $<l_1,\ t\ [\ ]>$ are merged in G_{PE}. In the other two cases, the node for $<p,\ t\ *>$ and the node for $<\&l,\ t\ *>$ are merged in G_{PE}.

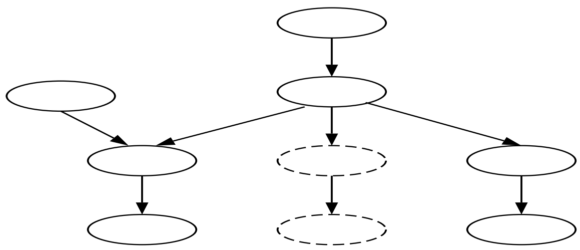
   Basically, the results of the points-to analysis induce a relation; In the first case above, $(<p,\ t>, <l_1,\ t\ [\ ]>)$ is in the relation. In the other two cases, $(<p,\ t>, <\&l,\ t\ *>)$ is in the relation.



the Initial Connected Component



the Connected Component after the First Iteration



the Connected Component after the Second Iteration

Figure 6.7: Changes in One Connected Component of G_{PE}

3. update G_{PE} with the equivalence relation implied by the access patterns which are found by the points-to analysis

   For each ($<l_1, t_1>$,$<l_2, t_2>$) in the equivalence relation implied by access patterns of declared unions and effective unions found by the points-to analysis, the node for $<l_1, t_1>$ and the node for $<l_2, t_2>$ are merged in G_{PE}.

Figure 6.7 shows the changes of one connected component through two iterations. The dashed nodes (equivalence classes) are either added or updated with new names.

Definition of PE Relation After Resolving Indirect Calls Let P^2 be the set of all reachable procedures after resolving indirect calls by partial points-to analysis. Let B^2 be the set of typed object names defined for P^2 according to Definition 6.2.2. Let R^2 be a relation on B^2 defined below:

$$\left\{ (<lhs, t_{lhs}>, <rhs, t_{rhs}>) \;\middle|\; \begin{array}{l} lhs = rhs \text{ is a pointer-related assignment in a} \\ \text{reachable procedure in } P^2,\ t_{lhs} \text{ is the declared} \\ \text{type of } lhs, \text{ and } t_{rhs} \text{ is the declared type of } rhs \end{array} \right\}$$

Let B^1_{pts} be the set of *all* new typed object names created in the points-to analysis. For each name in B^1_{pts}, there is a node in G_{PE}.

Let T^1 be the relation on $(B^2 \cup B^1_{pts})$ induced by the points-to analysis. Let $<p, t\ *>$ be any pointer name. For $<l_1[\,], t> \in pts(<p, t\ *>)$, $(<p, t\ *>, <l_1, t\ [\,]>) \in T^1$; the node for $<p, t\ *>$ and the node for $<l_1, t\ [\,]>$ are merged in G_{PE} by update-PE-graph(). For any other $<l[\,], t> \in pts(<p, t\ *>)$, $(<p, t\ *>, <\&l, t\ *>) \in T^1$; the node for $<p, t\ *>$ and the node for $<\&l, t\ *>$ are merged in G_{PE} by update-PE-graph().

Let E^2 be the equivalence relation implied by access patterns of declared unions and effective unions. For each $(<l_1, t_1>, <l_2, t_2>) \in E^2$, the node for $<l_1, t_1>$ and the node for $<l_2, t_2>$ are merged in G_{PE} by update-PE-graph().

R^2, T^1, and E^2 are relations on $(B^2 \cup B^1_{pts})$. The PE relation after resolving indirect calls is $(R^2 \cup T^1 \cup E^2)^{wr}$.

Note that the following are true:

- $P^2 \supseteq P^1$

When indirect calls are resolved, some procedures, which are not reachable before, may become reachable. P^2 contains procedures that are reachable due to both direct calls and indirect calls; P^1 contains procedures that are reachable due to direct calls only.

- $B^2 \supseteq B^1$

  By Definition 6.2.2, B^2 contains names in procedures in P^2 and B^1 contains names in procedures in P^1.

- $R^2 \supseteq R^1$

  For R^2, all pointer-related assignments in procedures in P^2 are considered; for R^1, those in procedures in P^1 are considered.

- $E^2 \supseteq E^1$

  E^2 is the equivalence relation implied by access patterns of the effective unions found by the partial points-to analysis and the declared unions in procedures reachable through direct/indirect calls; E^1 is the one implied by access patterns of the declared unions in procedures reachable through direct calls only.

6.3.3 Type Casting

If there is a pointer-related assignment with type casting in a reachable procedure, or if there are two pointer names of different types in a PE equivalence class, it is possible that a location may be accessed in more than one way, that is, the location may be used effectively as a union. In this case, we need to find these effective unions as they induce aliases. Again we will use the points-to analysis presented in Chapter 4 and moreover we will use the program decomposition provided by G_{PE} and analyze only program segments related to type casting. G_{PE} may be changed by the results of the partial points-to analysis, for instance, when two locations in an effective union are found to be aliased, their corresponding PE equivalence classes will be merged. Therefore an iterative approach is necessary.

Calculation of PE Relation After Resolving Type Casting In Figure 6.8, we

```
done = FALSE
do {
    equiv-class-set = find-PE-equiv-classes-for-type-casting()

    typed-obj-name-set = find-related-typed-object-names(equiv-class-set)

    lhs-rhs-set = find-related-lhs-rhs(equiv-class-set)

    done = incremental-points-to-analysis(typed-obj-name-set, lhs-rhs-set)

    if (done == FALSE)
    {
    update-PE-graph()
    }

} until (done == FALSE)
```

Figure 6.8: Resolving Type Casting by Points-to Analysis

show the high-level description of an iterative algorithm for resolving type casting. The algorithm is similar to the one for resolving indirect calls; the routine that is new here is find-PE-equiv-classes-for-type-casting() and all the other routines are the same as explained earlier. Basically, the routine find-PE-equiv-classes-for-type-casting() returns the set of PE equivalence classes, each of which contains two typed object names, $<o_1, t_1>$ and $<o_2, t_2>$, such that one of the following is true:

- $t_1 == t_1^{'}\ *$, $t_2 == t_2^{'}\ *$, $t_1^{'}$ and $t_2^{'}$ are not the same type.
- $t_1 == t_1^{'}\ [\ ]$, $t_2 == t_2^{'}\ *$, $t_1^{'}$ and $t_2^{'}$ are not the same type.

Intuitively, each of these equivalence classes has pointer names of different types. This handles cases of two pointers of different types in the same PE equivalence due to assignments with type casting in the program or due to merges of PE equivalence classes.

Definition of PE Relation After Resolving Type Casting Let B^2_{pts} be the set of *all* new typed object names created by the points-to analysis. For each name in B^2_{pts}, a node has been added to G_{PE} by update-PE-graph().

Let T^2 be the relation on $(B^2 \cup B^2_{pts})$ induced by results of the points-to analysis. Let $<p, t\ *>$ be any pointer name. For $<l_1[\ ], t> \in pts(<p, t\ *>)$, $(<p, t\ *>,<l_1, t\ [\ ]>) \in T^2$; the node for $<p, t\ *>$ and the node for $<l_1, t\ [\ ]>$ are merged in G_{PE} by update-PE-graph(). For any other $<l[\ ], t> \in pts(<p, t\ *>)$, $(<p, t\ *>,<\&l, t\ *>) \in T^2$; the node for $<p, t\ *>$ and the node for $<\&l, t\ *>$ are merged in G_{PE} by update-PE-graph().

Let E^3 be the equivalence relation implied by access patterns of declared unions and effective unions. It is a relation on $(B^2 \cup B^2_{pts})$. For each $(<l_1, t_1>,<l_2, t_2>) \in E^3$, the node for $<l_1, t_1>$ and the node for $<l_2, t_2>$ are merged in G_{PE} by update-PE-graph().

Note that new names are created and new access patterns are found by points-to analysis because of pointer assignments with type casting. The points-to analysis for type casting considers *all* these assignments in reachable procedures in P^2 and the points-to analysis for indirect calls considers these affecting function pointers used in indirect calls. Therefore we have $B^2_{pts} \supseteq B^1_{pts}$ and $E^3 \supseteq E^2$.

R^2 and T^1 are relations on $(B^2 \cup B^1_{pts})$; since $B^2_{pts} \supseteq B^1_{pts}$, they are also relations on $(B^2 \cup B^2_{pts})$. T^2 and E^3 are relations on $(B^2 \cup B^2_{pts})$. The PE relation after resolving type casting is $(R^2 \cup T^1 \cup T^2 \cup E^3)^{wr}$.

When there is no indirect call, union, or type casting, the following are true:

- $R^2 = R^1$
- $B^2_{pts} = \emptyset$
- $T^1 = T^2 = \emptyset$
- $E^3 = \emptyset$

Therefore the PE relation defined here subsumes the one defined in Chapter 5 where there is no indirect call, union, or type casting.

6.3.4 Final Program Decomposition

The resulting PE relation can be represented as a directed multi-graph (G_{PE}), where nodes are equivalence classes, edges indicate prefix relations between names in equivalence classes, and edges are annotated with typed accessors.

G_{PE} can be decomposed into weakly connected components. These components partition the names and the pointer-related assignments in the program.

Without the step for resolving type casting, we still get a program decomposition at the level of weakly connected components. With the step for resolving type casting, we have a further decomposition within each weakly connected component. Each node in a component has a set of pointer-related assignments; the assignments at one node n_1 will affect the assignments at another node n_2 *if and only if* there is a path from n_1 to n_2 in G_{PE}. This fine decomposition is used in partial points-to analysis for indirect calls through function pointers and type casting.

6.4 Flow-insensitive Aliasing Analysis with Unions and Type Casting

Definition of FA Relation Let n be a node in G_{PE}. $B_{\mathrm{PE}}(n)$ is the set of typed object names associated with the node n in G_{PE}.

Given the G_{PE} for a program, B is the set of typed object names defined below:

$$B = \left\{ <o,\, t> \;\middle|\; \begin{array}{l} \text{there is a path from a node } n \text{ to a node } m \text{ in } G_{\mathrm{PE}} \text{ such} \\ \text{that the path is annotated with a sequence of accessors} \\ <a_1,\, t_1>\ <a_2,\, t_2>...<a_j,\, t_j>\ (j \geq 0), \\ <o,\, t> \;=\; apply^{*}_{\mathrm{T}}(<o',\, t'>\ ,\ <a_1,\, t_1>\ <a_2,\, t_2>...<a_j,\, t_j>), \\ <o',\, t'> \in B_{\mathrm{PE}}(n), \text{ and } o \text{ is not of the form } \&o_1 \end{array} \right\}$$

Any name $<o,\, t> \in (B^2 \cup B^2_{pts})$ such that o is not of the form $\&o_1$, is in B.

Let R be a relation on B defined below:

$$R = \left\{ (tobj_1^{'}, tobj_2^{'}) \left| \begin{array}{l} \text{there is a path from a node } n \text{ to a node } m \text{ in } G_{\mathrm{PE}} \text{ such} \\ \text{that the path is annotated with a sequence of accessors} \\ <a_1,\, t_1> \; <a_2,\, t_2> ... <a_j,\, t_j> \; (j \geq 1), \\ \text{NumOfDerefs}(<a_1,\, t_1> \; <a_2,\, t_2> ... <a_j,\, t_j>) \geq 1, \\ tobj_1^{'} = apply^*_{\mathrm{T}}(tobj_1 \;,\, <a_1,\, t_1> \; <a_2,\, t_2> ... <a_j,\, t_j>), \\ tobj_2^{'} = apply^*_{\mathrm{T}}(tobj_2 \;,\, <a_1,\, t_1> \; <a_2,\, t_2> ... <a_j,\, t_j>), \\ \text{where } tobj_1 \in B_{\mathrm{PE}}(n) \text{ and } tobj_2 \in B_{\mathrm{PE}}(n) \end{array} \right. \right\}$$

Let E^3 be the equivalence relation implied by access patterns of declared unions and effective unions; it is calculated by the points-to analysis for resolving type casting. E^3 is a relation on $(B^2 \cup B^2_{pts})$ and none of the names in E^3 contains &; thus it is a relation on B. The FA relation is $(R \cup E^3)^{wr}$.

Calculation of FA Relation

We use the idea in Chapter 5 and represent the FA relation by a directed graph called G_{FA}. The graph has the similar structure as G_{PE}, that is, nodes represent equivalence classes of typed object names; edges represent prefix relation between names and are annotated with typed accessors such as $< * \,,\, t \;\; * >$ and $<m,\, t>$. The calculation of the FA relation, shown in Figure 6.9, is tantamount to the construction of G_{FA}. The algorithm represents nodes by equivalence classes and edges by tuples in PREFIX sets.

Given the set of typed object names in the final G_{PE}, $(B^2 \cup B^2_{pts})$, the initial G_{FA} has one node for each name $<o,\, t>$ in the set such that o is *not* of the form $\&o_1$ and has no edges.

First, each edge in G_{PE} annotated with a member of a structure or a union, $<m,\, t>$, is examined. Assume the edge is from node n_1 to node n_2 in G_{PE}. For each name $<o,\, t>$ in $B_{\mathrm{PE}}(n_1)$, an edge annotated with $<m,\, t>$ is added in G_{FA} from the node for $<o,\, t>$ to the node for $apply_{\mathrm{T}}(<o,\, t> \;,\, <m,\, t>)$ if $apply_{\mathrm{T}}(<o,\, t> \;,\, <m,\, t>) \in B_{\mathrm{PE}}(m)$.

Second, each edge in G_{PE} annotated with an array reference, $<[\,],\, t\,[\,]>$, is examined. Assume the edge is from node n_1 to node n_2 in G_{PE}. For each name $<o,\, t\,[\,]>$ in $B_{\mathrm{PE}}(n_1)$, an edge annotated with $<[\,],\, t\,[\,]>$ is added in G_{FA} from the node for $<o,\, t\,[\,]>$ to the node for $apply_{\mathrm{T}}(<o,\, t\,[\,]> \;,\, <[\,],\, t\,[\,]>)$ if $apply_{\mathrm{T}}(<o,\, t\,[\,]> \;,\, <[\,],\, t\,[\,]>) \in B_{\mathrm{PE}}(m)$.

```
calculate-FA-relation()
{
   for each <o, t> ∈ (B^2 ∪ B^2_pts) such that o is not of the form &o_1
   {
      INIT-EQUIV-CLASS(<o, t>)
      PREFIX(FIND(<o, t>)) = { }
   }

   for each edge in G_PE from n_1 to n_2 annotated with accessor <m, t>,
   where t is a structure or a union type and m is a member of the type
   {
      for each <o, t> ∈ B_PE(n_1) such that apply_T(<o, t> , <m, t>) ∈ B_PE(n_2)
         add (<m, t>, apply_T(<o, t> , <m, t>)) to PREFIX(FIND(<o, t>))
   }

   for each edge in G_PE from n_1 to n_2 annotated with accessor <[ ], t [ ]>
   {
      for each <o, t [ ]> ∈ B_PE(n_1) such that apply_T(<o, t [ ]> , <[ ], t [ ]>) ∈ B_PE(n_2)
         add (<[ ], t [ ]>, apply_T(<o, t [ ]> , <[ ], t [ ]>)) to PREFIX(FIND(<o, t [ ]>))
   }

   for each edge in G_PE from n_1 to n_2 annotated with accessor < * , t  * >
   {
      tobj-set = { tobj | tobj ∈ B_PE(n_2), tobj_1 ∈ B_PE(n_1),
                          and tobj = apply_T(tobj_1 , < * , t  * >) }
      let tobj be an arbitrary name in tobj-set
      for each tobj_2 ∈ obj-set other than tobj
         merge-two-equiv-classes(FIND(tobj_2),FIND(tobj))
      for each <o_1, t_1> ∈ B_PE(n_1) such that o_1 is not of the form &o_1'
         if there is not a (< * , t  * >, tobj_3) in PREFIX(FIND(<o_1, t_1>)) such that
           FIND(tobj_3) == FIND(tobj)
         {
           add (< * , t  * >,tobj) to PREFIX(FIND(<o_1, t_1>))
         }
   }

   for each (<l_1, t_1>,<l_2, t_2>) ∈ E^3
      merge-two-equiv-classes(FIND(<l_1, t_1>),FIND(<l_2, t_2>))
}
```

Figure 6.9: Calculation of the FA Relation

Finally, each edge in G_{PE} annotated with a pointer dereference, $< * \ , \ t \ \ * >$, is examined. Assume the edge is from node n_1 to node n_2 in G_{PE}. As the first step, we get all names $tobj$ in $B_{\mathrm{PE}}(m)$ such that there exits a name $tobj_1$ in $B_{\mathrm{PE}}(n_1)$ and $tob =$ $apply_{\mathrm{T}}(tobj_1 \ , < * \ , \ t \ \ * >)$. The equivalence classes for these names (nodes in G_{FA}) are merged in G_{FA}; let n be the node for the resulting equivalence class in G_{FA}. As the next step, for each name $<o_1, t_1>$ in $B_{\mathrm{PE}}(n_1)$ such that o_1 does not contain &, an edge annotated with $< * \ , \ t \ \ * >$ is added in G_{FA} from the node for $<o_1, t_1>$ to the node n if no such edge exists already.

In the final G_{FA}, each node represents an equivalence class and is annotated with the set of typed object names in the class; each edge from a node for equivalence class e_1 to a node for equivalence class e_2, corresponds to a tuple $(<a, t> \ , tobj)$ in PREFIX(e_1) such that $e_2 =$ FIND($tobj$), and the edge is annotated with the typed accessor $<a, t>$.

The partial FA relation, induced by the equivalence classes associated with nodes in G_{FA}, can be used as compile-time aliasing information and will be used to solve Thru-deref MOD/REF problems in our empirical study.

6.5 Empirical Results

We have implemented a prototype of the program decomposition algorithm and the flow-insensitive aliasing analysis presented in Section 6.3 and 6.4. Our prototype implementation is written in C and compiled by *gcc* with optimization turned on (*-O2*). Our implementation utilizes the blackbox language [AL95] to specify effects of library functions so that one call to a library function can be treated independent of other calls to the same function; by doing this, we avoid forcing actual arguments for the same formal argument of a library function to be in the same PE equivalence class.

In this section, we show some empirical results of the implementation.

Test Programs The test programs will be the same as those in Table 4.1.

Program Decompositin In Figure 6.10 and 6.11, we show the time used by the program decomposition for each of the test programs; for comparison, we also show the time of a simple compilation using *gcc* without any optimization for each program. The

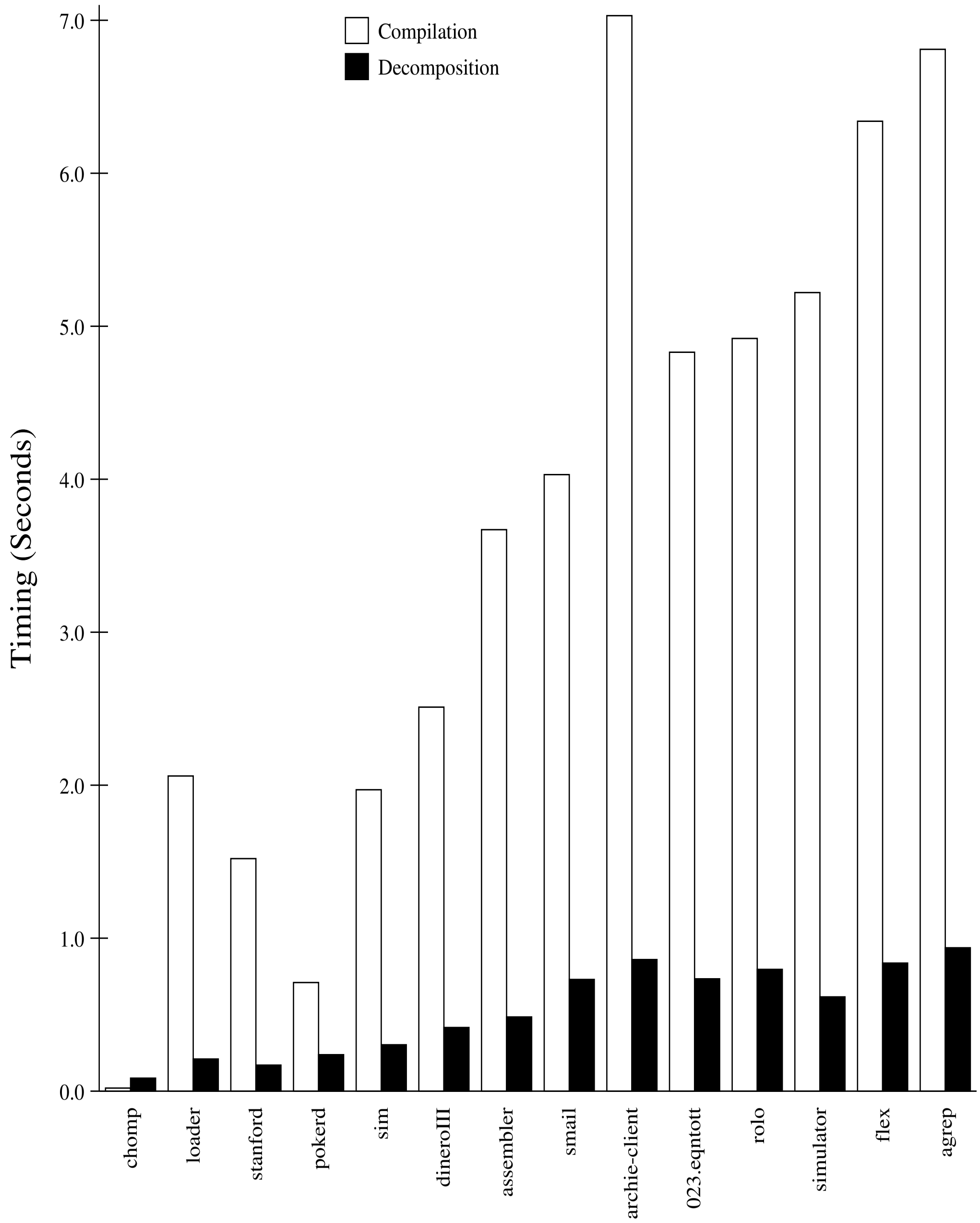


Figure 6.10: Timings for the Program Decomposition (Part 1)

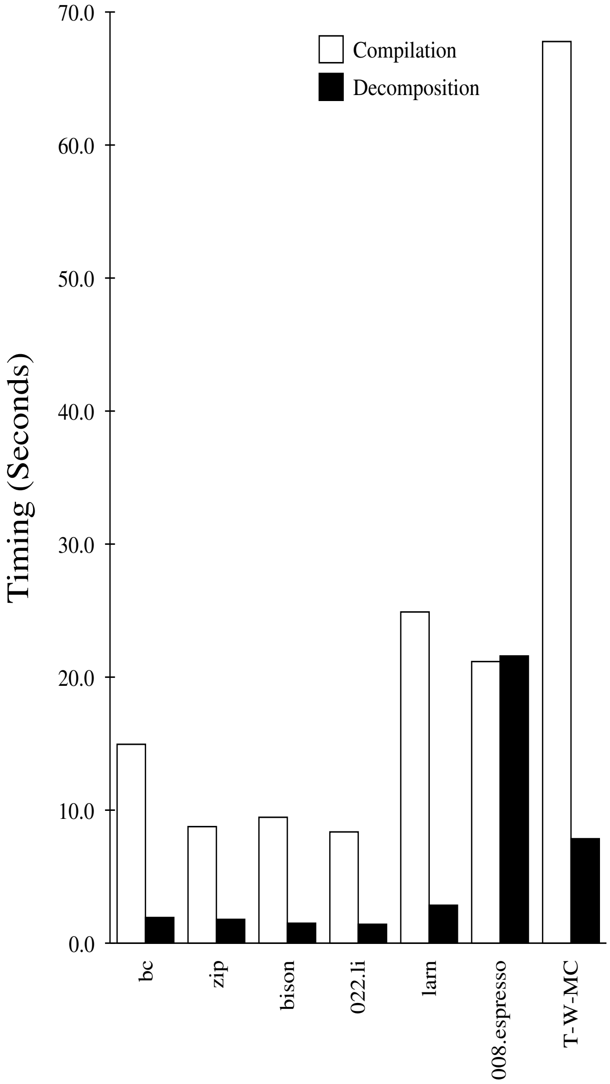


Figure 6.11: Timings for the Program Decomposition (Part 2)

program name	number of weakly connected components					number of pointer-related assignments				
	k=1	k=2	$2 < k \neq \infty$	$k=\infty$	total	k=1	k=2	$2<k\neq \infty$	$k=\infty$	total
chomp	4	2	0	1	7	6	4	0	136 (93.2%)	146
loader	9	4	0	2	15	149	75	0	58 (20.6%)	282
stanford	9	6	0	1	16	21	20	0	17 (29.3%)	58
pokerd	5	3	0	1	9	21	122	0	30 (17.3%)	173
sim	17	4	0	1	22	177	55	0	29 (11.1%)	261
dineroIII	7	3	0	1	11	78	86	0	90 (35.4%)	254
assembler	15	4	0	2	21	334	338	0	142 (17.4%)	814
smail	17	2	0	1	20	296	57	0	532 (60.1%)	885
archie-client	15	3	0	2	20	127	15	0	580 (80.3%)	722
023.eqntott	11	2	1	2	16	182	39	38	539 (67.5%)	798
rolo	9	1	0	1	11	235	2	0	661 (73.6%)	898
simulator	10	6	0	2	18	105	298	0	35 (8.0%)	438
flex	42	4	0	1	47	513	65	0	109 (15.9%)	687
agrep	18	6	0	2	26	311	81	0	264 (40.2%)	656
bc	5	3	0	4	12	70	10	0	1265 (94.1%)	1345
zip	22	1	2	1	26	239	2	174	849 (67.2%)	1264
bison	47	8	1	7	63	892	275	12	527 (30.9%)	1706
022.li	1	2	0	1	4	10	13	0	1460 (98.4%)	1483
larn	13	13	0	1	27	167	883	0	498 (32.2%)	1548
008.espresso	30	4	0	2	36	699	68	0	6785 (89.8%)	7552
T-W-MC	53	14	2	13	82	1526	121	75	2827 (62.1%)	4549

Table 6.1: Numbers of Weakly Connected Components and Pointer-related Assignments

timing results are obtained by averaging over ten runs of the program decomposition implementation or the compilation on a Sun SPARCstation 20 running Solaris 2.5.1 with 100M byte physical memory and 350M byte swap space. For all programs but 008.*espresso*, the program decomposition takes a fraction of the time for compilation. For all programs but two (008.*espresso* and T-W-MC), it takes less than 3 seconds; it uses about 22 seconds for 008.*espresso* and about 8 seconds for T-W-MC.

In Table 6.1, we show the results of program decomposition for the test programs; specifically, for each program, we give the total number of weakly connected components and pointer-related assignments resulting from our program decomposition. The numbers are broken down by the value of k for each weakly connected component, which is the maximum number of pointer dereferences ($*$) on any path in the component. If k is ∞ for a component, then there is a cycle in that component. The values of k for weakly connected components capture certain characteristics of the pointer-related assignments associated with them. For instance, if k is 1 for a component, then there are only single-level pointers in the pointer-related assignments for the component; if k is ∞ for a component, then some of the pointer-related assignments for the component involve recursive data structures. For example, the program *simulator* has 10 components with $k = 1$, 6 components with $k = 2$ and 2 components with $k = \infty$; there are 105 pointer-related assignments for components with $k = 1$, 298 for components with $k = 2$ and 35 for components with $k = \infty$.

For most of the programs, there are more than 10 weakly connected components; the minimum is 4 for 022.*li* and the maximum is 82 for T-W-MC. All programs have one or more components with $k = \infty$, which means there are recusive data structures in all programs. Two or more components with $k = \infty$ usually mean that there are recursive data structures which are not related to each other; for example, for *simulator*, there are two components, each of which is for a linked list that is independent of the other.

The distribution of pointer-related assignments is mixed. They can be very concentrated in components with $k = \infty$, for example, *chomp*, *bc*, 022.*li*, and 008.*espresso*; for these programs, the program decomposition is not effective at the component level. On the other hand, pointer-related assignments can also be concentrated in components

program	resolving indirec calls			resolving type casting		
name	itera-tions	pointer assignments	object names	itera-tions	pointer assignments	object names
chomp	0	0 (0.0%)	0 (0.0%)	1	0 (0.0%)	0 (0.0%)
loader	0	0 (0.0%)	0 (0.0%)	1	14 (5.0%)	44 (15.8%)
stanford	0	0 (0.0%)	0 (0.0%)	1	0 (0.0%)	0 (0.0%)
pokerd	0	0 (0.0%)	0 (0.0%)	1	98 (56.6%)	138 (66.0%)
sim	0	0 (0.0%)	0 (0.0%)	1	88 (33.7%)	109 (38.4%)
dineroIII	0	0 (0.0%)	0 (0.0%)	2	79 (31.1%)	130 (27.2%)
assembler	0	0 (0.0%)	0 (0.0%)	1	259 (31.8%)	258 (42.9%)
smail	0	0 (0.0%)	0 (0.0%)	2	515 (58.2%)	615 (84.9%)
archie-client	2	2 (0.3%)	6 (0.8%)	2	513 (71.1%)	700 (88.5%)
023.eqntott	2	8 (1.0%)	17 (2.9%)	2	379 (47.5%)	543 (91.3%)
rolo	2	3 (0.3%)	4 (0.6%)	1	422 (47.0%)	585 (88.2%)
simulator	0	0 (0.0%)	0 (0.0%)	2	86 (19.6%)	91 (24.6%)
flex	0	0 (0.0%)	0 (0.0%)	2	178 (25.9%)	227 (38.0%)
agrep	0	0 (0.0%)	0 (0.0%)	1	306 (46.6%)	314 (43.6%)
bc	2	5 (0.4%)	12 (0.9%)	2	924 (68.7%)	1223 (92.7%)
zip	2	1 (0.1%)	4 (0.3%)	2	632 (50.0%)	1164 (90.9%)
bison	0	0 (0.0%)	0 (0.0%)	2	783 (45.9%)	1233 (84.4%)
022.li	2	1030 (69.5%)	1396 (98.1%)	1	1079 (72.8%)	1418 (99.6%)
larn	2	7 (0.5%)	16 (1.7%)	2	500 (32.3%)	726 (79.1%)
008.espresso	2	31 (0.4%)	40 (0.7%)	2	4459 (59.0%)	4557 (75.6%)
T-W-MC	2	6 (0.1%)	18 (0.3%)	2	93 (2.0%)	58 (1.0%)

Table 6.2: Numbers of Pointer-related Assignments and Object Names Considered in Resolving Indirect Calls and Type Casting

with $k = 1$ or $k = 2$, for example, *sim*, *simulator*, and *flex*. For most programs, the results show we do get a reasonable program decomposition.

Partial Points-to Analysis for Indirect Calls and Type Casting In Figure 6.12 and 6.13, we show the time used by the points-to analysis for resolving indirect calls and type casting respectively in comparison with the time for program decomposition on each of test programs. The timing results are obtained by averaging over ten runs of the program decomposition implementation. There are 9 programs out of the test programs that have indirect calls through function pointers. For all but one (*T-W-MC*), the partial points-to analysis for resolving indirect calls takes less than 1 second; it takes about 1.8 seconds for *T-W-MC*. In most cases, the time is less than 1/4 of the time for program decomposition and in the worst case (022.*li*), it is close to 1/2 of the

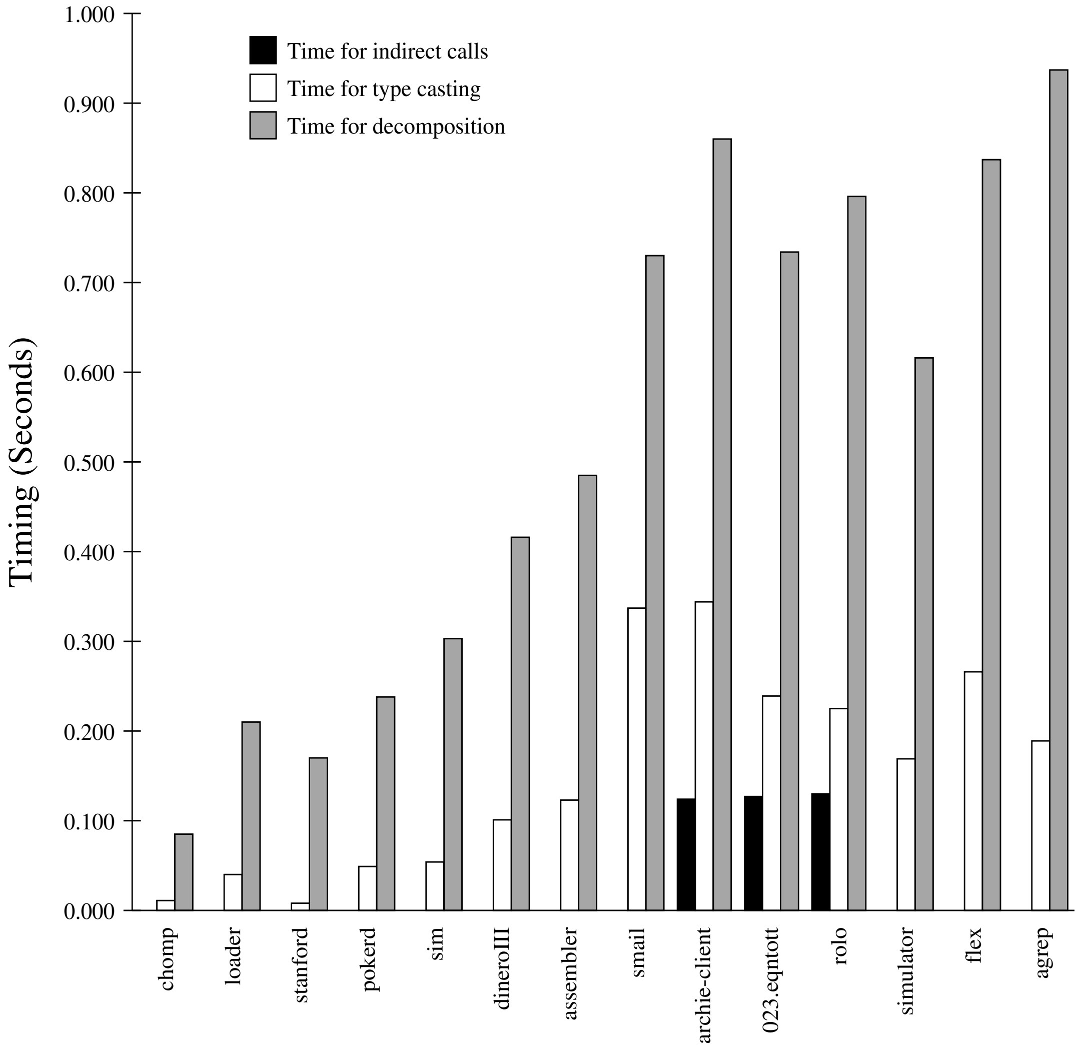


Figure 6.12: Timings for Resolving Indirect Calls and Type Casting (Part 1)

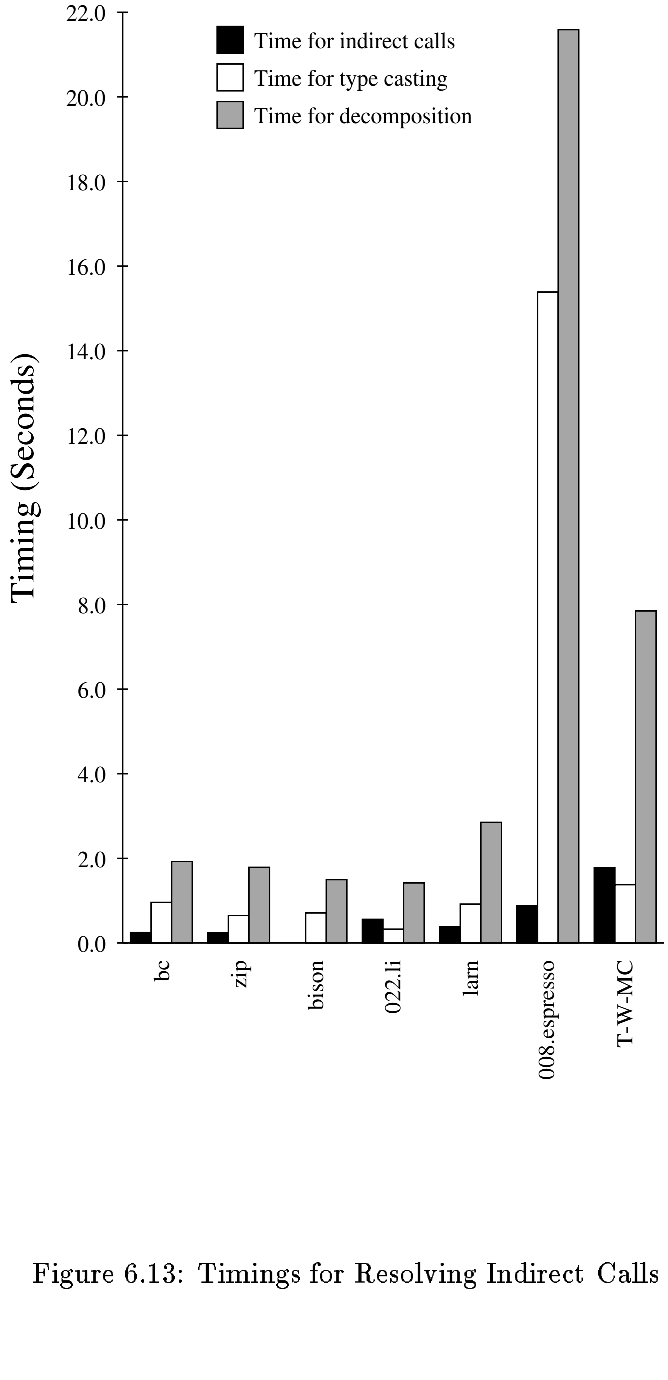


Figure 6.13: Timings for Resolving Indirect Calls and Type Casting (Part 2)

program name	indirect calls	procedure called avg	procedure called min	procedure called max	locations of union type	locations w/ >1 access patterns	access patterns avg	access patterns min	access patterns max
chomp	0	0.0	0	0	0	0	0.0	0	0
loader	0	0.0	0	0	0	0	0.0	0	0
stanford	0	0.0	0	0	0	0	0.0	0	0
pokerd	0	0.0	0	0	2	2	3.0	3	3
sim	0	0.0	0	0	0	0	0.0	0	0
dineroIII	0	0.0	0	0	0	0	0.0	0	0
assembler	0	0.0	0	0	0	0	0.0	0	0
smail	0	0.0	0	0	0	0	0.0	0	0
archie-client	1	2.0	2	2	3	7	2.4	2	3
023.eqntott	9	2.5	1	3	3	8	2.1	2	3
rolo	1	1.0	1	1	0	0	0.0	0	0
simulator	0	0.0	0	0	0	0	0.0	0	0
flex	0	0.0	0	0	0	2	2.0	2	2
agrep	0	0.0	0	0	0	5	5.0	5	5
bc	19	1.4	1	2	3	20	3.0	2	4
zip	1	1.0	1	1	0	5	2.2	2	3
bison	0	0.0	0	0	0	0	0.0	0	0
022.li	3	15.0	15	15	0	0	0.0	0	0
larn	1	6.0	6	6	0	8	2.0	2	2
008.espresso	15	3.2	1	4	0	99	2.0	2	3
T-W-MC	18	2.0	2	2	0	1	2.0	2	2

Table 6.3: Indirect Calls and Access Patterns

time for program decomposition. For all programs but two (008.*espresso* and *T-W-MC*), the partial points-to analysis for resolving type casting takes less than 1 second; it takes about 15.4 seconds for 008.*espresso* and about 1.4 seconds for *T-W-MC*. For all programs but one (008.*espresso*), the partial points-to analysis takes less than 1/2 of the time for program decomposition; for 008.*espresso*, it uses about 7/10 of the time for program decomposition.

In Table 6.2, we present the number of iterations required, the total number of pointer-related assignments and object names considered in resolving indirect calls and type casting. For indirect calls, only a very small fraction of pointer-related assignments and object names need to be considered and the exception is 022.*li*, where nearly 70 percent of pointer-related assignments and 98 percent of the names are taken into account; for the 9 programs with indirect calls, two iterations are required. For type casting, 1/3 of programs require more than 50 percent of pointer-related assignment be examined and the worst case is close to 73% (022.*li*); for most programs, we consider many object names (the worst case is 99.6% for 022.*li*) because of the way we determine names as related (see explanation of find-related-typed-object-names()).

In our implementation, we have done two optimizations when resolving type casting. One optimization is for heap names created for calls to library functions that allocate memory. For example, with the following assignment:

$$int *p = (int\ *)\ malloc(10*\text{sizeof(int)})).$$

a heap name of type *char* [ ] is created for the call to *malloc*(), that is, we have a name $<$*heap*, *char* [ ]$>$. This name is in the same equivalence class as the name $<$*p*, *int* $*$$>$, but the two have different types. In such a case, we will create a new name[4] $<$*heap*, *int* [ ]$>$ and add it to the same equivalence class *prior* to the partial points-to analysis for type casting. If this is the *only* casting in a program, no additional points-to analysis is necessary; this is the case for *chomp* and *stanford*, where no assignments or names are considered even though they both have type casting.

[4]If it is of a structure or union type, more new names are created.

Another optimization is to reduce number of iterations for resolving type casting. This is achieved by checking if there is any merging of PE equivalance classes in calls to routine update-PE-graph(). If there is no merging, no further iteration is necessary; this is the case for those programs such as *loader* and *pokerd*, where one iteration of the partial points-to analysis is required.

In Table 6.3, we show the number of procedures that are found to be invoked at indirect calls by the points-to analysis for resolving indirect calls and the number of access patterns found by the points-to analysis for resolving type casting. The column named "locations of union type" in the table gives the number of locations that are declared to be of union type in each program; the column named "locations w/ >1 access patterns" in the table shows the total number of declared and effective unions in each program.

FA Analysis In Figure 6.14 and 6.15, we present the Thru-deref MOD/REF results of the FA analysis for the test programs; for comparison, we also show the results of the PT analysis for all programs and the results of an extended version of Landi/Ryder FS analysis for 8 programs. Note that this version of the FS analysis handles unions and type casting in a different way from ours (see Page 76 for more explanation). Overall, the results show the FA analysis is less precise than the PT analysis; in some cases, for example, *assembler*, *smail*, *bc*, and 008.*espresso*, the average number of locations found modified or referenced by the FA analysis is twice as much as that found by the PT analysis. In a few cases, for example, *sim* and *dineroIII*, the FA analysis is almost as precise as the PT analysis.

In Figure 6.16 and 6.17, we present the time used by the FA analysis on each of the test programs; for comparison, we also show the time of a simple compilation using *gcc* without any optimization for each of the test programs. The timing results are collected by averaging over ten runs of the FA analysis or the compilation on a Sun SPARCstation 20 running Solaris 2.5.1 with 100M byte physical memory and 350M byte swap space. The time of the FA analysis does not include the time of program decomposition. From the two figures, we can see that the FA analysis is very fast after program decomposition. For all programs but two (008.*espresso* and *T-W-MC*),

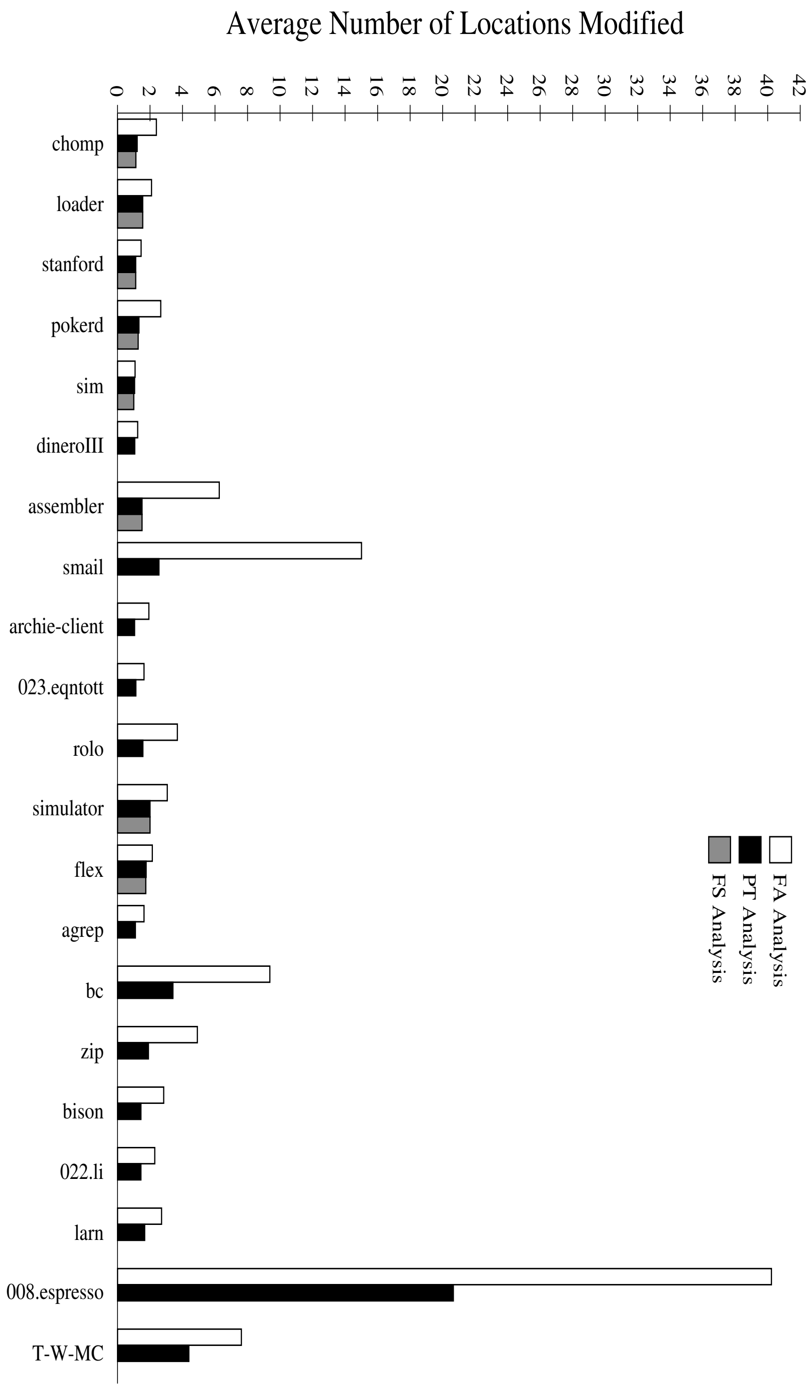


Figure 6.14: Thru-deref MOD Results for the FA Analysis

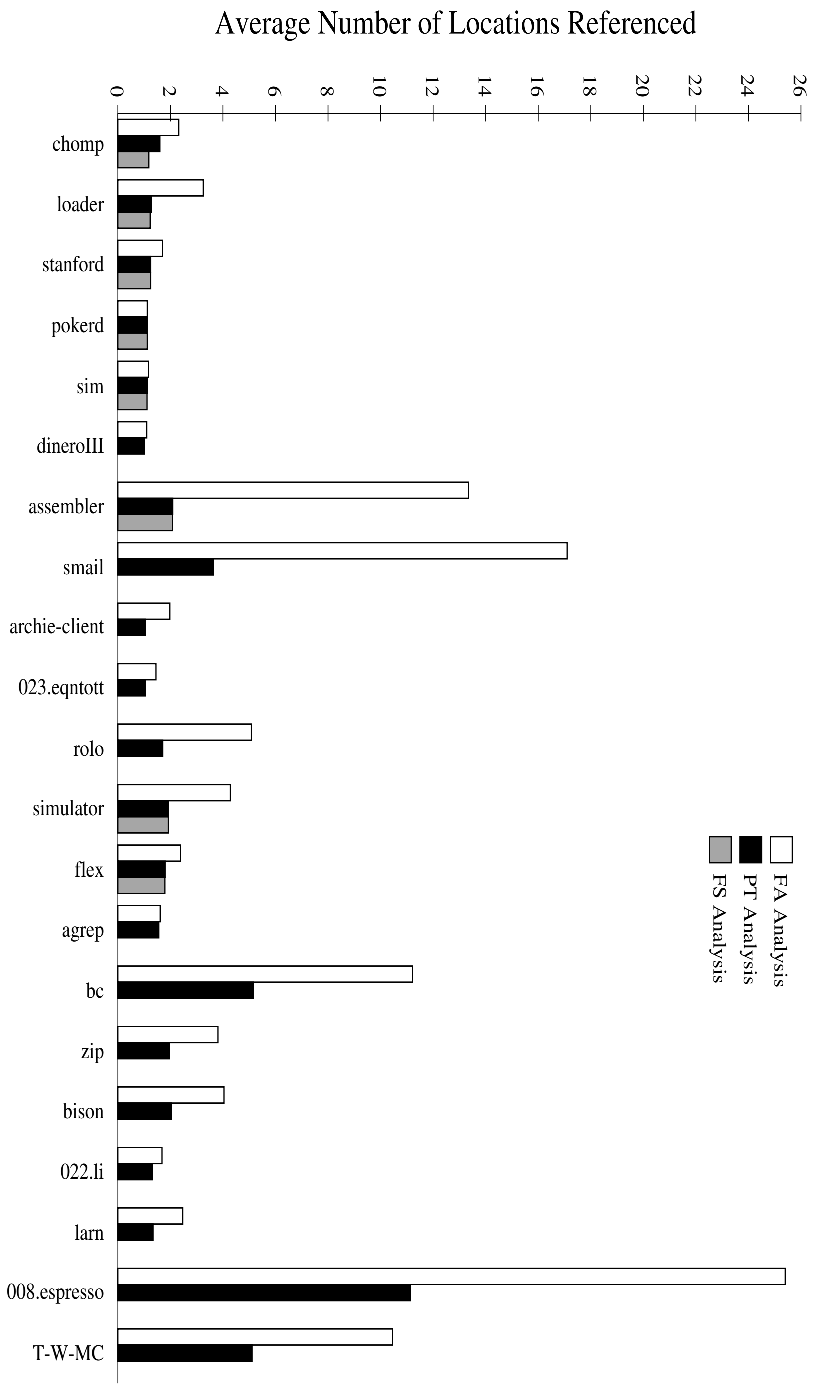


Figure 6.15: Thru-deref REF Results for the FA Analysis

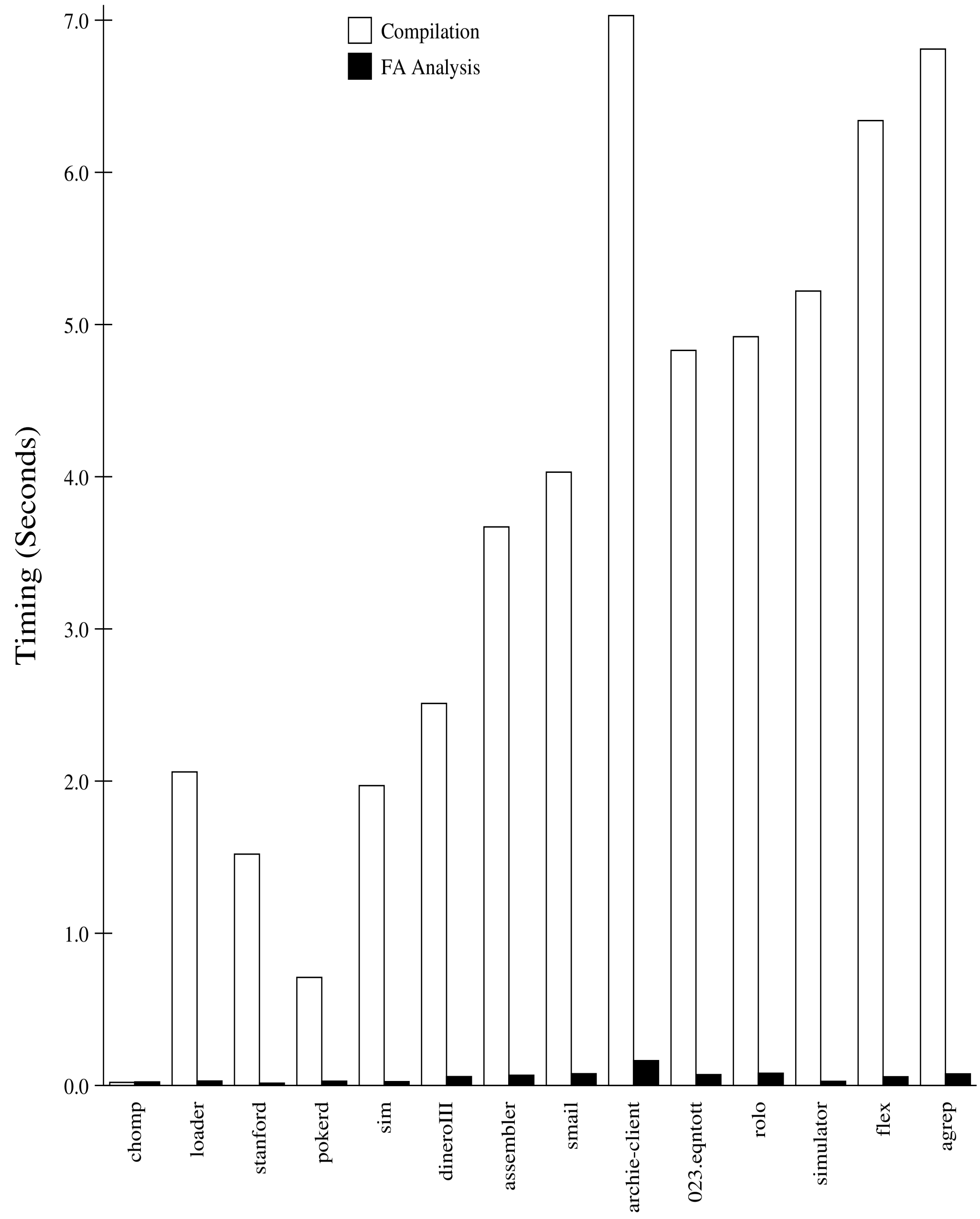


Figure 6.16: Timings for the FA Analysis (Part 1)

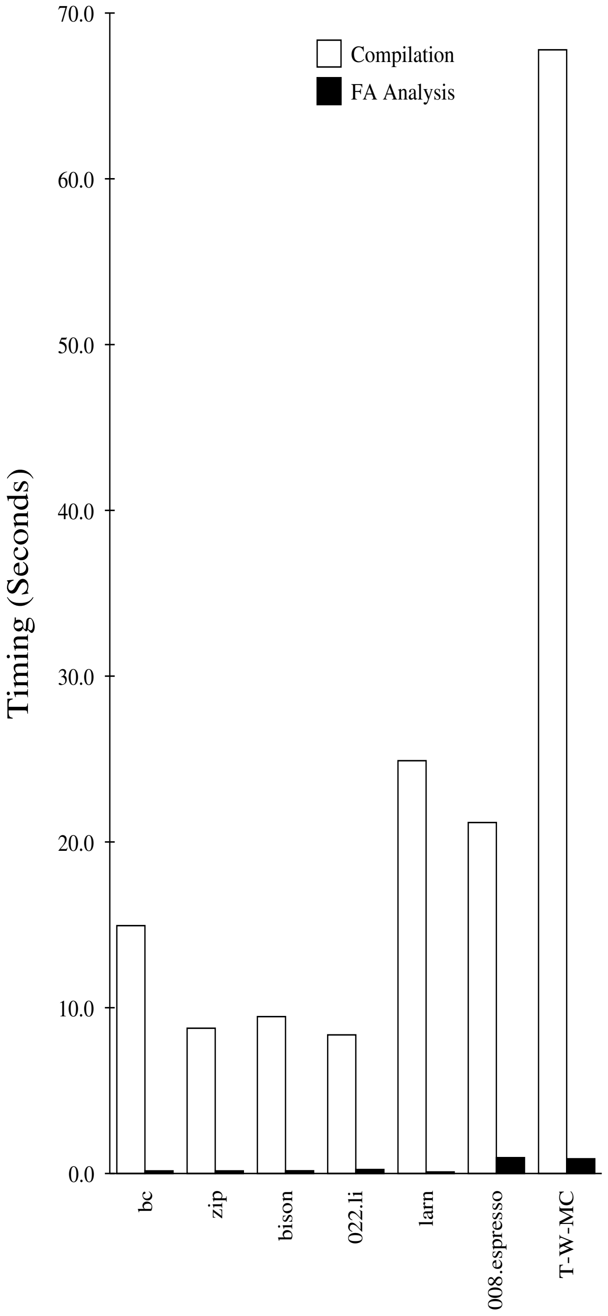


Figure 6.17: Timings for the FA Analysis (Part 2)

it takes less than 0.3 seconds; for 008.*espresso* and *T-W-MC*, it takes less than 1 second. In another words, after program decomposition, we can very quickly obtain the flow-insensitive aliasing information.

6.6 An Example for Program Decomposition and Flow-insensitive Aliasing Analysis

In this section, we show the results of the program decomposition and flow-insensitive aliasing analysis for an example program.

The program is given in Figure 6.18. There is an indirect call in the program ($fp()$). It takes three iterations to resolve the call: in the first iteration, $funcB$ is found being invoked through fp; in the second iteration, $funcA$ is found to be called. The G_{PE} after resolving indirect calls is shown in Figure 6.19, where typed object names in boldface are added in the first iteration and the underlined ones are added in the second iteration; these names are added because $FuncB$ and $funcA$ are found reachable due to the indirect call. The points-to analysis itself does not create any new names; thus the set B^1_{pts} is empty. Since there is no declared union and no effective union is discovered in the points-to analysis for indirect calls, both E^1 and E^2 are emtpy.

It takes two iterations to resolve type casting for the example program. In the first iteration, the following PE equivalence class is found to have names of different pointer types:

$$\{\ <p,\ struct\ foo\ *>,\ <r,\ struct\ bar\ *>,\ <\&s,\ struct\ bar\ *>\ \}$$

The following typed object names are created:

$$\left\{\begin{array}{l} <\&s,\ struct\ foo\ *>,\ <s,\ struct\ foo>, \\ <s.b,\ int>,\ <s.b,\ float\ *>,\ <s.c,\ float\ *> \end{array}\right\}$$

And an effective union is found for location s, which is accessed as *struct bar* through pointer r or itself and as $struct\ foo$ through pointer p. The equivalence relation implied by access patterns of the union indicates that $<s.b,\ float\ *>$ and $<s.e,\ int\ *>$ refer

```
struct foo
{
    int a;
    float *b;
    float *c;
} *p;

struct bar
{
    int d;
    int *e;
    float f;
    float *g;
} s, *r;

struct tor
{
    int *val;
    float *ptr;
    void (*fpr)();
} t, *q;

int u, **i;

float w, x, y, z, **f;

void (*fp)(), (*pp)();

void funcA()
{
    q→ptr = &w;
    *(q→ptr) = 1.0;

    f = &(r→g);
    *f = p→c;
    **f = 1.0;
}

void funcB()
{
    q→fpr = &funcA;
    q→val =&u;
    *(q→val) = 1;

    i = &(q→val);
    **i = 1;
}

void funcC()
{
    ...
}

main()
{
    r = &s;
    r→d = 1;
    r→g = &x;

    p = (struct foo * ) r;
    p→a = 2;

    p→b = &y;
    *(p→b) = 1.0;

    p→c = &z;
    *(p→c) = 2.0;

    *(r→e) = 1;

    q = &t;
    q→fpr = &funcB;

    fp = t.fpr;
    fp();

    pp = fp;
    pp = &funcC;
}
```

Figure 6.18: An Example Program for Program Decomposition

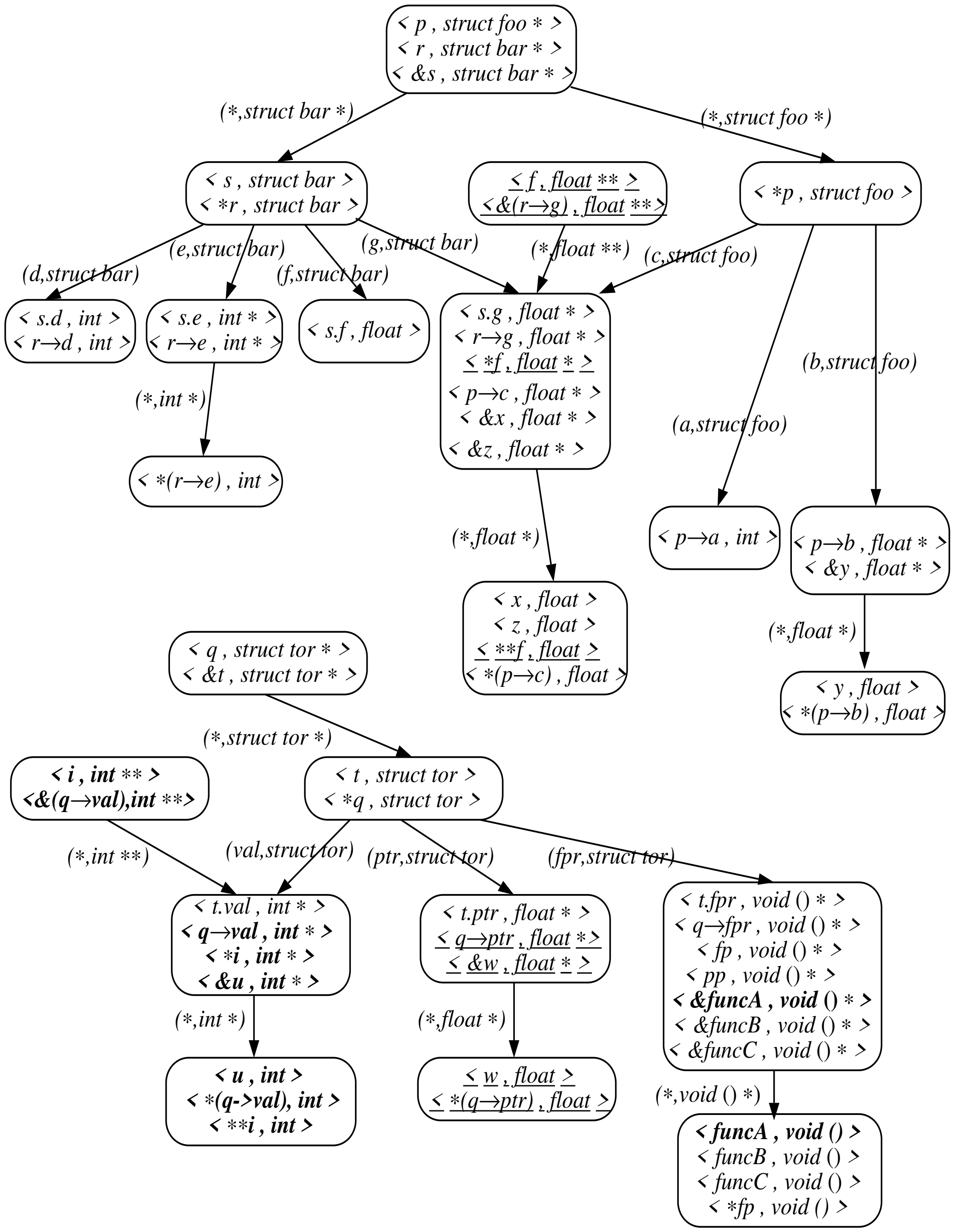


Figure 6.19: The G_{PE} after Points-to Analysis for Indirect Calls

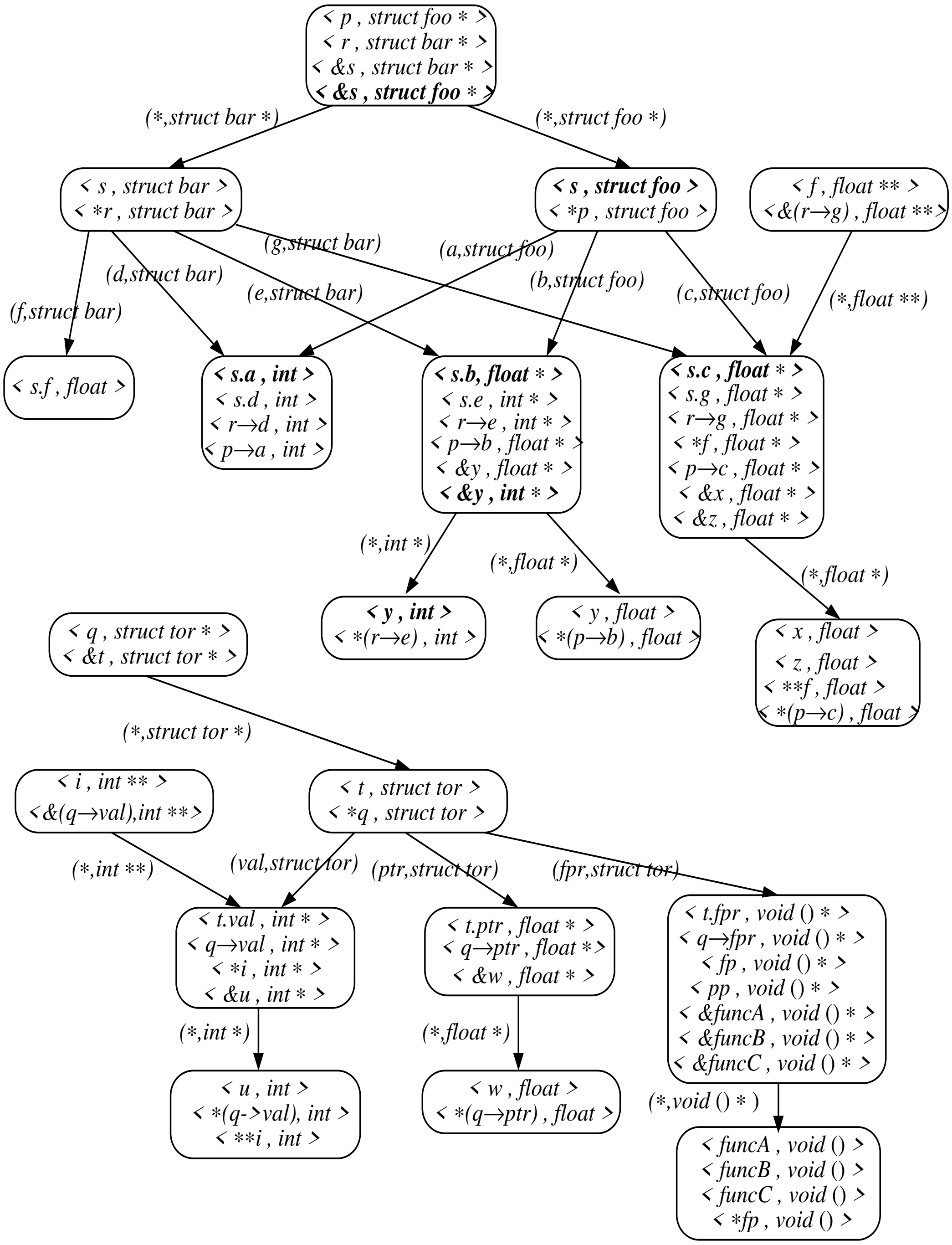


Figure 6.20: The G_{PE} after Points-to Analysis for Type Casting

to the same location. Therefore, in the second iteration, an additional PE equivalence class is found to have names of different pointer types:

$$\left\{ \begin{array}{l} <s.b,\ float\ *>,\ <s.e,\ int\ *>,\ <r{\rightarrow}e,\ int\ *>, \\ <p{\rightarrow}b,\ float\ *>,\ <\&y,\ float\ *> \end{array} \right\}$$

The following names are created:

$$\{\ <\&y,\ int\ *>,\ <y,\ int>\ \}$$

The G_{PE} after resolving type casting is shown in Figure 6.20, where names in boldfaces are added by the points-to analysis. The set B^2_{pts} consists of the following names:

$$\left\{ \begin{array}{l} <\&s,\ struct\ foo\ *>,\ <s,\ struct\ foo>, \\ <s.a,\ int>,\ <s.b,\ float\ *>,\ <s.c,\ float\ *>, \\ <\&y,\ int\ *>,\ <y,\ int> \end{array} \right\}$$

The relation E^3 consists of the following equivalence classes:

$\{\ <s.a,\ int>,\ <s.d,\ int>\ \}$

$\{\ <s.b,\ float\ *>,\ <s.e,\ int\ *>\ \}$

$\{\ <s.c,\ float\ *>\ \}$

$\{\ <s.f,\ float>\ \}$

$\{\ <s.g,\ float\ *>\ \}$

$\{\ <y,\ int>\ \}$

$\{\ <y,\ float>\ \}$

Finally the G_{FA} is shown in Figure 6.21.

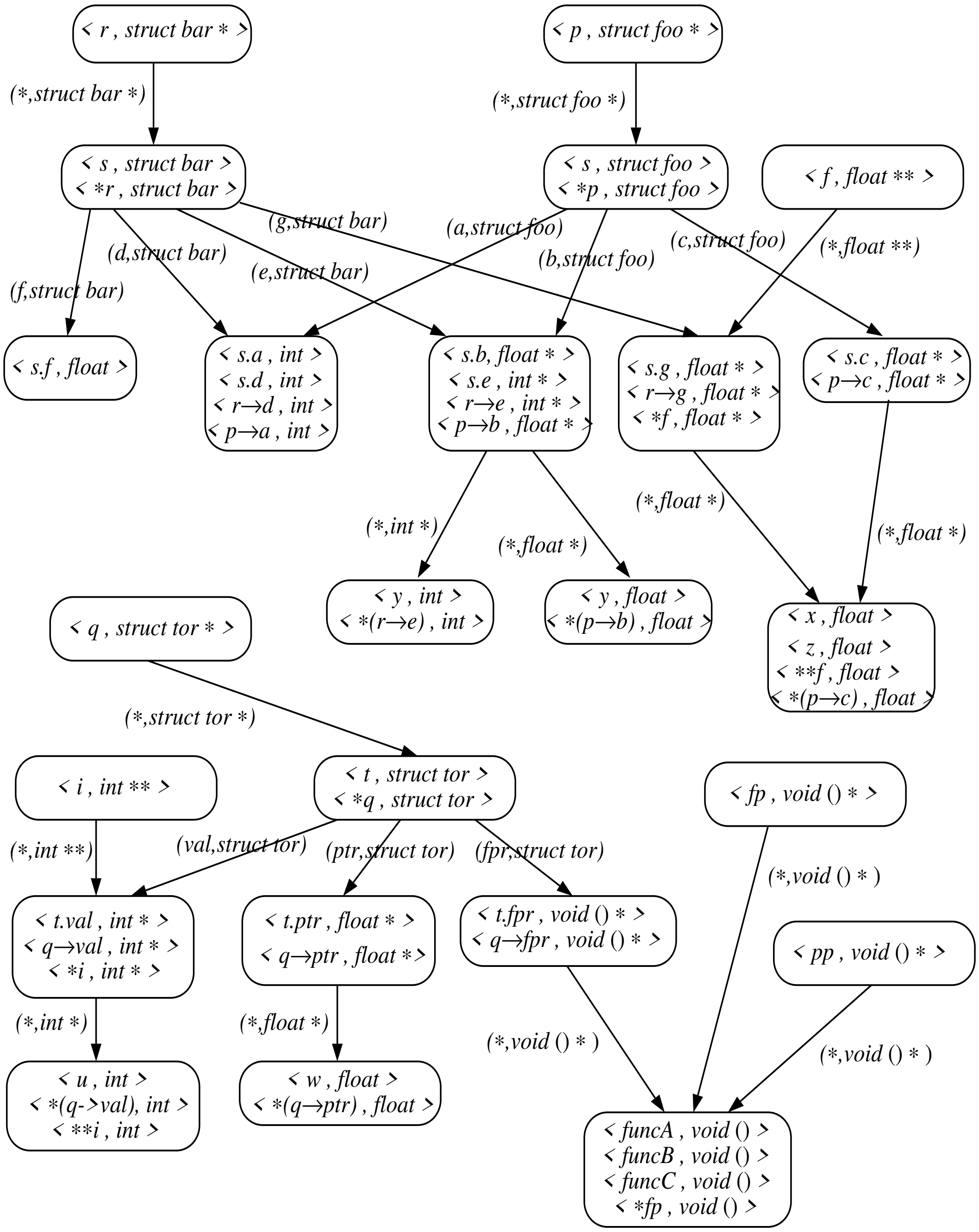


Figure 6.21: The G_{FA} for the Example Program

Chapter 7

Experiments: Combining Aliasing/Points-to Analyses

Our program decomposition enables the application of different aliasing or points-to analysis algorithms to parts of programs. In this chapter, we report the empirical results of applying different aliasing or points-to analyses algorithms to independent sets of pointer-related assignments determined by our program decomposition.

7.1 Test Programs

The test programs will be the same as those in Table 4.1.

7.2 Combined Analysis

Each weakly connected component in our program decomposition has a set of pointer-related assignments associated with it, which is independent of other pointer-related assignments in terms of the aliasing or points-to effects. This set of assignments induces a *program segment* including these assignments and other control statements, which can be analyzed for pointer aliasing independently.

Combined analysis chooses an aliasing or a points-to analysis algorithm for the program segment associated with each weakly connected component in G_{PE} and thus derives the aliasing information for the whole program. For our experiments with three combined analyses, we have used the following analysis algorithms:

1. the Landi/Ryder flow-sensitive and context-sensitive aliasing algorithm [LR92, SRLZ98] (the FS analysis)

2. a flow-insensitive and context-insensitive points-to analysis algorithm presented in Chapter 4 (the PT analysis)

3. a flow-insensitive and context-insensitive aliasing analysis algorithm presented in Chapter 6 (the FA analysis)

Both the FA and PT analyses can handle unions and type casting with some restrictions. One difference between them is that the pointer-related assignments are treated *symmetrically* in the FA analysis; that is, if there is a pointer-related assignment $lhs = rhs$ in a program, the FA analysis will union $*lhs$ and $*rhs$, put the two names in one equivalence class, and thus effectively assume there is also an assignment $rhs = lhs$. We believe this is one of the causes of approximation in the FA analysis.

In each of our combined analyses, we use only two of the three algorithms. Our experiments involve grouping the weakly connected components of G_{PE} into *two* sets, and assigning one algorithm to program segments associated with components in the first set, and another to those corresponding to components in the second set. We have tried two grouping approaches. In one, we applied the FS analysis to the first set and either the FA or the PT analysis to the second set (*FSandFA* and *FSandPT* analyses); in the other, we applied the FA analysis to the first set and the PT analysis to the second set (*FAandPT* analysis). We also used both the FA and PT analyses on the whole program for comparison. We will use the results of Thru-deref MOD/REF problems to compare various analyses.

The overall structure of our implementation is shown in Figure 7.1. The implementation is written in C and compiled by *gcc* with -*O*2. The timing results of each analysis are obtained by averaging over ten runs of the analysis on a SUN SPARCstation 20 running Solaris 2.5.1 with 100M byte physical memory and 350M byte swap space.

7.3 Combining a Flow-sensitive and a Flow-insensitive Analyses

The FS analysis is quite precise, but is sometimes slow. The FA and PT analyses are faster, but may not yield as precise a solution as the FS analysis. For the first two combined analyses, we want to assign the FS analysis to those program segments for which it is suitable, and assign either FA or PT analysis to the other segments. The weakly connected components of G_{PE} are grouped such that any component which

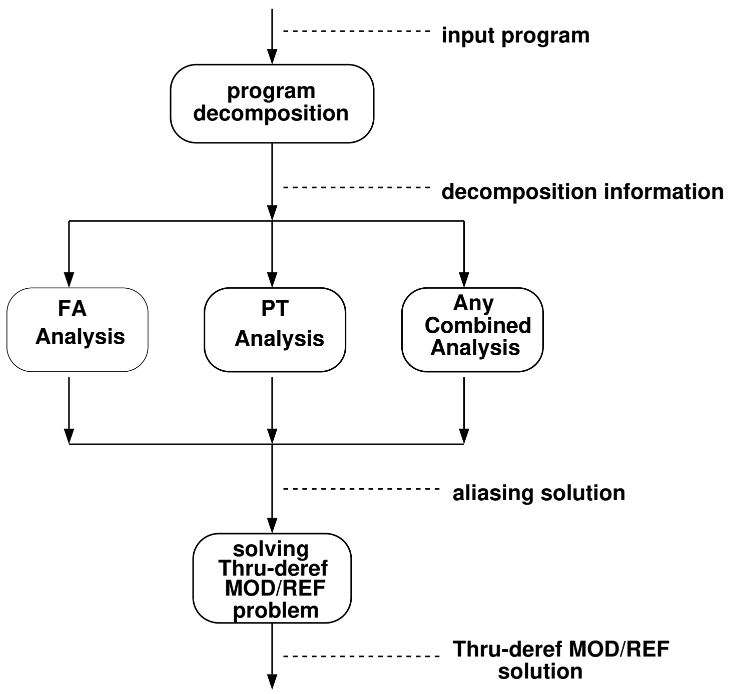


Figure 7.1: The Structure of our Implementation

satisfies *all* of the following conditions is in the *first* set and the other components are in the *second* set.

- There is no cycle in the component.
- There is no location of union type among the names associated with the component.
- There is no type casting in any pointer-related assignment associated with the component.

The presence of recursive data structures seems to be a good predictor of where FS analysis will be expensive as our experience shows that FS is slow in dealing with aliases involving recursive data structures. In our program decomposition, recursive data structures are associated with cyclic components. For efficiency, we will not apply FS analysis to program segments for these components. Also, FS analysis handles unions and type casting differently from the FA and PT analyses; therefore we will use FA or PT analysis for program segments with unions or casting.

In Table 7.1, we show the numbers of components in the first sets of weakly connected components for all test programs. We also present the numbers of pointer-related assignments, object names, variables, Thru-deref MOD/REF sites associated with components in the first sets, and the percentages of those constructs in the first sets over *all* in the programs. A Thru-deref MOD/REF site is associated with a component if the name referenced at the MOD/REF site is in the set of object names associated with the component.

For the second sets of components, we classify them into two kinds, ones with cycles and ones without cycles, but with unions or casting. The ones with cycles are related to recursive data structures. In Table 7.2, we present the numbers of components, pointer-related assignments, object names, and variables associated with each kind of components in the second set. Six programs have components with unions or casting, but without cycles; for these programs, most of the pointer-related assignments are with the components with cycles. The numbers of the Thru-deref MOD/REF sites associated with the second sets of components, are also reported in Table 7.2.

program name	the first set of weakly connected components					
	conn comp	pointer assignment	object name	variables	Thru-deref MOD	Thru-deref REF
chomp	6	10 (6.8%)	24 (13.0%)	12 (16.7%)	7 (18.4%)	9 (15.0%)
loader	13	224 (79.4%)	181 (64.9%)	89 (78.1%)	66 (84.6%)	61 (56.5%)
stanford	15	41 (70.7%)	109 (73.6%)	40 (87.0%)	35 (83.3%)	41 (71.9%)
pokerd	8	143 (82.7%)	175 (83.7%)	57 (89.1%)	52 (88.1%)	123 (93.9%)
sim	21	232 (88.9%)	265 (93.3%)	114 (95.8%)	123 (94.6%)	210 (89.4%)
dineroIII	10	164 (64.6%)	357 (74.7%)	112 (77.2%)	103 (75.7%)	553 (90.1%)
assembler	19	672 (82.6%)	470 (78.2%)	257 (86.0%)	203 (87.1%)	222 (83.5%)
smail	19	353 (39.9%)	279 (38.5%)	126 (35.9%)	87 (53.4%)	40 (19.9%)
archie-client	14	125 (17.3%)	148 (18.7%)	74 (27.5%)	3 (1.8%)	17 (4.9%)
023.eqntott	12	201 (25.2%)	104 (17.5%)	57 (24.8%)	22 (16.7%)	61 (13.8%)
rolo	10	237 (26.4%)	81 (12.2%)	45 (14.8%)	33 (29.7%)	3 (1.7%)
simulator	16	403 (92.0%)	307 (83.0%)	148 (91.9%)	93 (86.9%)	155 (91.7%)
flex	45	569 (82.8%)	474 (79.4%)	214 (78.4%)	232 (93.9%)	349 (89.3%)
agrep	24	392 (59.8%)	382 (53.1%)	234 (73.1%)	92 (56.4%)	244 (67.6%)
bc	8	80 (5.9%)	75 (5.7%)	34 (8.2%)	8 (3.2%)	8 (1.1%)
zip	22	237 (18.8%)	392 (30.6%)	183 (38.9%)	55 (16.6%)	63 (10.3%)
bison	56	1179 (69.1%)	757 (51.8%)	394 (65.4%)	393 (75.1%)	299 (52.8%)
022.li	3	23 (1.6%)	24 (1.7%)	13 (2.5%)	0 (0.0%)	2 (0.6%)
larn	25	1034 (66.8%)	353 (38.5%)	183 (45.3%)	66 (44.9%)	83 (32.8%)
008.espresso	34	767 (10.2%)	529 (8.8%)	250 (9.5%)	216 (14.0%)	199 (4.8%)
T-W-MC	68	1626 (35.7%)	775 (13.8%)	261 (26.8%)	1876 (47.4%)	1688 (24.2%)

Table 7.1: The First Sets of Weakly Connected Components for the FSandFA and FSandPT Analyses

program name	the second set of weakly connected components									
	component with cycles				component with union/casting				Thru-deref MOD	Thru-deref REF
	conn comp	ptr assgn	obj name	var	conn comp	ptr assgn	obj name	var		
chomp	1	136	161	60	0	0	0	0	31	51
loader	2	58	98	25	0	0	0	0	12	47
stanford	1	17	39	6	0	0	0	0	7	16
pokerd	1	30	34	7	0	0	0	0	7	8
sim	1	29	19	5	0	0	0	0	7	25
dineroIII	1	90	121	33	0	0	0	0	33	61
assembler	2	142	131	42	0	0	0	0	30	44
smail	1	532	445	225	0	0	0	0	76	161
archie-client	2	580	595	187	4	17	48	8	162	329
023.eqntott	2	539	410	144	2	58	81	29	110	382
rolo	1	661	582	260	0	0	0	0	78	177
simulator	2	35	63	13	0	0	0	0	14	14
flex	1	109	103	54	1	9	20	5	15	42
agrep	2	264	338	86	0	0	0	0	71	117
bc	4	1265	1245	380	0	0	0	0	243	742
zip	1	849	663	212	3	178	226	76	276	551
bison	7	527	704	208	0	0	0	0	130	267
022.li	1	1460	1399	503	0	0	0	0	139	351
larn	1	498	552	215	1	16	13	6	81	170
008.espresso	2	6785	5501	2385	0	0	0	0	1331	3973
T-W-MC	13	2827	4783	688	1	96	44	26	2081	5288

Table 7.2: The Second Sets of Weakly Connected Components for the FSandFA and FSandPT Analyses

By restricting the FS analysis to the first set of weakly connected components, we have the following combinations:

- FSandFA analysis: apply the FS analysis to program segments for the first set of components and apply the FA analysis to segments for the second set.
- FSandPT analysis: apply the FS analysis to program segments for the first set of components and apply the PT analysis to segments for the second set.

In Figures 7.2 and 7.3, we show the Thru-deref MOD/REF results for these two analyses, FA, and PT. Overall, the results of the FSandFA analysis are better than the results of the FA analysis, especially for *pokerd*, *assembler* and *simulator*. In some cases, the results of the two analyses are very close (e.g., *smail*, *bc*, and 008.*espresso*). We believe the FA part of the FSandFA analysis for these programs yields a very approximate aliasing solution and dominates any improvement made by the FS part. The results of the FSandPT analysis are the most precise, but the results of the PT analysis are almost identical. Also, the PT analysis is shown to be consistently much better than the FA analysis.

To further investigate the impact of the FS part of the two analyses, we present the Thru-deref MOD/REF results for the first sets of components only in Figure 7.4 and 7.5. A Thru-deref MOD/REF site in a program is associated with the first set of components if the dereferenced name is associated with a component in the set. For the program 022.*li*, there is no such Thru-deref MOD site. The results of the FS analysis are better than those of the FA or PT analysis; that is, the FS analysis yields more precise solutions. However, the PT analysis is very close in terms of the Thru-deref MOD/REF results for most programs; this indicates that flow-sensitivity and context-sensitivity may not play a very important role in aliasing analysis for the first sets of components of these programs.

In Figures 7.6 and 7.7, we present the Thru-deref MOD/REF results for the second sets of components to compare the FA and PT analyses for these components. For most programs, the difference is more dramatic and the PT analysis reports half the Thru-deref MOD/REF effects as compared to the FA analysis.

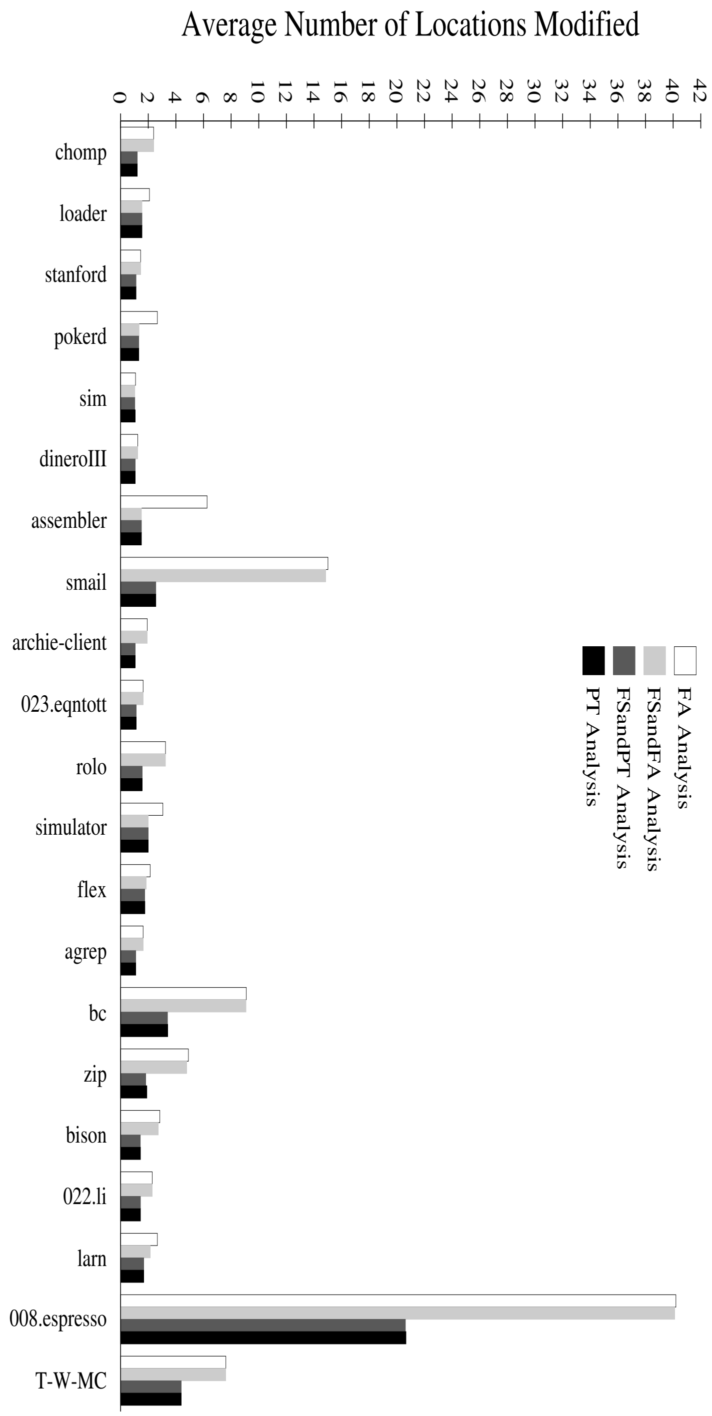


Figure 7.2: Thru-deref MOD Results for the FSandFA and FSandPT Analyses

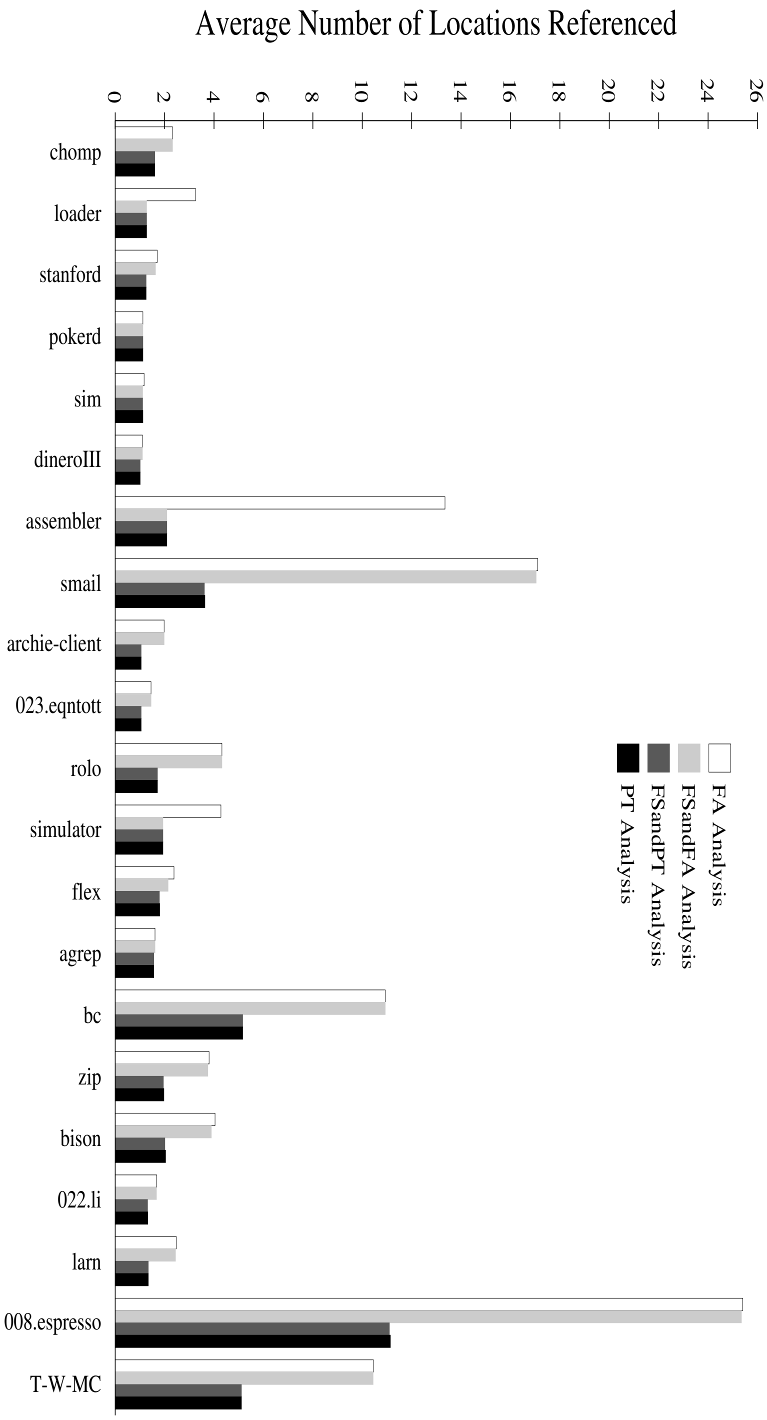


Figure 7.3: Thru-deref REF Results for the FSandFA and FSandPT Analyses

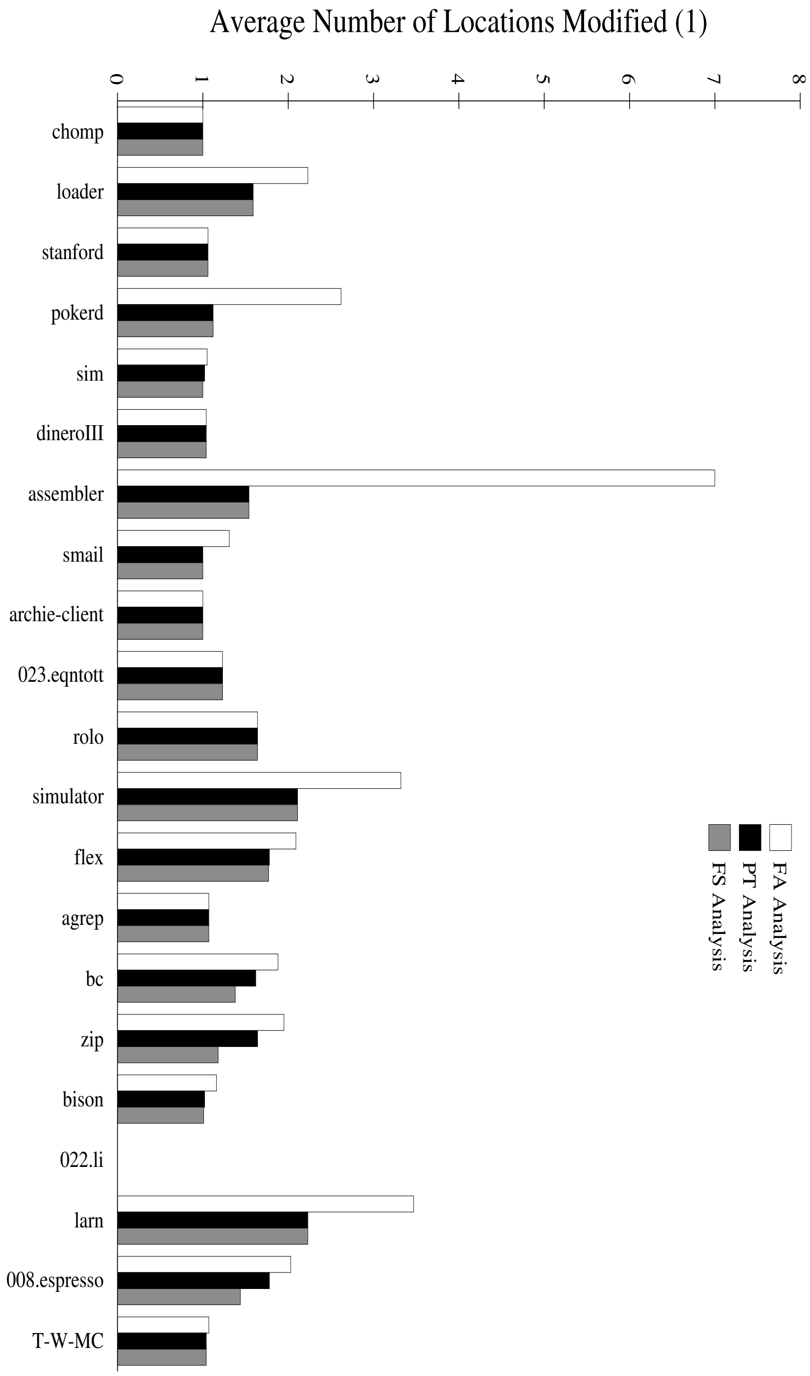


Figure 7.4: Thru-deref MOD Results for the First Sets of Weakly Connected Components (FSandFA and FSandPT Analyses)

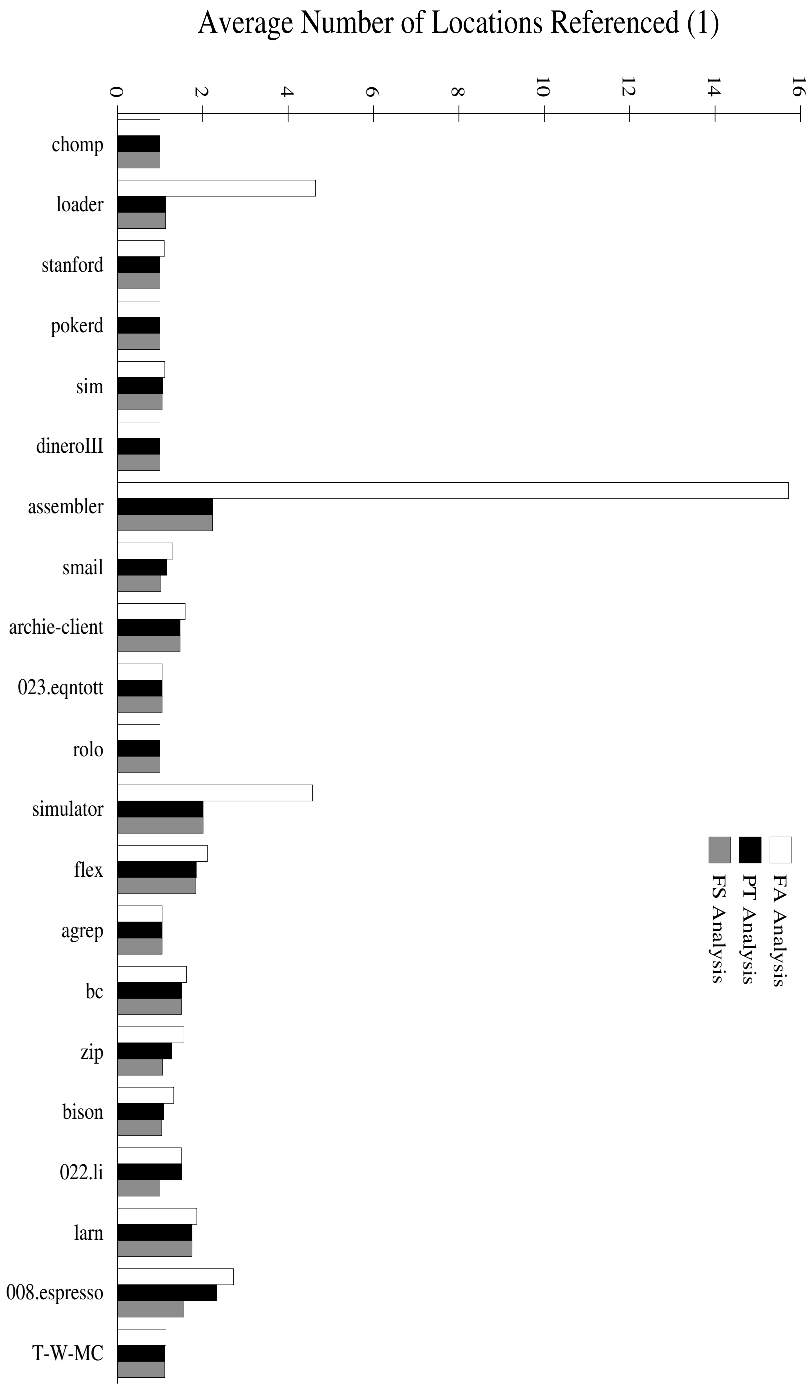


Figure 7.5: Thru-deref REF Results for the First Sets of Weakly Connected Components (FSandFA and FSandPT Analyses)

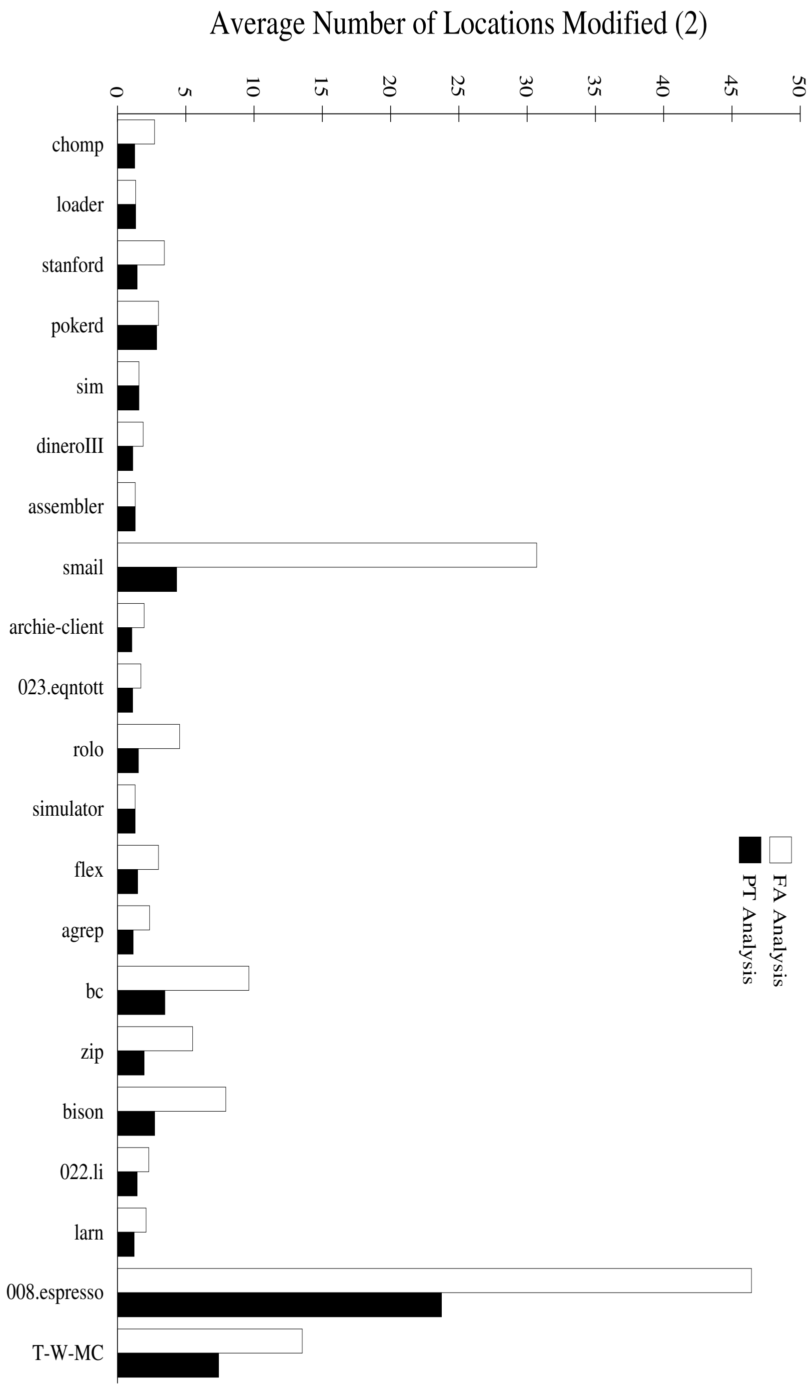


Figure 7.6: Thru-deref MOD Results for the Second Sets of Weakly Connected Components (FSandFA and FSandPT Analyses)

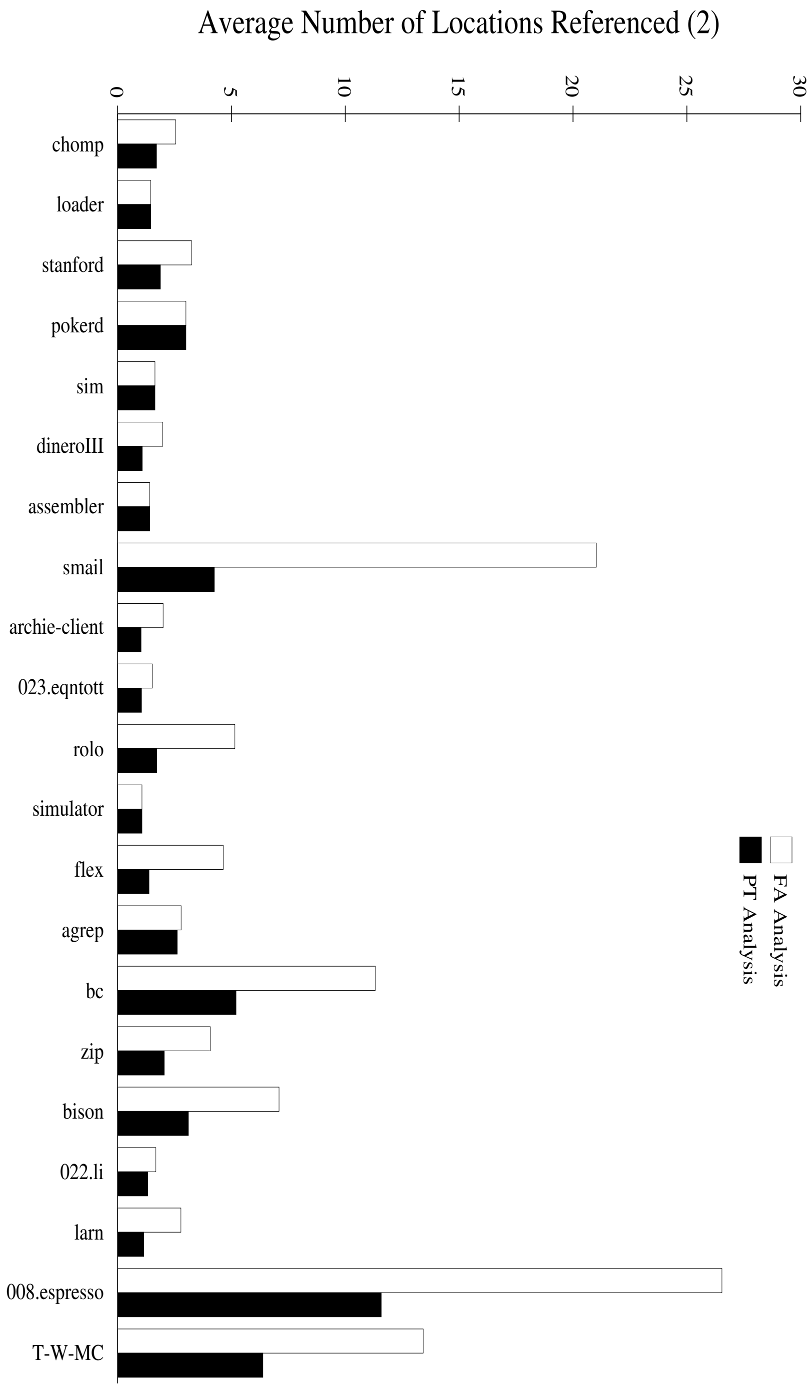


Figure 7.7: Thru-deref REF Results for the Second Sets of Weakly Connected Components (FSandFA and FSandPT Analyses)

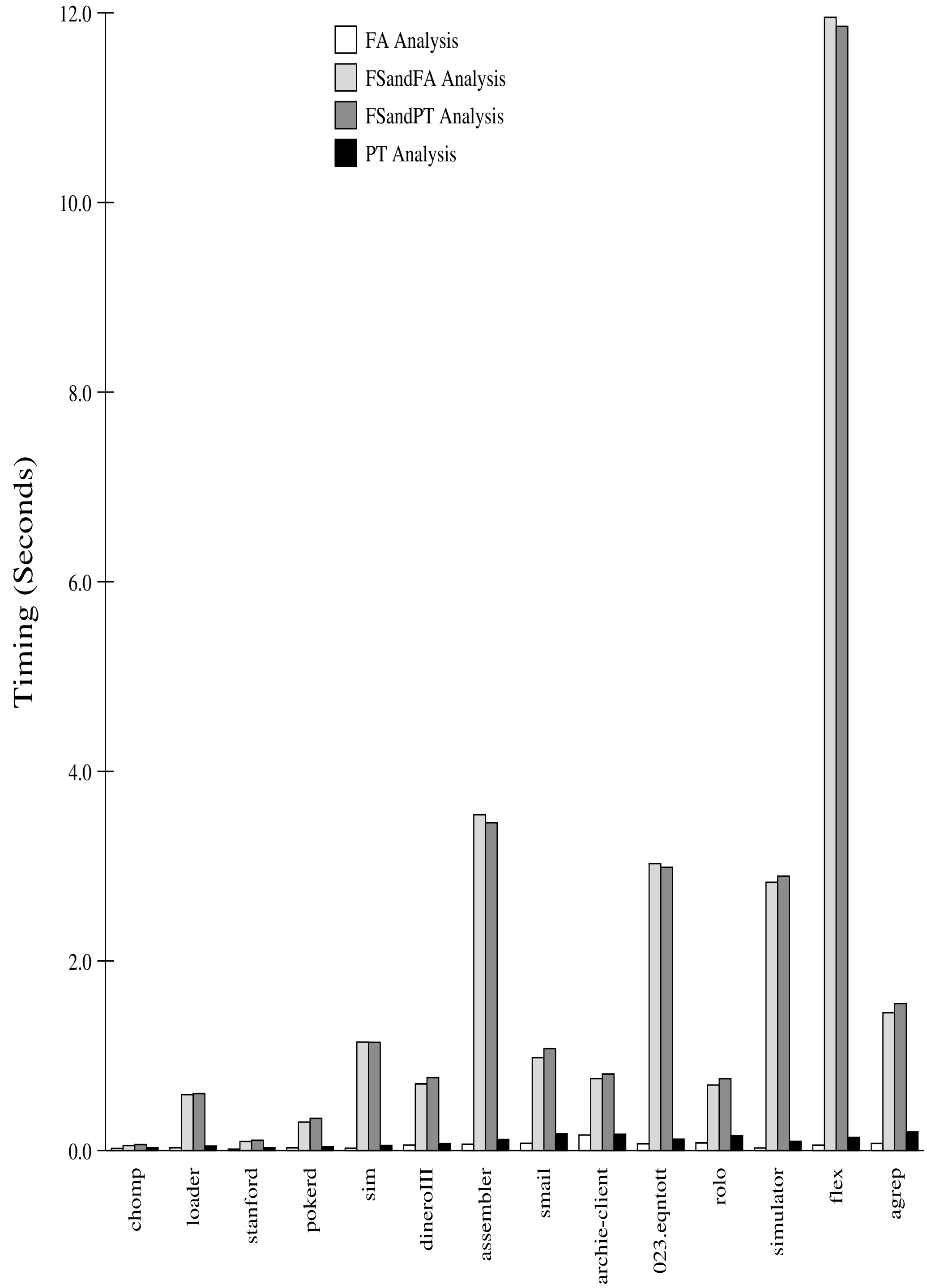


Figure 7.8: Timings for the FSandFA and FSandPT Analyses (Part 1)

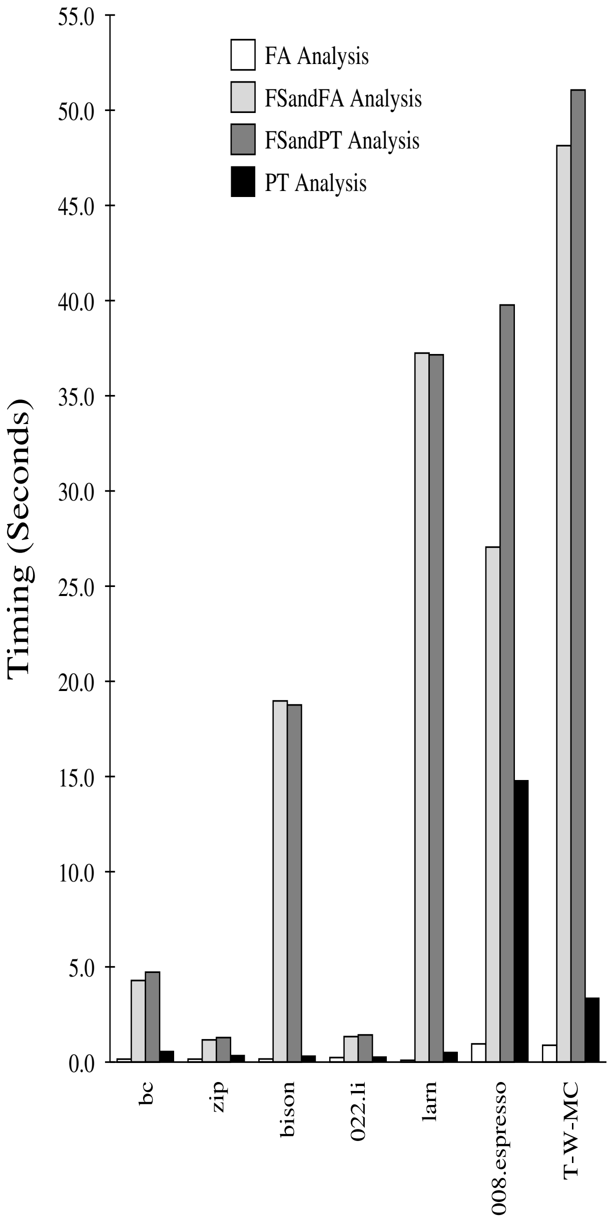


Figure 7.9: Timings for the FSandFA and FSandPT Analyses (Part 2)

In Figures 7.8 and 7.9, we present the timings for the two combined analyses, FA, and PT. The timings do not include the time required for program decomposition. The FSandFA and FSandPT analyses are slower than either the FA or the PT analysis alone, especially for large programs. This is caused by the slowness of the FS part of the analyses. The timings for FSandFA and FSandPT analyses are similar for most programs.

The two combined analyses allow the FS analysis to be used for parts of a program, even if it may be too slow to be applied to the whole program. However, the Thru-deref MOD/REF results show that the precision of the FS analysis may not be worth the cost because the PT analysis is almost as precise.

7.4 Combining Two Flow-insensitive Analyses

The FA analysis is fast[1], but may yield a very approximate aliasing solution because assignments are treated symmetrically. The PT analysis remedies this problem, but is slower. For the third combined analysis, we want to use the FA analysis as much as possible, but for program segments where we expect it to be overly approximate, we will use the PT analysis.

The grouping of weakly connected components is based on the maximum number of names of the form $\&o$ associated with any equivalence class in a component. If there is more than one name of the form $\&o$ in an equivalence class, a name aliased to *any* of the names o will be considered to be aliased to *all* of the names by the FA analysis. Thus, the maximum number of these names associated with any node in a component is a heuristic predictor of the approximation of the FA analysis for the program segment associated with the component.

For this combined analysis, any weakly connected component with the maximum number of names of the form $\&o$ less than or equal to a threshold, is put in the *first* set; all other components are placed in the *second* set. We apply the FA analysis to program segments for the first set of components and apply the PT analysis to segments for the

[1] That is, if we do not include the time required for program decomposition.

program name	first set of weakly connected components					
	conn comp	pointer assignment	object name	variables	Thru-deref MOD site	Thru-deref REF site
chomp	7	146 (100.0%)	185 (100.0%)	72 (100.0%)	38 (100.0%)	60 (100.0%)
loader	13	217 (77.0%)	211 (75.6%)	80 (70.2%)	70 (89.7%)	89 (82.4%)
stanford	16	58 (100.0%)	148 (100.0%)	46 (100.0%)	42 (100.0%)	57 (100.0%)
pokerd	8	56 (32.4%)	72 (34.4%)	29 (45.3%)	13 (22.0%)	42 (32.1%)
sim	22	261 (100.0%)	284 (100.0%)	119 (100.0%)	130 (100.0%)	235 (100.0%)
dineroIII	11	254 (100.0%)	478 (100.0%)	145 (100.0%)	136 (100.0%)	614 (100.0%)
assembler	20	489 (60.1%)	343 (57.1%)	145 (48.5%)	176 (75.5%)	95 (35.7%)
smail	19	353 (39.9%)	279 (38.5%)	126 (35.9%)	87 (53.4%)	40 (19.9%)
archie-client	19	312 (43.2%)	321 (40.6%)	133 (49.4%)	42 (25.5%)	115 (33.2%)
023.eqntott	16	798 (100.0%)	595 (100.0%)	230 (100.0%)	132 (100.0%)	443 (100.0%)
rolo	9	227 (25.3%)	63 (9.5%)	36 (11.8%)	30 (27.0%)	3 (1.7%)
simulator	14	286 (65.3%)	190 (51.4%)	73 (45.3%)	89 (83.2%)	68 (40.2%)
flex	43	491 (71.5%)	391 (65.5%)	174 (63.7%)	214 (86.6%)	325 (83.1%)
agrep	26	656 (100.0%)	720 (100.0%)	320 (100.0%)	163 (100.0%)	361 (100.0%)
bc	10	100 (7.4%)	103 (7.8%)	41 (9.9%)	14 (5.6%)	13 (1.7%)
zip	23	175 (13.8%)	263 (20.5%)	105 (22.3%)	59 (17.8%)	77 (12.5%)
bison	60	1266 (74.2%)	865 (59.2%)	430 (71.4%)	412 (78.8%)	328 (58.0%)
022.li	3	23 (1.6%)	24 (1.7%)	13 (2.5%)	0 (0.0%)	2 (0.6%)
larn	22	443 (28.6%)	220 (24.0%)	115 (28.5%)	52 (35.4%)	78 (30.8%)
008.espresso	33	1487 (19.7%)	1387 (23.0%)	485 (18.4%)	418 (27.0%)	811 (19.4%)
T-W-MC	75	2175 (47.8%)	1669 (29.8%)	403 (41.3%)	2336 (59.0%)	2673 (38.3%)

Table 7.3: The First Sets of Weakly Connected Components for the FAandPT Analysis

second set. The resultant analysis is called *FAandPT*. The idea is that when there are lots of names of the form $\&o$ in one equivalence class, the aliasing solution obtained by the FA analysis for the component is likely to be poor and we want to use the PT analysis to get a more precise solution.

We have chosen 5 for the threshold. By varying the threshold value, we can experiment with a spectrum of analyses. The extreme cases are: (1) the threshold is 0 forcing the first set to be empty, which means we use the PT analysis on the whole program; (2) the threshold is sufficiently large, forcing the second set to be empty, which means we use the FA analysis on the whole program.

In Table 7.3 and 7.4, we show the numbers of components in the two sets of weakly connected components by the above partition. We also present the numbers of pointer-related assignments, object names, variables, Thru-deref MOD/REF sites associated

program	second set of weakly connected components						
	$k=\infty$ comp	other comp	pointer assignment	object name	variables	Thru-deref MOD site	Thru-deref REF site
chomp	0	0	0	0	0	0	0
loader	0	2	65	68	34	8	19
stanford	0	0	0	0	0	0	0
pokerd	0	1	117	137	35	46	89
sim	0	0	0	0	0	0	0
dineroIII	0	0	0	0	0	0	0
assembler	0	1	325	258	154	57	171
smail	1	0	532	445	225	76	161
archie-client	1	0	410	470	136	123	231
023.eqntott	0	0	0	0	0	0	0
rolo	1	1	671	600	269	81	177
simulator	0	4	152	180	88	18	101
flex	1	3	196	206	99	33	66
agrep	0	0	0	0	0	0	0
bc	2	0	1245	1217	373	237	737
zip	1	2	1089	1018	366	272	537
bison	3	0	440	596	172	111	238
022.li	1	0	1460	1399	503	139	351
larn	1	4	1105	698	289	95	175
008.espresso	1	2	6065	4643	2150	1129	3361
T-W-MC	6	1	2374	3933	572	1621	4303

Table 7.4: The Second Sets of Weakly Connected Components for the FAandPT Analysis

with components in the two sets.

In Figures 7.10 and 7.11, we show the Thru-deref MOD/REF results for this combined analysis, FA, and PT. The FAandPT analysis yields results of similar precision to these of the PT analysis. For those programs, where the FA analysis produces approximate solutions (e.g., *assembler*, *smail*, *bc*, 008.*espresso*), the FAandPT analysis is able to improve the precision. For six programs (*chomp*, *stanford*, *sim*, *dineroIII*, 023.*eqntott*, and *agrep*), the FAandPT analysis is effectively the same as the FA analysis because the second sets of components are empty. For these programs, the precision of the FA analysis is comparable to the PT analysis. With the threshold of 5, the first set of components is not empty for any of the test programs, which means the FA part of the combined analysis is always applied for each program.

In Figures 7.12 and 7.13, we present the timings for this combined analysis, FA, and PT. Note that the timing results do not include the time required for program decomposition. In general, the FAandPT analysis is slower than the FA analysis and faster than the PT analysis. In many cases, the FAandPT analysis is very close to the PT analysis in cost. This is because the PT part of the FAandPT analysis is the dominant cost.

These figures indicate that this combined analysis is able to improve the aliasing precision when the FA analysis yields a very approximate solution. The heuristic predictor is effective in identifying program segments, for which the FA analysis is not good. In terms of efficiency, the FAandPT analysis is a little faster than the PT analysis. We plan to conduct more experiments by varying the threshold to explore the tradeoff of efficiency and precision.

7.5 Conclusion

We have experimented with three combined analyses using three different aliasing methods – two that are flow-insensitive and context-insensitive (FA and PT), and one that is flow-sensitive and context-sensitive (FS). The experimental results obtained yielded several conclusions:

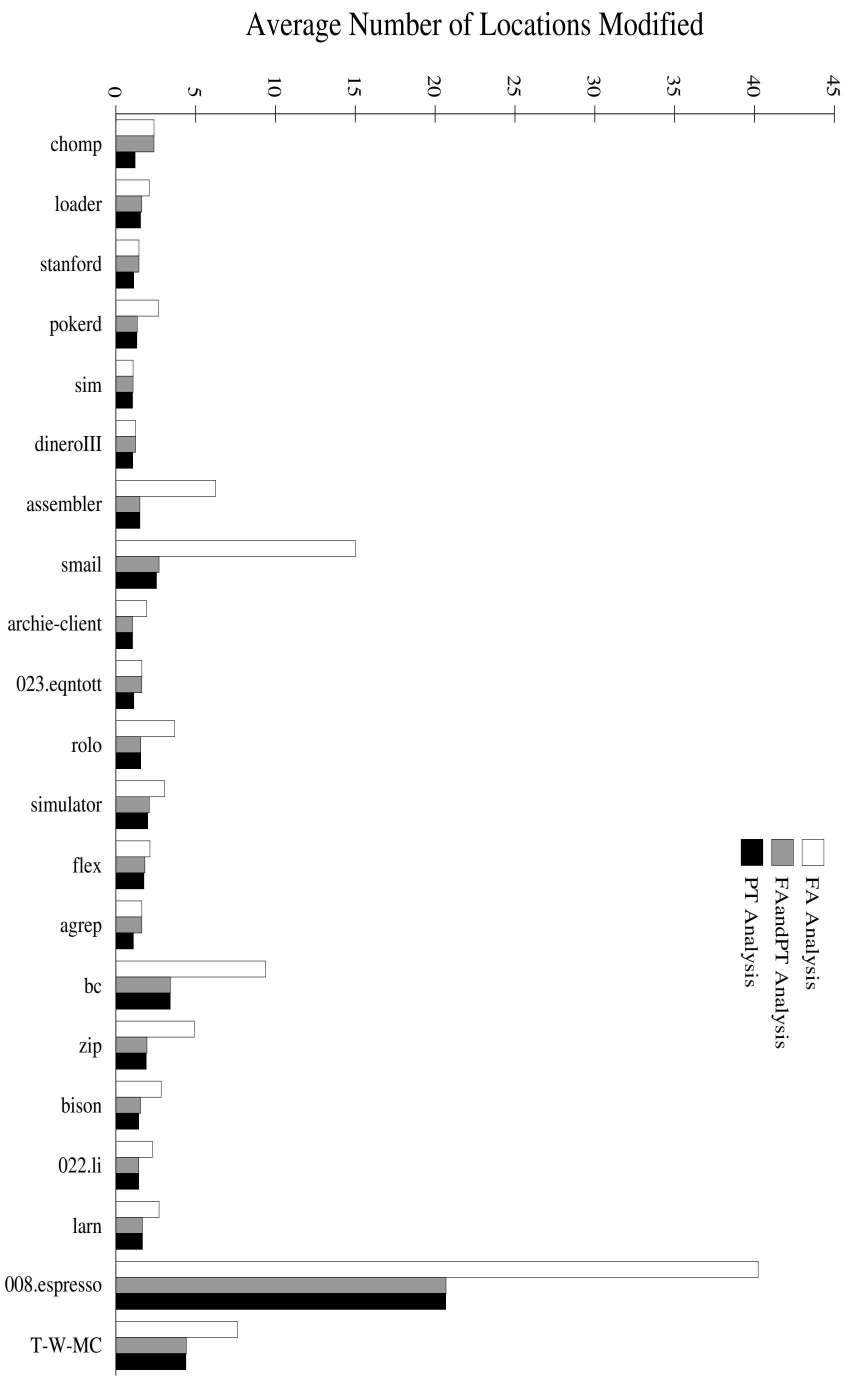


Figure 7.10: Thru-deref MOD Results for the FAandPT Analysis

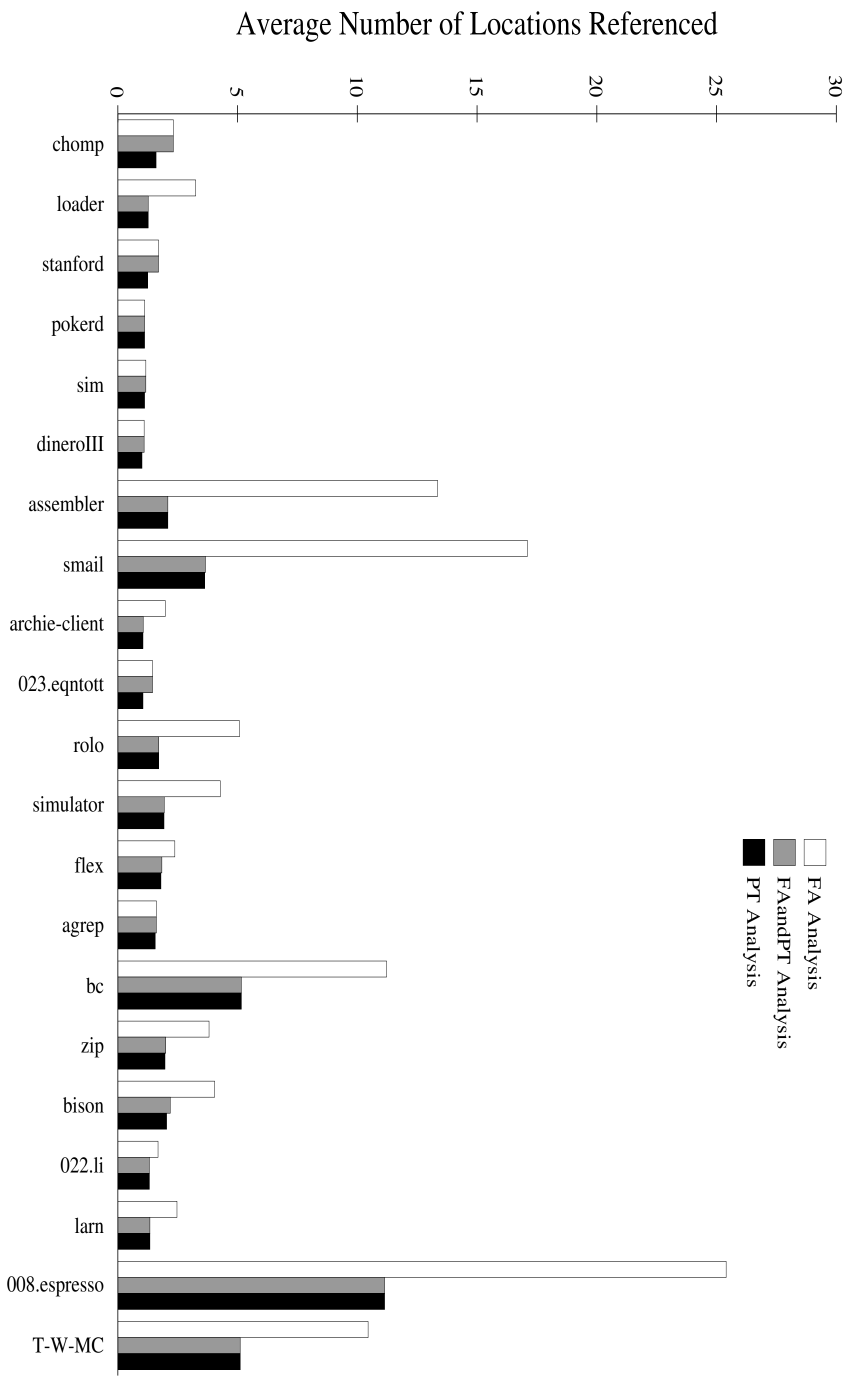


Figure 7.11: Thru-deref REF Results for the FAandPT Analysis

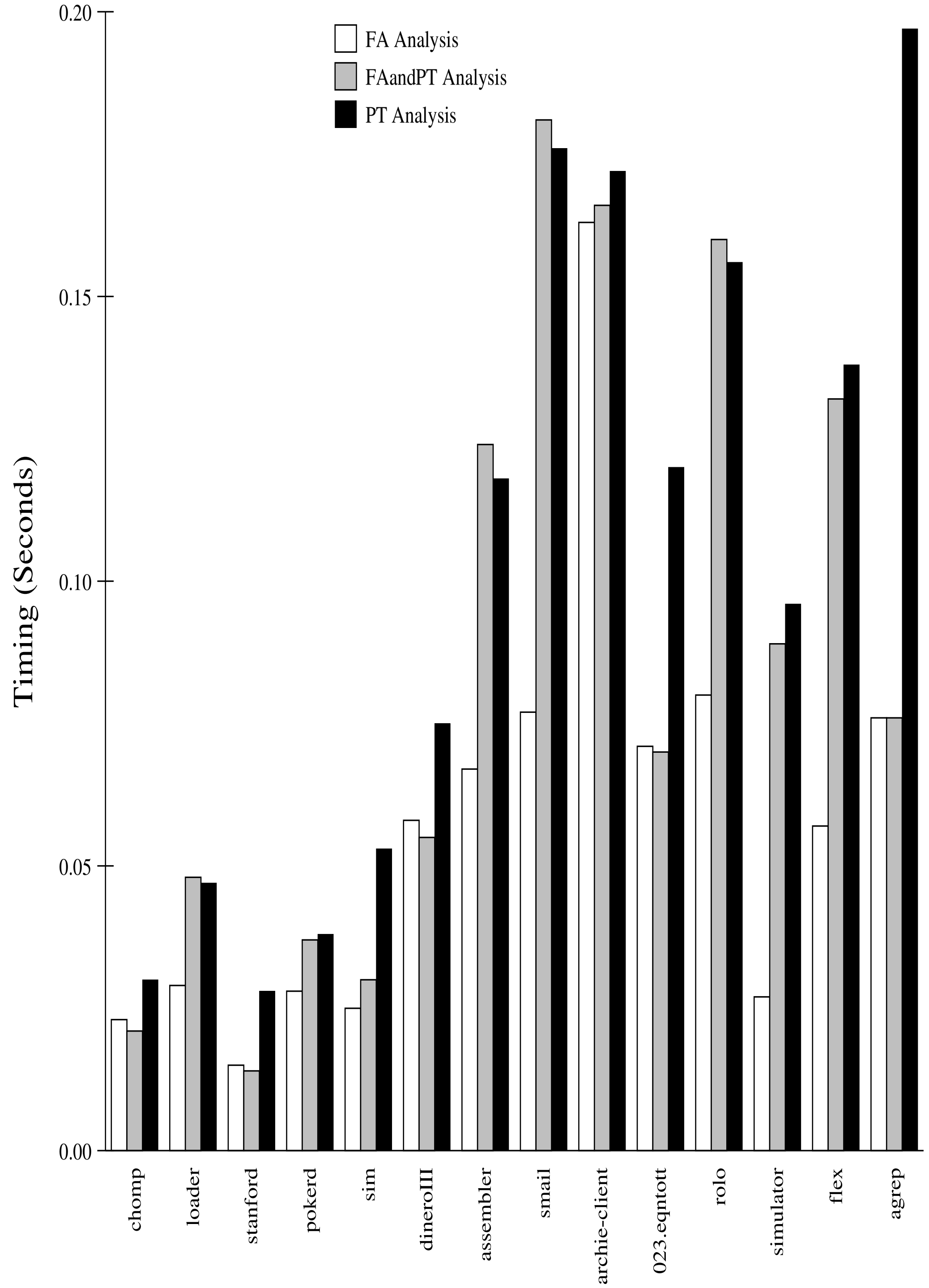


Figure 7.12: Timings for the FAandPT Analysis (Part 1)

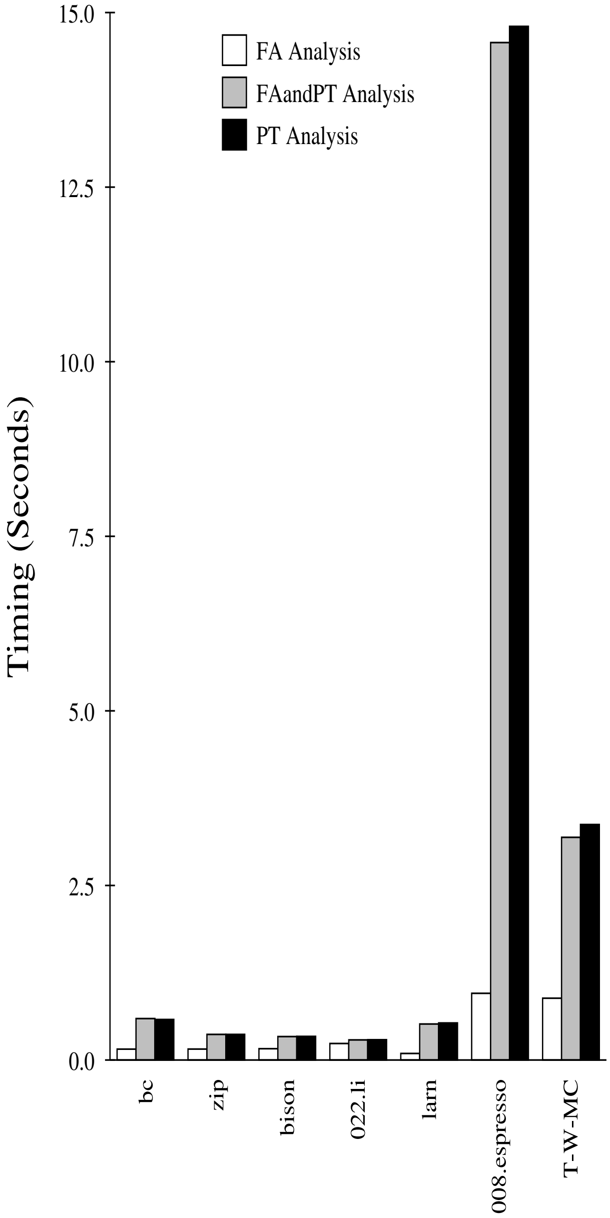


Figure 7.13: Timings for the FAandPT Analysis (Part 2)

- The first two combined analyses verify that FS can be applied to segments of a large program, which may be too large to be completely analyzed by FS.

- The use of FS on program segments based on our heuristics, does not yield increased precision unobtainable by other means (e.g., PT). We need to study other means of choosing where to use FS, for example, based on interesting variables so designated by a programmer.

- For all the programs, especially the large ones such as *larn*, 008.*espresso*, and *T-W-MC*, PT analysis, and combinations with PT are much better than FA analysis at relatively little increased cost. There is evidence that for many dataflow applications, the FA analysis may not yield useful results.

- The heuristic predictor used in the third combined analysis proves more useful than the one used in the first two combined analyses. The FAandPT analysis provides accurate aliasing solutions; although more expensive overall, the precision gained seems worthwhile.

There are two potential areas of future work with the combined analysis. One is to trade off analysis efficiency and precision. We can easily achieve the tradeoff at the program level, but with our program decomposition, we can do this at the program segment level. Another is to study the relationship between program characteristics and the suitable aliasing/points-to analysis algorithm. For instance, our initial experiments show the FA analysis applied to the components involving recursive data structures, unions, or casting (the second sets), yields aliasing information that is much more approximate than the one of the PT analysis. This indicates for more precise aliasing solutions on pointers related to recursive data structures, the FA analysis may not be a good choice. On the other hand, for the components only involving pointers of finite levels of dereferences, our experiments show that the FA analysis seems to yield quite a precise aliasing solution, except for few components, the FS and PT analyses yield aliasing solutions of similar precision. This suggests that for pointers of finite levels of dereferences, the FS analysis may not be necessary.

Chapter 8
Summary and Future Work

8.1 Summary of the Thesis

The work in this thesis provides solutions to several problems of practical pointer aliasing analysis. Here we summarize the results.

We proposed a uniform way of dealing with unions and type casting in aliasing analyses. Our approach to handling unions is consistent with the C standard; our approach to handling type casting as a way of creating unions at run-time is unique and is proved practical by our empirical results. The use of type information to determine common initial sequences makes our approach independent of specific implementations or machines. Being able to handle these features helps to scale analyses to large programs.

Our program decomposition technique is able to separate pointers/assignments into independent sets in terms of their aliasing effects. For pointers/assignments considered related, the technique can show how they are related. This kind of information can potentially be useful for aliasing analysis. We can focus on a set of pointers and analyze their aliases only, without analyzing the whole program; for example, we can analyze indirect calls through function pointers, shapes of some dynamic data structures, or shapes of some dynamically allocated arrays. We can also choose different analysis methods for different sets of pointers/assignments; for example, we can use a flow- and context-insensitive analysis for functions pointers.

Both the points-to analysis (PT) and the aliasing analysis (FA) scale well to large programs as shown by our empirical results; both handle unions and type casting. The PT analysis treats assignments asymmetrically while the FA analysis treats them symmetrically. This provides a good comparison of analysis cost/precision tradeoff.

Our experiments with combined analysis explore possible tradeoff of analysis cost and precision using the results of program decomposition and several analysis methods. We are able to apply the FS analysis to parts of programs even though the analysis is too costly for the whole program. We also show a way to improve the aliasing results for some program segments by using the PT analysis instead of the FA analysis.

We have shown empirical results from our prototype implementation for all the algorithms in this thesis. We believe empirical study on real programs is very important in the research area of program analysis, in particular, the aliasing analysis.

8.2 Future Work

While the results in the thesis are promising, there is more that can be done. Here we discuss some future work.

- Our prototype implementation of the various analyses can be improved to be more efficient in terms of time and space. We plan to optimize the prototype implementation.

- The current algorithm for program decomposition treats procedure calls (other than calls to library functions) in a context-insensitive way. The effect of this is that pointers that otherwise are not related, are now considered related; for instance, if the same insert routine is invoked for two list data structures, our algorithm will consider the two lists related. We want to study the use of context-sensitive (polyvariant) analysis to improve the program decomposition.

- We have some restrictions on unions and type casting; because of them, there are programs that can not be analyzed by our analyses. We want to relax some of these restrictions and hope to apply the program decomposition technique, the aliasing analysis, and the points-to analysis to more programs.

- We want to study empirically what pointer/program characteristics may affect the cost/precision of analysis methods and how to choose effective analysis methods based on these characteristics.

- The graph representation of the equivalence relation for program decomposition can be thought as the result of a *preliminary* shape analysis for all pointers in a program. We want to use the result to identify dynamic data structures or dynamically allocated arrays and study ways to analyze them effectively.

- The methodology of program decomposition and combined analysis can be extended to compile-time analyses other than pointer aliasing. We believe there is potential for achieving scalability as well as investigating tradeoffs between efficiency and precision.

Appendix A

G_{FA} as Representation of the FA Relation

In this appendix, we show that the G_{FA} constructed by the algorithm given in Section 5.4.2 represents the FA relation defined in Section 5.4.1.

In Section 5.4.1, we define the set B as follows:

$$B = \left\{ o \;\middle|\; \begin{array}{l} \text{there is a path from a node } n \text{ to a node } m \text{ in } G_{\text{PE}} \text{ such that the} \\ \text{path is annotated with a sequence of accessors } a_1 a_2 ... a_j \ (j \geq 0) \\ \text{and } o = \mathit{apply}^*(o_1, a_1 a_2 ... a_j), \text{ where } o_1 \in (B_{\text{PE}}(n) \cap B_1) \end{array} \right\}$$

In Section 5.4.2, we present an algorithm that constructs G_{FA} and define the set $O_{\text{FA}}(x)$ for any node x in G_{FA} as follows.

$$O_{\text{FA}}(x) = \left\{ o \;\middle|\; \begin{array}{l} \text{there is a path in } G_{\text{FA}} \text{ from a node } y \text{ to } x, \text{ which is} \\ \text{annotated with a sequence of accessors } a_1 a_2 ... a_j \ (j \geq 0), \\ \text{such that } o = \mathit{apply}^*(o_1, a_1 a_2 ... a_j), \text{ where } o_1 \in B_{\text{FA}}(y) \end{array} \right\}$$

In this appendix, we will first prove that these sets, $O_{\text{FA}}(x)$, where x is a node in G_{FA}, form a partition of the set B and therefore induce an equivalence relation on B (Section A.2); we will then show that the induced equivalence relation is exactly the FA relation defined (Section A.3). Thus G_{FA} constructed by the algorithm is a representation of the FA relation.

A.1 Notations

In order to use inductive proofs, we will consider the effects of the algorithm in Section 5.4.2 as applying a sequence of operations to an initial graph to construct the final graph G_{FA}. The initial graph has one node for each object name in B_1 and has no edges between nodes. The following are four kinds of operations that can be applied:

- add an edge annotated with a member name

  This corresponds to adding a tuple (*member*, *apply*(o,*member*)) to PREFIX(FIND(o)) in the algorithm.

- add an edge annotated with $*$

  This corresponds to adding a tuple $(*, o_1)$ to PREFIX(FIND(o)) in the algorithm.

- union two nodes

  A new node is used to replace the two nodes, such that all incoming edges to any of the two nodes will be to the new node and all outgoing edges from any of the two nodes will be from the new node.

  This corresponds to union(e_1,e_2) in MERGE() routine.

- delete a redundant edge

  An edge between two nodes is *redundant* if there is another edge between the two nodes with the same edge annotation. When two nodes are unioned, there might be redundant edges, which will be deleted.

  This corresponds to the calculation of the new prefix relation in MERGE() routine, which allows only one tuple for each possible edge annotation.

Given G_{PE}, we assume total k operations of the above kinds are applied to construct G_{FA} and we will use G_{FA_i} for the partial G_{FA} after the i^{th} operations, where $1 \leq i \leq k$. G_{FA_0} is the initial graph and G_{FA_k} is G_{FA}.

We will use $B_{\mathrm{FA}_i}(x)$ for the set of object names for a node x in G_{FA_i}. Given G_{FA_i}, where $0 \leq i \leq k$, we define the set $O_{\mathrm{FA}_i}(x)$ for each node x in the graph as follows.

$$O_{\mathrm{FA}_i}(x) = \left\{ o \;\middle|\; \begin{array}{l} \text{there is a path in } G_{\mathrm{FA}_i} \text{ from a node } y \text{ to } x\text{, which is} \\ \text{annotated with a sequence of accessors } a_1 a_2 ... a_j \ (j \geq 0), \\ \text{such that } o = \mathit{apply}^*(o_1, a_1 a_2 ... a_j)\text{, where } o_1 \in B_{\mathrm{FA}_i}(y) \end{array} \right\}$$

By definition, $B_{\mathrm{FA}_i}(x) \subseteq O_{\mathrm{FA}_i}(x)$, where x is any node in G_{FA_i}.

The following are the other notations that are used in this appendix:

- n, n', n_1, n_2, ..., n_j, n_{j+1}, m, m', etc. are nodes in G_{PE}.
- v, w, x, x_1, x_2, ..., x_j, x_{j+1}, y, z, etc. are nodes in G_{FA_i}, where $0 \leq i \leq k$.
- $node_{\mathrm{PE}}(o)$, where $o \in B_0$, is the node n in G_{PE} such that $o \in B_{\mathrm{PE}}(n)$.
- $node_{\mathrm{FA}}(o)$, where $o \in B_1$, is the node x in G_{FA} such that $o \in B_{\mathrm{FA}}(x)$.

A.2 G_{FA} as a Representation of Object Names in B

In this section, we show that the sets, $O_{\mathrm{FA}}(x)$, where x is a node in G_{FA}, constitute a partition of B and therefore induce an equivalence relation on B. In Section A.2.1, we prove that each object name in B is in some $O_{\mathrm{FA}}(x)$, where x is a node in G_{FA}. In Section A.2.2, we prove that $O_{\mathrm{FA}}(x)$, where x is a node in G_{FA}, is a subset of B. In Section A.2.3, we show that $O_{\mathrm{FA}}(x)$ and $O_{\mathrm{FA}}(y)$ do not have any object name in common if x and y are different nodes in G_{FA}.

A.2.1 Object Names in B Are Represented by G_{FA}

In this section, we show that each edge in G_{PE} corresponds to one or more edges in G_{FA} (Lemma A.2.1.1) and each path in G_{PE} corresponds to one or more paths in G_{FA} (Lemma A.2.1.2). We conclude that each object name in B is in $O_{\mathrm{FA}}(x)$, for some node x in G_{FA} (Lemma A.2.1.3).

Lemma A.2.1.1 *Let n and m be any two nodes in G_{PE} such that there is an edge in G_{PE} from n to m, annotated with an accessor a.*
For any object name o in $(B_{\mathrm{PE}}(n) \cap B_1)$[1]*, there is an object name $o_1 \in (B_{\mathrm{PE}}(m) \cap B_1)$, such that there is an edge in G_{FA} from $node_{\mathrm{FA}}(o)$ to $node_{\mathrm{FA}}(o_1)$, that is annotated with the accessor a.*

□

Proof:

[1] $(B_{\mathrm{PE}}(n) \cap B_1)$, where n is a node in G_{PE}, is the set of object names in $B_{\mathrm{PE}}(n)$ that do not contain &.

The accessor a for the edge from n to m in G_{PE} can be either $*$ or a member name. We consider the two cases:

- The accessor a is *member*.

  In this case, o is of structure type and *member* is a member of the structure type.

  Let $o_1 = apply(o{,}member)$. By the construction of G_{PE}, $o_1 \in B_{\mathrm{PE}}(m)$.

  By definition of B_1, $o \in B_1$ and $o_1 \in B_1$. By the algorithm that constructs G_{FA}, for the edge from n to m in G_{PE}, an edge from $node_{\mathrm{FA}}(o)$ to $node_{\mathrm{FA}}(o_1)$, annotated with *member*, is in G_{FA}.

- The accessor a is $*$.

  By the algorithm that constructs G_{FA}, for the edge from n to m in G_{PE}, there is an object name $o_1 \in B_{\mathrm{PE}}(m)$ such that $o_1 = apply(o, *)$, where $o \in B_{\mathrm{PE}}(n)$, and there is an edge from $node_{\mathrm{FA}}(o)$ to $node_{\mathrm{FA}}(o_1)$, annotated with $*$, in G_{FA}.

□

Lemma A.2.1.2 *Let n and m be any two nodes in G_{PE} such that there is a path in G_{PE} from n to m, which is annotated with a sequence of accessors A.*
For any object name o in $(B_{\mathrm{PE}}(n) \cap B_1)$, there is an object name $o_1 \in (B_{\mathrm{PE}}(m) \cap B_1)$ such that there is a path in G_{FA} from $node_{\mathrm{FA}}(o)$ to $node_{\mathrm{FA}}(o_1)$, that is annotated with A.

□

Proof: By Lemma A.2.1.1 and induction on the length of the path from n to m in G_{PE}.

□

Lemma A.2.1.3 *Each object name in B is in some $O_{\mathrm{FA}}(x)$, where x is a node in G_{FA}.*

□

Proof:

Let o be any object name in B.

By definition of B, o corresponds to a path from a node n to a node m in G_{PE}, such that the path is annotated with a sequence of accessors A and $o = apply^*(o_1,A)$, where $o_1 \in (B_{\mathrm{PE}}(n) \cap B_1)$.

By Lemma A.2.1.2, for o_1, there is an object name $o_1^{'} \in (B_{\mathrm{PE}}(m) \cap B_1)$, such that there is a path in G_{FA} from $node_{\mathrm{FA}}(o_1)$ to $node_{\mathrm{FA}}(o_1^{'})$, that is annotated with A.

Let x be $node_{\mathrm{FA}}(o_1^{'})$ in G_{FA}. Then by definition of $O_{\mathrm{FA}}(x)$, $o \in O_{\mathrm{FA}}(x)$.

$\square$

A.2.2 Object Names Represented by G_{FA} Are in B

In this section, we show that each node in G_{FA} can be mapped to a node in G_{PE} so that each edge in G_{FA} corresponds to an edge in G_{PE} (Lemma A.2.2.1) and each path in G_{FA} corresponds to a path in G_{PE} (Lemma A.2.2.2). We conclude that each object name in $O_{\mathrm{FA}}(x)$, where x is any node in G_{FA}, is in B (Lemma A.2.2.3).

By the construction of G_{PE}, if there is an edge from a node n to a node m in G_{PE}, annotated with an accessor a, then there exist $o_1 \in B_{\mathrm{PE}}(n)$ and $o_2 \in B_{\mathrm{PE}}(m)$ such that $o_2 = apply(o_1,a)$. Since we consider only well-typed C programs without type casting, all object names in B_{PE}(n), where n is any node in G_{PE}, have the same type; type-wise an edge from a node to itself in G_{PE} is not possible.

Lemma A.2.2.1 *For any G_{FA_i}, where $0 \leq i \leq k$, the following are true:*

(1) Let x be any node in G_{FA_i} and o be any object name in $B_{\mathrm{FA}_i}(x)$. If $o \in B_{\mathrm{PE}}(n)$, where n is a node in G_{PE}, then $B_{\mathrm{FA}_i}(x) \subseteq B_{\mathrm{PE}}(n)$.

In other words, given a node x in G_{FA_i}, there is a unique node n in G_{PE} such that $B_{\mathrm{FA}_i}(x) \subseteq B_{\mathrm{PE}}(n)$.

(2) Let x and y be any two nodes in G_{FA_i} such that there is an edge in G_{FA_i} from x to y, annotated with an accessor a.

Let n be the node in G_{PE} such that $B_{\mathrm{FA}_i}(x) \subseteq B_{\mathrm{PE}}(n)$ and m be the node in G_{PE} such that $B_{\mathrm{FA}_i}(y) \subseteq B_{\mathrm{PE}}(m)$. Then there is an edge in G_{PE} from n to m, which is annotated with the accessor a.

□

Proof:

We will prove this lemma by induction on the number of operations applied to construct G_{FA}.

Induction Basis:

G_{FA_0} has one node for each object name in B_1 and has no edges. So the lemma holds for G_{FA_0}.

Induction Hypothesis:

The lemma holds for G_{FA_i}, where $i \geq 0$.

Induction Step:

Consider the following case analysis of the $(i+1)^{th}$ operation that is applied to G_{FA_i}:

- An edge annotated with a member name *member*, is added.

  Assume the edge is added from node x to node y in G_{FA_i}. The resulting graph is $G_{\mathrm{FA}_{i+1}}$.

  There is no change to object names associated with nodes in G_{FA_i}, that is, for any node x in $G_{\mathrm{FA}_{i+1}}$, $B_{\mathrm{FA}_{i+1}}(x) = B_{\mathrm{FA}_i}(x)$.

  By the induction hypothesis, (1) holds for any node in $G_{\mathrm{FA}_{i+1}}$ and (2) holds for all edges in $G_{\mathrm{FA}_{i+1}}$ except the one added. We just have to prove that (2) holds for the edge added.

  By the algorithm that constructs G_{FA}, this edge is added because the following are true:

  - There is an edge in G_{PE} from a node n to a node m, annotated with *member*.
  - There is an object name $o_1 \in (B_{\mathrm{PE}}(n) \cap B_1)$ such that

    1. $o_1 \in B_{\mathrm{FA}_i}(x)$
    2. $o_2 = apply(o_1, member) \in B_{\mathrm{FA}_i}(y)$

  By definition, $o_2 \in B_0$. By the construction of G_{PE}, $o_2 \in B_{\mathrm{PE}}(m)$.

Because $o_1 \in B_{\text{PE}}(n)$ and $o_1 \in B_{\text{FA}_i}(x)$, we have $B_{\text{FA}_i}(x) \subseteq B_{\text{PE}}(n)$ by the induction hypothesis.

Because $o_2 \in B_{\text{PE}}(m)$ and $o_2 \in B_{\text{FA}_i}(y)$, we have $B_{\text{FA}_i}(y) \subseteq B_{\text{PE}}(m)$ by the induction hypothesis.

So (2) holds for the edge added.

The lemma holds for $G_{\text{FA}_{i+1}}$ in this case.

- An edge annotated with $*$, is added.

  Assume the edge is added from node x to node y in G_{FA_i}. The resulting graph is $G_{\text{FA}_{i+1}}$.

  For any node x in $G_{\text{FA}_{i+1}}$, $B_{\text{FA}_{i+1}}(x) = B_{\text{FA}_i}(x)$.

  By the induction hypothesis, (1) holds for any node in $G_{\text{FA}_{i+1}}$ and (2) holds for all edges in $G_{\text{FA}_{i+1}}$ except the one added. We just have to prove that (2) holds for the edge added.

  By the algorithm that constructs G_{FA}, this edge is added because the following are true:

  - There is an edge in G_{PE} from a node n to a node m, annotated with $*$.
  - There are two object names, $o_1 \in B_{\text{PE}}(n)$ and $o_2 \in B_{\text{PE}}(m)$, such that
    1. $o_1 \in B_{\text{FA}_i}(x)$
    2. $o_2 \in B_{\text{FA}_i}(y)$
    3. $o_2 = \mathit{apply}(o, *)$, where $o \in B_{\text{PE}}(n)$

  Because $o_1 \in B_{\text{PE}}(n)$ and $o_1 \in B_{\text{FA}_i}(x)$, by the induction hypothesis, we have $B_{\text{FA}_i}(x) \subseteq B_{\text{PE}}(n)$.

  Because $o_2 \in B_{\text{PE}}(m)$ and $o_2 \in B_{\text{FA}_i}(y)$, by the induction hypothesis, we have $B_{\text{FA}_i}(y) \subseteq B_{\text{PE}}(m)$.

  So (2) holds for the edge added.

  The lemma holds for $G_{\text{FA}_{i+1}}$ in this case.

- Two nodes are unioned.

  Assume two nodes in G_{FA_i}, x and y, are unioned. The resulting graph is $G_{\mathrm{FA}_{i+1}}$. The effect of this union is as follows.

  - A new node z in $G_{\mathrm{FA}_{i+1}}$ supersedes x and y.
  - $B_{\mathrm{FA}_{i+1}}(z) = B_{\mathrm{FA}_i}(x) \cup B_{\mathrm{FA}_i}(y)$
  - All incoming edges to x or y in G_{FA_i} will be to z in $G_{\mathrm{FA}_{i+1}}$.
  - All outgoing edges from x or y in G_{FA_i} be from z in $G_{\mathrm{FA}_{i+1}}$.

  By the induction hypothesis, (1) holds for any node in $G_{\mathrm{FA}_{i+1}}$ except z and (2) holds for any edge in $G_{\mathrm{FA}_{i+1}}$ except the ones to z or from z.

  Let n_1 be the node in G_{PE} such that $B_{\mathrm{FA}_i}(x) \subseteq B_{\mathrm{PE}}(n_1)$ and n_2 be the node in G_{PE} such that $B_{\mathrm{FA}_i}(y) \subseteq B_{\mathrm{PE}}(n_2)$.

  We first prove that n_1 and n_2 are the same node in G_{PE}; then we argue that (1) holds for z and (2) holds for any edge to z or from z in $G_{\mathrm{FA}_{i+1}}$.

  Nodes are unioned in two cases, either in the call to MERGE() routine in calculate-FA-relation(), i.e., the top-level call, or in a recursive call to MERGE() routine. We prove that n_1 and n_2 are the same node in G_{PE} by a case analysis:

  - x and y are unioned in the top-level call to MERGE().

    In this case, x and y are unioned because the following are true:

    * There is an edge in G_{PE} from a node n to a node m, annotated with $*$.
    * There are two object names in $B_{\mathrm{PE}}(n)$, b_1 and b_2, such that
      · $b_1' = apply(b_1, *) \in B_{\mathrm{PE}}(m)$
      · $b_2' = apply(b_2, *) \in B_{\mathrm{PE}}(m)$
      · $b_1' \in B_{\mathrm{FA}_i}(x)$
      · $b_2' \in B_{\mathrm{FA}_i}(y)$

    Because $b_1' \in B_{\mathrm{PE}}(m)$ and $b_1' \in B_{\mathrm{FA}_i}(x)$, by the induction hypothesis, we have $B_{\mathrm{FA}_i}(x) \subseteq B_{\mathrm{PE}}(m)$.

Because $b_2^{'} \in B_{\mathrm{PE}}(m)$ and $b_2^{'} \in B_{\mathrm{FA}_i}(y)$, by the induction hypothesis, we have $B_{\mathrm{FA}_i}(y) \subseteq B_{\mathrm{PE}}(m)$.

So n_1 is node m and n_2 is node m.

– x and y are unioned in a recursive call to MERGE().

In this case, x and y are unioned because there is a node w in G_{FA_i} such that there are two edges, one from w to x and another from w to y in G_{FA_i}, both of which are annotated with an accessor a.

Let $n^{'}$ be the node in G_{PE} such that $B_{\mathrm{FA}_i}(w) \subseteq B_{\mathrm{PE}}(n^{'})$.

Because of the edge from w to x in G_{FA_i}, by the induction hypothesis, there is an edge in G_{PE} from node $n^{'}$ to node n_1, which is annotated with a.

Because of the edge from w to y in G_{FA_i}, by the induction hypothesis, there is an edge in G_{PE} from node $n^{'}$ to node n_2, which is annotated with a.

By construction of G_{PE}, there can be only one outgoing edge from $n^{'}$ in G_{PE}, annotated with a. Thus n_1 and n_2 must be the same node.

So we have proved that n_1 and n_2 are the same node in G_{PE}. Let n be the node; we have $B_{\mathrm{FA}_i}(x) \subseteq B_{\mathrm{FA}_i}(n)$ and $B_{\mathrm{FA}_i}(y) \subseteq B_{\mathrm{FA}_i}(n)$.

Because $B_{\mathrm{FA}_{i+1}}(z) = (B_{\mathrm{FA}_i}(x) \cup B_{\mathrm{FA}_i}(y))$, we have $B_{\mathrm{FA}_{i+1}}(z) \subseteq B_{\mathrm{PE}}(n)$, that is, (1) holds for the node z in $G_{\mathrm{FA}_{i+1}}$.

Now we prove that (2) holds for any edge to z in $G_{\mathrm{FA}_{i+1}}$. Similarly, we can show that (2) holds for any edge from z in $G_{\mathrm{FA}_{i+1}}$.

Assume there is an edge from a node v to z in $G_{\mathrm{FA}_{i+1}}$. By the induction hypothesis, each edge in G_{FA_i} corresponds to an edge in G_{PE}. Since there is no edge from a node to itself in G_{PE}, there is no edge from x to y in G_{FA_i}; otherwise there would have been an edge from n to n in G_{PE}. Also there is no edge from a node to itself in G_{FA_i} either. So v is not the same node as z.

This edge from v to z in $G_{\mathrm{FA}_{i+1}}$ must correspond to an edge from v to x or an edge from v to y in G_{FA_i}. Without loss of generality, assume it corresponds to an edge from v to x in G_{FA_i}.

Let m' be the node in G_{PE} such that $B_{\mathrm{FA}_i}(v) \subseteq B_{\mathrm{PE}}(m')$. Because of the edge from v to x in G_{FA_i}, by the induction hypothesis, there is an edge in G_{PE} from m' to n. So (2) holds for this edge to z in $G_{\mathrm{FA}_{i+1}}$.

The lemma holds for $G_{\mathrm{FA}_{i+1}}$ in this case.

- A redundant edge is deleted.

  Because the edge is redundant, both (1) and (2) hold for the resulting graph, $G_{\mathrm{FA}_{i+1}}$.

So we have proved that no matter what the $(i+1)^{th}$ operation is, the lemma holds for $G_{\mathrm{FA}_{i+1}}$.

□

Lemma A.2.2.2 *For any G_{FA_i}, where $0 \leq i \leq k$, let x and y be any two nodes in G_{FA_i} such that there is a path in G_{FA_i} from x to y, annotated with a sequence of accessors A.*

Let n be the node in G_{PE} such that $B_{\mathrm{FA}_i}(x) \subseteq B_{\mathrm{PE}}(n)$ and m be the node in G_{PE} such that $B_{\mathrm{FA}_i}(y) \subseteq B_{\mathrm{PE}}(m)$. Then there is a path in G_{PE} from n to m, that is annotated with A.

□

Proof: By Lemma A.2.2.1 and induction on the length of the path from x to y in G_{FA_i}.

□

Lemma A.2.2.3 *For any G_{FA_i}, where $0 \leq i \leq k$, let x and y be any two nodes in G_{FA_i} such that there is a path in G_{FA_i} from x to y, annotated with a sequence of accessors A. For any $o \in B_{\mathrm{FA}}(x)$, apply$^*(o,A) \in B$. In other words, $O_{\mathrm{FA}_i}(y) \subseteq B$, where y is any node in G_{FA_i}.*

□

This lemma is a result of Lemma A.2.2.2.

□

By this lemma, $O_{\mathrm{FA}}(x) \subseteq B$, for any node x in G_{FA},

A.2.3 Object Names Represented by G_{FA} Partitions B

In this section, we show that $O_{\mathrm{FA}}(x)$ and $O_{\mathrm{FA}}(y)$, for two different nodes x and y in G_{FA}, do not have any object name in common. Together with the results in the last two sections,

Lemma A.2.3.1 *Let o be any object name in B_1. Let a be an accessor such that o_1 = apply$(o,a) \in B_1$. There is an edge in G_{FA} from node$_{\mathrm{FA}}(o)$ to node$_{\mathrm{FA}}(o_1)$, annotated with the accessor a.*

□

Proof:

Since $o \in B_1$, $o_1 \in B_1$ and $B_1 \subseteq B_0$, we have $o \in B_0$ and $o_1 \in B_0$. Let n be $node_{\mathrm{PE}}(o)$ and m be $node_{\mathrm{PE}}(o_1)$. By construction of G_{PE}, there is an edge in G_{PE} from n to m, annotated with the accessor a.

The accessor a can be either $*$ or a member name. We consider the two cases:

- The accessor a is *member*.

  By the algorithm that constructs G_{FA}, for the edge from n to m in G_{PE}, an edge from $node_{\mathrm{FA}}(o)$ to $node_{\mathrm{FA}}(o_1)$, annotated with *member*, is in G_{FA}.

- The accessor a is $*$.

  By the algorithm that constructs G_{FA}, for the edge from n to m in G_{PE}, all object names o_1' such that $o_1' \in B_{\mathrm{PE}}(m)$ such that $o_1' = apply(o', *)$, where $o' \in B_{\mathrm{PE}}(n)$, will be collected; nodes for these object names will be merged. In other words, there is a node x in G_{FA} such that all object names collected are in $B_{\mathrm{FA}}(x)$. Clearly $o_1 \in B_{\mathrm{FA}}(x)$, that is, x is $node_{\mathrm{FA}}(o_1)$.

  By the algorithm that constructs G_{FA}, there is an edge from $node_{\mathrm{FA}}(o)$ to x (i.e., $node_{\mathrm{FA}}(o_1)$), annotated with $*$, in G_{FA}.

□

Lemma A.2.3.2 *Let o be any object name in B_1. Let A be a sequence of accessors such that $o_1 = apply^*(o,A) \in B_1$. There is a path in G_{FA} from $node_{\mathrm{FA}}(o)$ to $node_{\mathrm{FA}}(o_1)$, annotated with A.*

□

Proof: By Lemma A.2.3.1 and induction on the number of accessors in A.

□

Lemma A.2.3.3 *Let o be any object name in B_1. If there is a path in G_{FA} from $node_{\mathrm{FA}}(o)$ to a node x, such that the path is annotated with a sequence of accessors A and $o_1 = apply^*(o,A)$ is in B_1, then $o_1 \in B_{\mathrm{FA}}(x)$.*

□

Proof:

By Lemma A.2.3.2, there is a path in G_{FA} from $node_{\mathrm{FA}}(o)$ to $node_{\mathrm{FA}}(o_1)$, annotated with A.

By the algorithm that constructs G_{FA}, there can be only one path in G_{FA} from $node_{\mathrm{FA}}(o)$, which is annotated with A, $node_{\mathrm{FA}}(o_1)$ must be the node x. $node_{\mathrm{FA}}(o_1)$ is node x, that is, $o_1 \in B_{\mathrm{FA}}(x)$.

□

Lemma A.2.3.4 *For any two different nodes in G_{FA}, x and y, $O_{\mathrm{FA}}(x) \cap O_{\mathrm{FA}}(y) = \emptyset$.*

□

Proof: By contradiction.

Assume that x and y are *different* nodes in G_{FA} such that there is an object name o in both $O_{\mathrm{FA}}(x)$ and $O_{\mathrm{FA}}(y)$.

By definition of $O_{\mathrm{FA}}(x)$, o corresponds to a path in G_{FA} from a node z_1 to x, annotated with a sequence of accessors A_1, such that $o = apply^*(o_1,A_1)$, where $o_1 \in B_{\mathrm{FA}}(z_1)$.

By definition of $O_{\mathrm{FA}}(y)$, o corresponds to a path in G_{FA} from a node z_2 to y, annotated with a sequence of accessors A_2, such that $o = apply^*(o_2,A_2)$, where $o_2 \in B_{\mathrm{FA}}(z_2)$.

Since $apply^*(o_1,A_1)$ and $apply^*(o_2,A_2)$ are the same object name, there exist a sequence of accessors A such that either $o_1 = apply^*(o_2,A)$ or $o_2 = apply^*(o_1,A)$.

Without loss of generality, we assume $o_2 = apply^*(o_1,A)$.

Because $apply^*(o_1,A_1) = apply^*(o_2,A_2)$, we have $A_1 = A.A_2$.

Because $o_2 = apply^*(o_1,A,\ o_1) \in B$ and $o_2 \in B$, by Lemma A.2.3.2, there is a path in G_{FA} from z_1 (i.e., $node_{\mathrm{FA}}(o_1)$) to z_2 (i.e., $node_{\mathrm{FA}}(o_2)$), annotated with A.

So we have two paths from z_1 annotated with the same sequence of accessors A, one from z_1 to x and another from z_1 through z_2 to y. By the algorithm that constructs G_{FA}, there can be only one path from z_1 annotated with A; so x and y must be the same node. This contradicts our assumption that they are different nodes.

Therefore, the lemma holds.

□

A.3 G_{FA} as a Representation of the FA Relation

In Section A.2, we have shown that the sets, $O_{\mathrm{FA}}(x)$, where x is a node in G_{FA}, induce an equivalence relation on B. In this section, we prove that the equivalence relation is the FA relation.

First, we define the equivalence relation.

Definition A.3.0.1 *The sets, $O_{\mathrm{FA}}(x)$, where x is a node in G_{FA}, induce an equivalence relation on B. We call it the $EQ_{G_{\mathrm{FA}}}$ relation.*

□

Now, we show that the $\mathrm{EQ}_{G_{\mathrm{FA}}}$ relation contains the FA relation.

Lemma A.3.0.1 *Let $n_1 n_2 ... n_j n_{j+1}$ be a path in G_{PE}, annotated with a sequence of accessors, A, such that NumOfDerefs(A) $\geq$ 1. Let o_1 and o_2 be any two object names in $B_{\mathrm{PE}}(n_1)$. Let $o_1' = apply^*(o_1,A)$ and $o_2' = apply^*(o_2,A)$. There is a node $x \in G_{\mathrm{FA}}$ such that $B_{\mathrm{FA}}(x) \subseteq B_{\mathrm{PE}}(n_{j+1})$, $o_1' \in O_{\mathrm{FA}}(x)$, and $o_2' \in O_{\mathrm{FA}}(x)$.*

□

Proof: By induction on the length of the path $n_1 n_2 ... n_j n_{j+1}$ in G_{PE}.

Note that the path length is at least one since NumOfDerefs$(A) \geq 1$.

Induction Basis:

The path length is 1, that is, it consists of one edge in G_{PE} from n_1 to n_2.

Since the annotation for the path has at least one $*$, the edge from n_1 to n_2 must be annotated with $*$, $o_1^{'} = \mathit{apply}(o_1, *)$ and $o_2^{'} = \mathit{apply}(o_2, *)$.

We proceed by a case analysis of whether or not o_1 and o_2 contain &:

- Both o_1 and o_2 contain &.

  In this case, $o_1 = \&o_1^{'}$ and $o_2 = \&o_2^{'}$. $o_1^{'}$ and $o_2^{'}$ do not contain &.

  Because both o_1 and o_2 are in B_0, by definition of B_0, $o_1^{'} \in B_0$ and $o_2^{'} \in B_0$.

  By construction of G_{PE}, $o_1^{'} \in B_{\text{PE}}(n_2)$ and $o_2^{'} \in B_{\text{PE}}(n_2)$.

  Since $o_1^{'}$ and $o_2^{'}$ do not contain &, $o_1^{'} \in B_1$ and $o_2^{'} \in B_1$. Because of the edge from n_1 to n_2 in G_{PE}, by the algorithm that constructs G_{FA}, the node for $o_1^{'}$ and the node for $o_2^{'}$ are unioned. In other words, there is a node x in G_{FA} such that both $o_1^{'}$ and $o_2^{'}$ are in $B_{\text{FA}}(x)$.

  By definition of $O_{\text{FA}}(x)$, $B_{\text{FA}}(x) \subseteq O_{\text{FA}}(x)$; therefore $o_1^{'} \in O_{\text{FA}}(x)$ and $o_2^{'} \in O_{\text{FA}}(x)$.

  Since $o_1^{'} \in B_{\text{FA}}(x)$ and $o_1^{'} \in B_{\text{PE}}(n_2)$, by Lemma A.2.2.1, $B_{\text{FA}}(x) \subseteq B_{\text{PE}}(n_2)$.

- Only one of o_1 and o_2 contains &.

  Without loss of generality, assume o_1 contains & and o_2 does not. In this case, $o_1 = \&o_1^{'}$.

  Because $o_1 \in B_0$, by definition of B_0, $o_1^{'} \in B_0$.

  By construction of G_{PE}, $o_1^{'} \in B_{\text{PE}}(n_2)$.

  Since neither $o_1^{'}$ nor o_2 contain &, $o_1^{'} \in B_1$ and $o_2 \in B_1$. Because of the edge from n_1 to n_2 in G_{PE}, by the algorithm that constructs G_{FA}, an edge will be added in G_{FA} from $\mathit{node}_{\text{FA}}(o_2)$ to $\mathit{node}_{\text{FA}}(o_1^{'})$, annotated with $*$. Let x be $\mathit{node}_{\text{FA}}(o_1^{'})$ and y be $\mathit{node}_{\text{FA}}(o_2)$; in other words, $o_1^{'} \in B_{\text{FA}}(x)$ and $o_2 \in B_{\text{FA}}(y)$.

  By definition of $O_{\text{FA}}(x)$, $\mathit{apply}(o_2, *) \in O_{\text{FA}}(x)$, i.e., $o_2^{'} \in O_{\text{FA}}(x)$.

Because $o_1^{'} \in B_{\text{FA}}(x)$, by definition of $O_{\text{FA}}(x)$, $o_1^{'} \in O_{\text{FA}}(x)$.

Since $o_1^{'} \in B_{\text{FA}}(x)$ and $o_1^{'} \in B_{\text{PE}}(n_2)$, by Lemma A.2.2.1, $B_{\text{FA}}(x) \subseteq B_{\text{PE}}(n_2)$.

- Neither o_1 nor o_2 contains &.

  In this case, we have both $o_1 \in B_1$ and $o_2 \in B_1$. Because of the edge from n_1 to n_2 in G_{PE}, by the algorithm that constructs G_{FA}, the following will be true in G_{FA}:

  - Nodes for any object name in the following set:

    $$\{\ o \mid o \in B_{\text{PE}}(n_2) \text{ and } o = \mathit{apply}(o^{'}, *\,), \text{ where } o^{'} \in B_{\text{PE}}(n_1)\ \}$$

    will be unioned. By construction of G_{PE}, there is at least one such object name. Let o be any such name and x be the node in G_{FA} such that $o \in B_{\text{FA}}(x)$.

  - There is an edge from $\mathit{node}_{\text{FA}}(o_1)$ to node x in G_{FA}, annotated with $*$.

  - There is an edge from $\mathit{node}_{\text{FA}}(o_2)$ to node x in G_{FA}, annotated with $*$.

  Therefore, $\mathit{apply}(o_1, *\,) \in O_{\text{FA}}(x)$ and $\mathit{apply}(o_2, *\,) \in O_{\text{FA}}(x)$, that is, $o_1^{'} \in O_{\text{FA}}(x)$ and $o_2^{'} \in O_{\text{FA}}(x)$.

  Since $o \in B_{\text{PE}}(n_2)$ and $o \in B_{\text{FA}}(x)$, by Lemma A.2.2.1, $B_{\text{FA}}(x) \subseteq B_{\text{PE}}(n_2)$.

So we have proved that the lemma holds for paths of length 1.

Induction Hypothesis:

Assume the lemma is true for path length of $(j-1)$ in G_{PE}, where $j \geq 2$.

Induction Step:

Consider a path, $n_1 n_2 ... n_j n_{j+1}$, of length j in G_{PE}, annotated with a sequence of accessors A=$a_1 a_2 ... a_{j-1} a_j$, such that

- NumOfDerefs(A) ≥ 1, and
- $o_1^{'} = \mathit{apply}^*(o_1, A)$, where $o_1 \in B_{\text{PE}}(n_1)$, and
- $o_2^{'} = \mathit{apply}^*(o_2, A)$, where $o_2 \in B_{\text{PE}}(n_1)$.

The path from n_1 to n_j in G_{PE} is of length $(j-1)$ and is annotated with $a_1 a_2 ... a_{j-1}$. There is an edge in G_{PE} from n_j to n_{j+1}, annotated with accessor a_j.

Consider the number of pointer dereferences ($*$) in $a_1 a_2 ... a_{j-1}$:

- NumOfDerefs($a_1a_2...a_{j-1}$) = 0.

  In this case, a_1, a_2, ..., a_{j-1} are all member names. Since NumOfDerefs($a_1a_2...a_{j-1}a_j$) ≥ 1, a_j must be $*$.

  Because a_1 is a member name, both o_1 and o_2 are of structure type; therefore neither o_1 nor o_2 contain &. We have $o_1 \in B_1$ and $o_2 \in B_1$.

  Because of the path in G_{PE} from n_1 to n_j, by Lemma A.2.1.2, for o_1, there is an object name b_1, such that:

  - $b_1 \in (B_{\text{PE}}(n_j) \cap B_1)$ and
  - there is a path in G_{FA} from $node_{\text{FA}}(o_1)$ to $node_{\text{FA}}(b_1)$, annotated with the sequence of accessors $a_1a_2...a_{j-1}$.

  Similarly, for o_2, there is an object name b_2, such that:

  - $b_2 \in (B_{\text{PE}}(n_j) \cap B_1)$ and
  - there is a path in G_{FA} from $node_{\text{FA}}(o_2)$ to $node_{\text{FA}}(b_2)$, annotated with the sequence of accessors $a_1a_2...a_{j-1}$.

  Because the edge in G_{PE} from n_j to n_{j+1} is annotated with $*$ (i.e., a_j), by the algorithm that constructs G_{FA}, there is an object name $b \in B_{\text{PE}}(n_{j+1})$ such that there are two edges, one from $node_{\text{FA}}(b_1)$ to $node_{\text{FA}}(b)$ and another from $node_{\text{FA}}(b_1)$ to $node_{\text{FA}}(b)$, both of which are annotated with $*$. Let x be $node_{\text{FA}}(b)$, i.e., $b \in B_{\text{FA}}(x)$.

  So there are a path from $node_{\text{FA}}(o_1)$ through $node_{\text{FA}}(b_1)$ to x and a path from $node_{\text{FA}}(o_2)$ through $node_{\text{FA}}(b_2)$ to x in G_{FA}, both of which are annotated with $a_1a_2...a_{j-1}a_j$. By definition, we have

  $$apply^*(o_1,a_1a_2...a_j) \in O_{\text{FA}}(x) \text{ and } apply^*(o_2,a_1a_2...a_j) \in O_{\text{FA}}(x)$$

  Since $b \in B_{\text{PE}}(n_{j+1})$ and $b \in B_{\text{FA}}(x)$, by Lemma A.2.2.1, $B_{\text{FA}}(x) \subseteq B_{\text{PE}}(n_{j+1})$.

- NumOfDerefs($a_1a_2...a_{j-1}$) ≥ 1.

  By the induction hypothesis, there is a node $y \in G_{\text{FA}}$ such that

- $B_{\text{FA}}(y) \subseteq B_{\text{PE}}(n_j)$, and
- $apply^*(o_1,a_1a_2...a_{j-1}) \in O_{\text{FA}}(y)$, and
- $apply^*(o_2,a_1a_2...a_{j-1}) \in O_{\text{FA}}(y)$.

Let o be an object name such that $o \in B_{\text{FA}}(y)$.

Since $B_{\text{FA}}(y) \subseteq B_{\text{PE}}(n_j)$, $o \in B_{\text{PE}}(n_j)$.

Because of the edge in G_{PE} from n_j to n_{j+1} annotated with accessor a_j, by Lemma A.2.1.1, for o, there is an object name $o' \in (B_{\text{PE}}(n_{j+1}) \cap B_1)$, such that there is an edge in G_{FA} from $node_{\text{FA}}(o)$ (i.e., y) to $node_{\text{FA}}(o')$, that is annotated with a_j.

Let x be $node_{\text{FA}}(o')$. In other words, $o' \in B_{\text{FA}}(x)$.

Because we have

$$apply^*(o_1,a_1a_2...a_{j-1}) \in O_{\text{FA}}(y) \text{ and } apply^*(o_2,a_1a_2...a_{j-1}) \in O_{\text{FA}}(y)$$

and there is an edge from y to x annotated with a_j, we have

$$apply^*(o_1,a_1a_2...a_j) \in O_{\text{FA}}(x) \text{ and } apply^*(o_2,a_1a_2...a_j) \in O_{\text{FA}}(x)$$

Since $o' \in B_{\text{PE}}(n_{j+1})$ and $o' \in B_{\text{FA}}(x)$, by Lemma A.2.2.1, $B_{\text{FA}}(x) \subseteq B_{\text{PE}}(n_{j+1})$.

So we have proved that the lemma is true for paths of length j in G_{FA}.

□

Lemma A.3.0.2 *The $EQ_{G_{\text{FA}}}$ relation contains the FA relation.*

□

Proof:

By definition, FA is the smallest equivalence relation containing R_1, where R_1 is defined as follows:

$$\text{R}_1 = \left\{ (o_1',o_2') \;\middle|\; \begin{array}{l} \text{there is a path from a node } n \text{ to a node } m \text{ in } G_{\text{PE}} \text{ such that} \\ \text{the path is annotated with a sequence of accessors } a_1a_2...a_j, \\ \text{NumOfDerefs}(a_1a_2...a_j) \geq 1,\ o_1' = apply^*(o_1,a_1a_2...a_j) \text{ and} \\ o_2' = apply^*(o_2,a_1a_2...a_j), \text{ where } o_1 \in B_{\text{PE}}(n) \text{ and } o_2 \in B_{\text{PE}}(n) \end{array} \right\}$$

By Lemma A.3.0.1, the $\mathrm{EQ}_{G_{\mathrm{FA}}}$ relation contains R_1. Since $\mathrm{EQ}_{G_{\mathrm{FA}}}$ is an equivalence relation, the $\mathrm{EQ}_{G_{\mathrm{FA}}}$ relation contains the FA relation.

□

We show that the FA relation contains the $\mathrm{EQ}_{G_{\mathrm{FA}}}$ relation.

Lemma A.3.0.1 *Given G_{FA_i}, where $0 \leq i \leq k$, the following are true:*

(1) Let x be any node in G_{FA_i}. For any two object names in $B_{\mathrm{FA}_i}(x)$, o_1 and o_2, $(o_1,o_2) \in FA$.

(2) Let x and y be two nodes in G_{FA_i} such that there is an edge from x to y in the graph and the edge is annotated with an accessor a. There exist $o_1 \in B_{\mathrm{FA}_i}(x)$ and $o_2 \in B_{\mathrm{FA}_i}(y)$ such that $(apply(o_1,a),o_2) \in FA$.

□

Proof: By induction on the number of operations to construct G_{FA}.

Induction Basis:

Initially, G_{FA_0} has one node for each object name in B_1 and has no edges. So the lemma holds for G_{FA_0}.

Induction Hypothesis:

The lemma holds for G_{FA_i}, where $i \geq 0$.

Induction Step:

Consider a case analysis of the $(i+1)^{th}$ operation that is applied to G_{FA_i}:

- An edge annotated with a member name *member*, is added.

  Assume the edge is added from node x to node y in G_{FA_i}. The resulting graph is $G_{\mathrm{FA}_{i+1}}$.

  There is no change to object names associated with nodes in G_{FA_i}; so for any node x in $G_{\mathrm{FA}_{i+1}}$, $B_{\mathrm{FA}_{i+1}}(x) = B_{\mathrm{FA}_i}(x)$.

  By the induction hypothesis, (1) holds for any node in $G_{\mathrm{FA}_{i+1}}$ and (2) holds for all edges in $G_{\mathrm{FA}_{i+1}}$ except the one added. We just have to prove that (2) holds for the edge added.

By the algorithm that constructs G_{FA}, this edge is added because the following are true:

- There is an edge in G_{PE} from a node n to a node m, annotated with *member*.
- There is an object name $o_1 \in B_{\mathrm{PE}}(n)$ such that
  1. $o_1 \in B_{\mathrm{FA}_i}(x)$
  2. $o_2 = apply(o_1, member) \in B_{\mathrm{FA}_i}(y)$

Since the FA relation is reflexive, $(apply(o_1, member), o_2) \in \mathrm{FA}$.

So (2) holds for the edge added.

The lemma holds for $G_{\mathrm{FA}_{i+1}}$ in this case.

- An edge annotated with $*$, is added.

Assume the edge is added from node x to node y in G_{FA_i}. The resulting graph is $G_{\mathrm{FA}_{i+1}}$.

For any node x in $G_{\mathrm{FA}_{i+1}}$, $B_{\mathrm{FA}_{i+1}}(x) = B_{\mathrm{FA}_i}(x)$.

By the induction hypothesis, (1) holds for any node in $G_{\mathrm{FA}_{i+1}}$ and (2) holds for all edges in $G_{\mathrm{FA}_{i+1}}$ except the one added. We just have to prove that (2) holds for the edge added.

By the algorithm that constructs G_{FA}, this edge is added because the following are true:

- There is an edge in G_{PE} from a node n to a node m, annotated with $*$.
- There are two object names, $o_1 \in B_{\mathrm{PE}}(n)$ and $o_2 \in B_{\mathrm{PE}}(m)$, such that
  1. $o_1 \in B_{\mathrm{FA}_i}(x)$
  2. $o_2 \in B_{\mathrm{FA}_i}(y)$
  3. $o_2 = apply(o, *)$, where $o \in B_{\mathrm{PE}}(n)$

By definition of FA relation, $(apply(o_1, *), o_2) \in \mathrm{FA}$.

So (2) holds for the edge added.

The lemma holds for $G_{\mathrm{FA}_{i+1}}$ in this case.

- Two nodes are unioned.

  Assume two nodes in G_{FA_i}, x and y, are unioned. The resulting graph is $G_{\mathrm{FA}_{i+1}}$. The effect of this union is as follows.

  - A new node z in $G_{\mathrm{FA}_{i+1}}$ supersedes x and y.
  - $B_{\mathrm{FA}_{i+1}}(z) = B_{\mathrm{FA}_i}(x) \cup B_{\mathrm{FA}_i}(y)$
  - All incoming edges to x or y in G_{FA_i} will be to z in $G_{\mathrm{FA}_{i+1}}$.
  - All outgoing edges from x or y in G_{FA_i} be from z in $G_{\mathrm{FA}_{i+1}}$.

  By the induction hypothesis, (1) holds for any node in $G_{\mathrm{FA}_{i+1}}$ except z and (2) holds for any edge in $G_{\mathrm{FA}_{i+1}}$. By the induction hypothesis, (1) holds for node x and node y in G_{FA_i}. So if we can prove that there exist $o_1 \in B_{\mathrm{FA}_i}(x)$ and $o_2 \in B_{\mathrm{FA}_i}(y)$ such that $(o_1,o_2) \in \mathrm{FA}$, then (1) holds for node z. Below we show that is the case.

  Nodes are unioned in two cases, either in the call to MERGE() routine in calculate-FA-relation(), i.e., the top-level call, or in a recursive call to MERGE() routine. Consider the following case analysis:

  - x and y are unioned in the top-level call to MERGE().

    In this case, x and y are unioned because the following are true:

    * There is an edge in G_{PE} from a node n to a node m, annotated with $*$.
    * There are two object names in $B_{\mathrm{PE}}(n)$, b_1 and b_2, such that
      · $o_1 = \mathit{apply}(b_1,\mathrm{DerefOp}) \in B_{\mathrm{PE}}(m)$
      · $o_2 = \mathit{apply}(b_2, *) \in B_{\mathrm{PE}}(m)$
      · $o_1 \in B_{\mathrm{FA}_i}(x)$
      · $o_2 \in B_{\mathrm{FA}_i}(y)$

    By definition of FA relation, $(o_1,o_2) \in \mathrm{FA}$.

  - x and y are unioned in a recursive call to MERGE().

In this case, x and y are unioned because there is a node w in G_{FA_i}, such that there are two edges, one from w to x and another from w to y in G_{FA_i}, both of which are annotated with an accessor a.

Due to the edge from w to x in G_{FA_i}, by the induction hypothesis, there is $o \in B_{\mathrm{FA}_i}(w)$ and $o_1 \in B_{\mathrm{FA}_i}(x)$ such that $(apply(o,a,,)o_1) \in \mathrm{FA}$.

Similarly, due to the edge from w to y in G_{FA_i}, by the induction hypothesis, there is $o^{'} \in B_{\mathrm{FA}_i}(w)$ and $o_2 \in B_{\mathrm{FA}_i}(y)$ such that $(apply(o^{'},a),o_2) \in \mathrm{FA}$.

By the induction hypothesis, $(o,o^{'}) \in \mathrm{FA}$. Because FA is a weakly right-regular relation, $(apply(o,a),apply(o^{'},a)) \in \mathrm{FA}$. Because FA is an equivalence relation, $(o_1,o_2) \in \mathrm{FA}$.

So (1) holds for z.

The lemma holds for $G_{\mathrm{FA}_{i+1}}$ in this case.

- A redundant edge is deleted.

  Because the edge is redundant, both (1) and (2) hold for $G_{\mathrm{FA}_{i+1}}$.

So no matter what the $(i+1)^{th}$ operation is, the lemma holds for $G_{\mathrm{FA}_{i+1}}$.

□

Lemma A.3.0.2 *Given G_{FA_i}, where $0 \le i \le k$, let x be any node in the graph. For any $o_1 \in O_{\mathrm{FA}_i}(x)$, there is $o_2 \in B_{\mathrm{FA}_i}(x)$ such that $(o_1,o_2) \in FA$.*

□

Proof: By Lemma A.3.0.1 and induction on the length of the path in G_{FA_i} for o_1.

□

Lemma A.3.0.3 *Given G_{FA_i}, where $0 \le i \le k$, let x be any node in the graph.*

For any two object names in $O_{\mathrm{FA}_i}(x)$, o_1 and o_2, $(o_1,o_2) \in FA$.

□

Proof: By Lemma A.3.0.1 and Lemma A.3.0.2.

□

Since G_{FA} is same as G_{FA_k}, by this lemma, the FA relation contains the $\mathrm{EQ}_{G_{\mathrm{FA}}}$ relation.

Appendix B

The FA Relation as Compile-time Alias Information

In this appendix, we proof that the FA relation defined in Section 5.4.1 is safe as compile-time aliasing information.

We first present an operational semantics for programs in our intermediate representation given in Figure 5.1. By using the semantics, we can describe run-time alias relations at program points. We then prove that the FA relation for a program safely estimates the run-time aliases at any program point along any execution path of the program.

This appendix has four sections. In Section B.1, we present the syntax of our intermediate language. In Section B.2, we give the operational semantics. In Section B.3, we define run-time aliases. Finally, in Section B.4, we show that the FA relation is safe to be used as compile-time aliasing relation.

B.1 Syntax

The syntax of our intermediate representation is given in Figure 5.1 (Chapter 5). Each procedure has a unique entry statement and a unique exit statement; each procedure call in the source has a call statement and a corresponding return statement in our representation. Any *return*(*exp*) in a C procedure is transformed into an assignment for a special variable representing the returned value of the procedure and a goto to the exit statement for the procedure. An important feature of the representation is that each intermediate value in arithmetic or relational expressions is explicitly named in the intermediate language.

B.2 Semantics

B.2.1 Tagged Location Names

CallString ::=	(Call | Return)*	
HeapCounter ::=	Integer	
Tag ::=	CallString	(for user names)
	| HeapCount	(for heap names)
Loc ::=	VarName	(variable names)
	HeapName	(heap names)
	Loc.member	(structure members)
TaggedLoc ::=	<Tag , Loc>	

Figure B.1: Tagged Location Names

We will define an operational semantics that explicitly represents pointer values by location names. Because of recursion and dynamic allocation, simple location names do not uniquely identify locations. We will use tagged location names to distinguish locations allocated in different invocations of the same procedure or locations dynamically allocated at the same statement. There are basically two kinds of tags. One is a call string consisting of a sequence of calls and returns, which uniquely identifies formal parameters and local variables allocated for an invocation of a procedure. Another is simply an integer value, which is used to distinguish locations dynamically allocated at the same statements. The syntax for tags and tagged location names is given in Figure B.1.

We assume global variables are considered locals of the special procedure *_init_*(); thus their tag is the empty call string ϵ. Given a call string and a variable name visible in the procedure current being called, the function $Tag()$ returns the tag for the variable, which is either the call string (if the variable is a formal parameter or a local variable) or the empty call string ϵ (if the variable is a global).

B.2.2 A Transition System Semantics

LValues ::=	TaggedLoc	(locations of primitive types)
Values ::=	integer	
	| float	
	| *nil*	(NULL pointer value)
	| TaggedLoc	(pointer values)
Environment:	TaggedLoc $\longrightarrow$ $\text{TaggedLoc}_\perp$	
Store:	Lvalues $\longrightarrow$ $\text{Values}_\perp$	
State:	CallString $\times$ HeapCount $\times$ Environment $\times$ Store	

Figure B.2: Abstract Machine

We define the semantics of programs via an abstract machine that explicitly represents pointer values by tagged location names. The state of the machine consists of a call string, a heap counter, an environment and a store. The call string and the heap counter are mainly used for tagging object names. The environment maps a tagged location name to itself if the location has been allocated and to $\perp$ if the location is out of scope or freed. The store maps a tagged location of primitive type to its value, which can be either a constant (i.e., a numeric value) or a tagged location name (i.e., a pointer value). The definition of the machine state is given in Figure B.2.

Before we present the transition rules, we give a few auxiliary functions in Figure B.3, which will be used in the transition rules. Given a variable name or a heap name, *locations*() returns a set of locations that can be derived from the name and *members*() returns a set of member sequences that can be applied to the name to derive locations of primitive type. Let v be either a variable name or a heap name. Intuitively, $locations(v)$ is the set of locations that can be referenced through v after v is allocated and $members(v)$ is the set of sequences of member accessors, which can be applied to v to obtain locations of primitive type, which are locations that can be assigned values. By definition of B_1, $locations(v) \subseteq B_1$.

In Figure B.4, we show two functions for evaluating object names and expressions.

$$locations(o) = \begin{cases} \{ \ o \ \} & o \text{ is of a primitive type} \\ \{ \ o \ \} \ \cup \displaystyle\bigcup_{i=1}^{k} locations(o.f_i) & o \text{ is of a structure type} \\ & \text{with members } f_1, ..., f_k \end{cases}$$

$$members(o) = \begin{cases} \{ \ \epsilon \ \} & \begin{array}{l} o \text{ is either a variable or a heap name and} \\ \text{is of a primitive type} \end{array} \\ \{ \ F \ \} & \begin{array}{l} o \text{ is of a primitive type and } o = apply^*(loc,F), \\ \text{where } loc \text{ is either a variable or heap name} \\ \text{and } F \text{ is a sequence of members} \end{array} \\ \displaystyle\bigcup_{i=1}^{k} members(o.f_i) & o \text{ is of a structure type with members } f_1, ..., f_k \end{cases}$$

Examples:

$struct \ \{ \ int \ a; \ struct \ \{ \ int \ b; \ float \ c; \ \} \ s; \ float \ d; \ \} \ v;$
$locations(v) = \{ \ v, \ v.a, \ v.s, \ v.s.b, \ v.s.c, \ v.d \ \}$
$members(v) = \{ \ a, \ s.b, \ s.c, \ d \ \}$

Figure B.3: Auxiliary Functions

Function $\mathcal{LE}$ evaluates an object name to its l-value and function $\mathcal{E}$ evaluates an expression to its r-value.

For an object name o, VAR(o) is the variable or heap name from which o is derived. An object name o can get its tag from VAR(o) and the current call string c by using the function $Tag(c,\text{VAR}(o))$. Function $\mathcal{LE}_{\prime}$ evaluates a tagged object name to its l-value.

Function $next(l)$ returns the label of the statement immediate following the statement labeled with l.

The semantics of statements are defined through transition rules, where a configuration

$$l\text{: } (c,h,e,s)$$

consists of a label and a state, where the label represents a program point. We will use the notation $e[<tag \ , \ loc> \ \rightarrow \ <tag \ , \ loc>]$ to denote an environment that is same as e except that it maps $<tag \ , \ loc>$ to $<tag \ , \ loc>$, and use the notation $s[<tag \ , \ loc> \ \rightarrow \ val]$ to denote a store that is same as s except that it maps $<tag \ , \ loc>$ to

$$\mathcal{LE}[\![\ o\ ]\!]\ (c,h,e,s) \quad = \quad \mathcal{LE}_1[\![\ <Tag(c,\mathrm{VAR}(o))\ ,\ o>\ ]\!]\ (e,s)$$

$$\begin{aligned}
\mathcal{LE}_1[\![\ <tag\ ,\ var>\ ]\!]\ (e,s) &= e(<tag\ ,\ var>) \\
\mathcal{LE}_1[\![\ <tag\ ,\ heap_{id}>\ ]\!]\ (e,s) &= e(<tag\ ,\ heap_{id}>) \\
\mathcal{LE}_1[\![\ <tag\ ,\ *o>\ ]\!]\ (e,s) &= e(s(\ \mathcal{LE}_1[\![\ <tag\ ,\ o>\ ]\!]\ (e,s))) \\
\mathcal{LE}_1[\![\ <tag\ ,\ o.f>\ ]\!]\ (e,s) &= e(<tag_1\ ,\ loc_1.f>), \\
&\quad \text{where } <tag_1\ ,\ loc_1> = \mathcal{LE}_1[\![\ <tag\ ,\ o>\ ]\!]\ (e,s)
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}[\![\ k\ ]\!]\ (c,h,e,s) &= k \\
\mathcal{E}[\![\ o\ ]\!]\ (c,h,e,s) &= s(\mathcal{LE}[\![\ o\ ]\!]\ (c,h,e,s)) \\
\mathcal{E}[\![\ e_1 \text{ op } e_2\ ]\!]\ (c,h,e,s) &= \mathcal{E}[\![\ e_1\ ]\!]\ (c,h,e,s) \text{ op } \mathcal{E}[\![\ e_2\ ]\!]\ (c,h,e,s) \\
\mathcal{E}[\![\ \&o\ ]\!]\ (c,h,e,s) &= \mathcal{LE}[\![\ o\ ]\!]\ (c,h,e,s)
\end{aligned}$$

Figure B.4: Evaluation Functions

val. The transition rules for statements are given in Figure B.5 and B.6. The initial configuration is l_0:(ϵ,0,e_0,s_0), where l_0 is the label for entry of the procedure *_init_*(), e_0 maps any tagged location name to $\perp$ and s_0 maps any tagged location name to $\perp$. An execution path can be described by a sequence of configurations:

$$l_0{:}t_0 \ \longmapsto\ l_1{:}t_1 \ \longmapsto\ \ldots\ \longmapsto\ l_n{:}t_n$$

where $t_0 = (\epsilon,0,e_0,s_0)$. The intuition of the transition rules for various kinds of statements is given below:

- l: entry of P

  At entry of a procedure, a number of locations are allocated for local variables of the procedure.

- l: exit of P

  At exit of a procedure, a number of locations are deallocated.

- l: call P(a_1,...,a_m)

  At a call statement, a number of locations are allocated for formal parameters of the procedure being called and the call string is adjusted with the label of this call statement. Formal-actual parameter bindings are handled as assignments.

Because it is possible that formal and actual parameters are of structure types, values of structure members may be evaluated and assigned.

- l: return from P

  At a return statement, a number of locations are deallocated and the call string is adjusted with the label of this return statement.

- l: $o = exp$

  At a non-pointer assignment statement, the object name on the left hand side is evaluated to a location and the expression on the right hand side is evaluated to a value. The value at the location is updated.

- l: $o =$ NULL

  At a pointer assignment with NULL as right hand side, the object name on the left hand side is evaluated to a location and the value of the location is updated to be *nil*.

- l: $o = \&o_1$

  At a pointer assignment with & in its right hand side, the object name on the left hand side is evaluated to a location and the right hand side is evaluated to the location name whose address is being taken. The value of the former is updated to be the latter, i.e., the latter is pointed to by the former.

- l: $o = o_1$

  At a pointer assignment with object names as both sides, the object name on the left hand side is evaluated to a location and the right hand side is evaluated to a pointer value, which is a location name. The value of the former is updated to be the latter.

- l: $s = s_1$

  At a structure assignment, the value of each member of primitive type for the right hand side is assigned to the corresponding member for the left hand side.

- l: HeapAlloc(o)

  At a dynamic allocation statement, a number of locations are allocated. The object name is evaluated to a location and the heap location name is stored at the location.

- l: HeapDealloc(o)

  At a dynamic deallocation statement, a number of locations are deallocated.

- l: if (o) (goto l_{true}) (goto l_{false})

  At a conditional statement, the object name is evaluated to its value. Depending on the value, one of the two branches is taken.

- l: goto l'

  At a goto statement, the target branch is taken.

B.3 Run-time Aliases and Store Structures

For a configuration on an execution path, an alias relation can be defined.

Definition B.3.1 *Given a configuration l:(c,h,e,s), two tagged object names,*

$$<tag_1\ ,\ obj_1>\ \textit{and}\ <tag_2\ ,\ obj_2>$$

are aliased with respect to the configuration if and only if the following are true:

- $\mathcal{LE}_1[\![\ <tag_1\ ,\ obj_1>\ ]\!]\ (e,s) \neq \perp$
- $\mathcal{LE}_1[\![\ <tag_2\ ,\ obj_2>\ ]\!]\ (e,s) \neq \perp$
- $\mathcal{LE}_1[\![\ <tag_1\ ,\ obj_1>\ ]\!]\ (e,s) = \mathcal{LE}_1[\![\ <tag_2\ ,\ obj_2>\ ]\!]\ (e,s)$

□

The alias relation with respect to a configuration is an equivalence relation on a set of tagged object names. Therefore it implies a set of equivalence classes, each of which contains exactly one tagged location name. The alias relation can also be represented as a directed multi-graph.

$$\dfrac{\begin{array}{c} l\text{: entry of P} \qquad l^{'} = next(l) \\ \{\ v_1, ..., v_{k_P}\ \} = \text{localVariables(P)} \qquad \{\ loc_1, ..., loc_{n_l}\ \} = \cup_{i=1}^{k_P}\ locations(v_i) \end{array}}{l\text{: } (c,h,e,s) \longmapsto l^{'}\text{: } (c,\ h,\ e[<c\ ,\ loc_1> \rightarrow <c\ ,\ loc_1>]\ ...\ [<c\ ,\ loc_{n_l}> \rightarrow <c\ ,\ loc_{n_l}>],\ s)}$$

$$\dfrac{\begin{array}{c} l\text{: exit of P} \qquad c^{'}.l_{call} = c \qquad l^{'} = next(l_{call}) \\ \{\ v_1, ..., v_{k_P}\ \} = \text{localVariables(P)} \qquad \{\ loc_1, ..., loc_{n_l}\ \} = \cup_{i=1}^{k_P}\ locations(v_i) \end{array}}{l\text{: } (c,h,e,s) \longmapsto l^{'}\text{: } (c,\ h,\ e[<c\ ,\ loc_1> \rightarrow \perp]\ ...\ [<c\ ,\ loc_{n_l}> \rightarrow \perp],\ s)}$$

$$\dfrac{\begin{array}{c} l\text{: call } \mathrm{P}(a_1, ..., a_m) \qquad l^{'} = entry(\mathrm{P}) \\ \{\ f_1, ..., f_m\ \} = \text{formalParameters(P)} \qquad \{\ loc_1, ..., loc_{n_l}\ \} = \cup_{i=1}^{m}\ locations(f_i) \\ \{\ F_{i,1}, ..., F_{i,k_i}\ \} = members(f_i) \qquad val_{i,j} = \mathcal{E}[\![\ apply^*(a_i, F_{i,j})\ ]\!]\ (c,h,e,s) \qquad 1 \leq i \leq m \qquad 1 \leq j \leq k_i \\ c^{'} = c.l \qquad s^{'} = s[<c^{'}\ ,\ apply^*(f_1, F_{1,1})> \rightarrow val_{1,1}]\ ...\ [<c^{'}\ ,\ apply^*(f_m, F_{m,k_m})> \rightarrow val_{m,k_m}] \end{array}}{l\text{: } (c,h,e,s) \longmapsto l^{'}\text{: } (c^{'},\ h,\ e[<c^{'}\ ,\ loc_1> \rightarrow <c^{'}\ ,\ loc_1>]\ ...\ [<c^{'}\ ,\ loc_{n_l}> \rightarrow <c^{'}\ ,\ loc_{n_l}>],\ s^{'})}$$

$$\dfrac{\begin{array}{c} l\text{: return from P} \qquad l^{'} = next(l) \qquad c^{'} = c.l \\ \{\ f_1, ..., f_m\ \} = \text{formalParameters(P)} \qquad \{\ loc_1, ..., loc_{n_l}\ \} = \cup_{i=1}^{m}\ locations(f_i) \end{array}}{l\text{: } (c,h,e,s) \longmapsto l^{'}\text{: } (c^{'},\ h,\ e[<c\ ,\ loc_1> \rightarrow \perp]\ ...\ [<c\ ,\ loc_{n_l}> \rightarrow \perp],\ s)}$$

$$\dfrac{l\text{: } o = exp \qquad l^{'} = next(l) \qquad <tag\ ,\ loc> = \mathcal{LE}[\![\ o\ ]\!]\ (c,h,e,s) \qquad val = \mathcal{E}[\![\ exp\ ]\!]\ (c,h,e,s)}{l\text{: } (c,h,e,s) \longmapsto l^{'}\text{: } (c,\ h,\ e,\ s[<tag\ ,\ loc> \rightarrow val])}$$

$$\dfrac{l\text{: } o = \text{NULL} \qquad l^{'} = next(l) \qquad <tag\ ,\ loc> = \mathcal{LE}[\![\ \mathrm{o}\ ]\!]\ (c,h,e,s)}{l\text{: } (c,h,e,s) \longmapsto l^{'}\text{: } (c,\ h,\ e,\ s[<tag\ ,\ loc> \rightarrow nil])}$$

Figure B.5: Transition Rules (Part 1)

$$\frac{l\text{: } o = \&o_1 \qquad l' = next(l) \qquad <tag\ ,\ loc> = \mathcal{LE}[\![\ o\ ]\!]\ (c,h,e,s) \qquad <tag_1\ ,\ loc_1> = \mathcal{E}[\![\ \&o_1\ ]\!]\ (c,h,e,s)}{l\text{: } (c,h,e,s) \longmapsto l'\text{: } (c,\ h,\ e,\ s[<tag\ ,\ loc> \rightarrow <tag_1\ ,\ loc_1>])}$$

$$\frac{l\text{: } o = o_1 \qquad l' = next(l) \qquad <tag\ ,\ loc> = \mathcal{LE}[\![\ o\ ]\!]\ (c,h,e,s) \qquad <tag_1\ ,\ loc_1> = \mathcal{E}[\![\ o_1\ ]\!]\ (c,h,e,s)}{l\text{: } (c,h,e,s) \longmapsto l'\text{: } (c,\ h,\ e,\ s[<tag\ ,\ loc> \rightarrow <tag_1\ ,\ loc_1>])}$$

$$\frac{\begin{array}{c} l\text{: } s = s_1 \qquad l' = next(l) \qquad <tag\ ,\ loc> = \mathcal{LE}[\![\ s\ ]\!]\ (c,h,e,s) \\ \{\ F_1, ..., F_{n_l}\ \} = members(loc) \qquad val_i = \mathcal{E}[\![\ apply^*(s_1,F_i)\ ]\!]\ (c,h,e,s) \qquad 1 \leq i \leq n_l \\ s' = s[<tag\ ,\ apply^*(loc,F_1)> \rightarrow val_1]\ ...\ [<tag\ ,\ apply^*(loc,F_{n_l})> \rightarrow val_{n_l}] \end{array}}{l\text{: } (c,h,e,s) \longmapsto l'\text{: } (c,\ h,\ e,\ s')}$$

$$\frac{\begin{array}{c} l\text{: HeapAlloc(o)} \qquad l' = next(l) \\ <tag\ ,\ loc> = \mathcal{LE}[\![\ o\ ]\!]\ (c,h,e,s) \qquad \{\ loc_1, ..., loc_{n_l}\ \} = locations(heap_l) \\ e' = e[<h\ ,\ loc_1> \rightarrow <h\ ,\ loc_1>]\ ...\ [<h\ ,\ loc_{n_l}> \rightarrow <h\ ,\ loc_{n_l}>] \end{array}}{l\text{: } (c,h,e,s) \longmapsto l'\text{: } (c,\ h{+}1,\ e',\ s[<tag\ ,\ loc> \rightarrow <h\ ,\ heap_l>])}$$

$$\frac{\begin{array}{c} l\text{: HeapDealloc(o)} \qquad l' = next(l) \\ <tag\ ,\ heap_{l''}> = \mathcal{E}[\![\ o\ ]\!]\ (c,h,e,s) \qquad \{\ loc_1, ..., loc_{n_l}\ \} = locations(heap_{l''}) \end{array}}{l\text{: } (c,h,e,s) \longmapsto l'\text{: } (c,\ h,\ e[<tag\ ,\ loc_1> \rightarrow \bot]\ ...\ [<tag\ ,\ loc_{n_l}> \rightarrow \bot],\ s)}$$

$$\frac{l\text{: if } (o) \text{ (goto } l_{true}\text{) (goto } l_{false}\text{)} \qquad val = \mathcal{E}[\![\ o\ ]\!]\ (c,h,e,s) \qquad val \neq 0}{l\text{: } (c,h,e,s) \longmapsto l_{true}\text{: } (c,h,e,s)}$$

$$\frac{l\text{: if } (o) \text{ (goto } l_{true}\text{) (goto } l_{false}\text{)} \qquad val = \mathcal{E}[\![\ o\ ]\!]\ (c,h,e,s) \qquad val == 0}{l\text{: } (c,h,e,s) \longmapsto l_{false}\text{: } (c,h,e,s)}$$

$$\frac{l\text{: goto } l'}{l\text{: } (c,h,e,s) \longmapsto l'\text{: } (c,h,e,s)}$$

Figure B.6: Transition Rules (Part 2)

Definition B.3.2 *Given a configuration l:(c,h,e,s), a directed graph can be constructed as follows:*

- *There is a node in the graph for each $<tag\ ,\ loc>$ such that $e(<tag\ ,\ loc>) \neq \perp$.*
- *There is an edge in the graph annotated with $*$ from the node for $<tag\ ,\ loc>$ to the node for $<tag_1\ ,\ loc_1>$ if $s(<tag\ ,\ loc>) = <tag_1\ ,\ loc_1>$.*
- *There is an edge in the graph annotated with member from the node for $<tag\ ,\ loc>$ to the node for $<tag\ ,\ loc.member>$.*

We call this graph the store structure graph.

□

Let VAR(o), where $o \in B_0$, be the variable name or heap name from which the object name o is derived. Note that this name is always a location name. The next two lemmas shows that the store structure graph corresponding to a configuration captures the alias relation defined for the configuration.

Lemma B.3.1 *Let l:(c,h,e,s) be a configuration and SG be the corresponding store structure graph. $\mathcal{LE}_1[\![\ <tag\ ,\ obj>\ ]\!]\ (e,s) = <tag_1\ ,\ loc_1>$ if and only if there is a path in SG from the node for $<tag\ ,\ VAR(obj)>$ to the node for $<tag_1\ ,\ loc_1>$ such that the path is annotated with a sequence of accessors A and $obj = apply^*(VAR(obj),A)$.*

□

This lemma can be proved by induction on the path length in SG and induction on the number of accessors applied to derive obj.

Let l:(c,h,e,s) be a configuration and SG be the corresponding store structure graph. By Lemma B.3.1, if $\mathcal{LE}_1[\![\ <tag\ ,\ obj>\ ]\!]\ (e,s) = <tag_1\ ,\ loc_1>$, there is a path in SG for $<tag\ ,\ obj>$ from the node for $<tag\ ,\ \mathrm{VAR}(obj)>$ to the node for $<tag_1\ ,\ loc_1>$.

Lemma B.3.2 *Let l:(c,h,e,s) be a configuration and SG be the corresponding store structure graph. Two tagged object names,*

$$<tag_1\ ,\ obj_1>\ \textit{and}\ <tag_2\ ,\ obj_2>$$

are aliased with respect to the configuration if and only if the path in SG *for* $<tag_1\ , obj_1>$ *and the path in* SG *for* $<tag_2\ , obj_2>$ *end up at the same node.*

□

This lemma is a direct result of Lemma B.3.1.

Given an execution path $l_0{:}t_0 \longmapsto l_1{:}t_1 \longmapsto \ldots \longmapsto l_n{:}t_n$, we can derive a sequence of store structure graphs, each of which corresponds to one configuration. Each transition step performs a sequence of operations on store structure graphs. The following are the basic operations on a store structure graph:

- add a new node and if necessary, add edges annotated with member names to or from the node

  This operation corresponds to a change to the environment such as

  $$e[<tag\ , loc> \rightarrow <tag\ , loc>]$$

  in the transition rules.

  The effect of this operation is as follows:

  – A new node added. Suppose the node is for the tagged location name $<tag\ , loc>$.

  – If $loc = apply(loc_1, member)$, where *member* is an member accessor, and there is a node for $<tag\ , loc_1>$, an edge annotated with *member* is added from the node for $<tag\ , loc_1>$ to the new node.

  – If there is a node for $<tag\ , apply(loc, member)>$ in the graph, where *member* is a member accessor, an edge annotated with *member* is added from the new node to that node.

- delete a node and delete all edges to or from that node

  This operation corresponds to a change to the environment such as

  $$e[<tag\ , loc> \rightarrow \perp]$$

in the transition rules.

- add an edge annotated with $*$

  This operation corresponds to a change to the store such as

  $$s[<tag\ ,\ loc> \to <tag_1\ ,\ loc_1>]$$

  in the transition rules. The effect of this operation on the store structure graph is that an edge annotated with $*$ is added (e.g., from the node for $<tag\ ,\ loc>$ to the node for $<tag_1\ ,\ loc_1>$).

- remove an edge annotated with $*$

  This operation corresponds to a change to the store such as

  $$s[<tag\ ,\ loc> \to \perp]$$

  in the transition rules. The effect of this operation on the store structure graph is that an edge (e.g. from the node for $<tag\ ,\ loc>$) annotated with $*$ is deleted.

B.4 The FA Relation is Safe as Alias Information

Here are some of the notations used in this section:

- VAR(o), where $o \in B_0$, is the variable name or heap name from which the object name o is derived. VAR(o) is always a location name.

- $node_{\mathrm{PE}}(o)$, where $o \in B_0$, is the node n in G_{PE} such that $o \in B_{\mathrm{PE}}(n)$. That is, n represents the equivalence class of the PE relation containing o.

- $node_{\mathrm{FA}}(o)$, where $o \in B_1$, is the node x in G_{FA} such that $o \in B_{\mathrm{FA}}(x)$.

- $node_{SG}(<tag\ ,\ loc>)$, where SG is a store structure graph, is the node in SG for $<tag\ ,\ loc>$.

Definition B.4.0.1 *Let $G_1 = (N_1,E_1)$ and $G_2 = (N_2,E_2)$ be two directed graphs, in which edges are annotated with $*$ or member names. We say G_1 is embedded in G_2*

under a mapping $\mathcal{M}$: $N_1 \rightarrow N_2$ if for any edge $e_1 \in E_1$: $n \longrightarrow m$, there is an edge e_2 $\in E_2$: $\mathcal{M}(n) \longrightarrow \mathcal{M}(m)$, which has the same edge annotation as e_1.

We will use $G_1 \sqsubseteq_{\mathcal{M}} G_2$ to mean that G_1 is embedded in G_2 under mapping $\mathcal{M}$.

□

The mapping $\mathcal{M}$ in the definition can be either one-to-one or many-to-one.

For instance, given G_{FA} and G_{PE} for a program, we have $G_{\text{FA}} \sqsubseteq_{\mathcal{M}_0} G_{\text{PE}}$, where

$$\mathcal{M}_0(node_{\text{FA}}(o)) = node_{\text{PE}}(o), \text{ for any object name } o \in B_1.$$

This can be easily verified by use of Lemma A.2.2.1.

In the following lemma, we show that given a program, at any program point on an execution path of the program, the store structure graph SG is embedded in the graph G_{FA} under a mapping. Since G_{FA} represents the FA relation (Appendix A) and SG captures the alias relation at the program point (Lemma B.3.2), we effectively prove that the FA relation of the program is a safe estimate of run-time aliases.

Lemma B.4.0.1 *Given a program, let G_{FA} represent the FA relation for the program. Given an execution path of the program, $l_0{:}t_0 \longmapsto l_1{:}t_1 \longmapsto \ldots \longmapsto l_n{:}t_n$, let SG_i, $0 \le i \le n$, be the store structure graph corresponding to the configuration $l_i{:}t_i$.*
For any $0 \le i \le n$, $SG_i \sqsubseteq_{\mathcal{M}_1} G_{\text{FA}}$, where $\mathcal{M}_1(node_{SG_i}(<tag\ ,\ loc>)) = node_{\text{FA}}(loc)$.

□

Proof:

We will prove by induction on i that $SG_i \sqsubseteq_{\mathcal{M}_1} G_{\text{FA}}$.

Induction Basis: $i = 0$

The initial store structure graph does not have any node. The lemma holds.

Induction Hypothesis:

$SG_i \sqsubseteq_{\mathcal{M}_1} G_{\text{FA}}$, where $i \ge 0$.

Induction Step:

Let SG_i be the store structure graph corresponding to the configuration $l_i{:}t_i$. Let the state t_i be (c,h,e,s). We will prove that $SG_{(i+1)} \sqsubseteq_{\mathcal{M}_1} G_{\text{FA}}$.

Because of the induction hypothesis, we have the following two results, which will be useful in the induction proof. We will refer them as (1) and (2).

(1) If $o \in B_1$ and $<tag\ ,\ loc> = \mathcal{LE}[\![\ o\ ]\!]\ (c,h,e,s)$, then there is a node x in G_{FA} such that both loc and o are in $B_{\text{FA}}(x)$.

Proof: Since we have

$$<tag\ ,\ loc> = \mathcal{LE}[\![\ o\ ]\!]\ (c,h,e,s)\ (\text{i.e., } \mathcal{LE}_1[\![\ <Tag(c,\text{VAR}(o))\ ,\ o>\ ]\!]\ (e,s))$$

by Lemma B.3.1, there is a path from $node_{SG_i}(<Tag(c,\text{VAR}(o))\ ,\ \text{VAR}(o)>)$ to $node_{SG_i}(<tag\ ,\ loc>)$ in SG_i such that the path is annotated with a sequence of accessors A and $o = apply^*(\text{VAR}(o),A)$. By the induction hypothesis that $SG_i \sqsubseteq_{\mathcal{M}_1} G_{\text{FA}}$, there is a path in G_{FA} from $node_{\text{FA}}(\text{VAR}(o))$ to $node_{\text{FA}}(loc)$ and the path is annotated with A. Because of this path and the fact that $o \in B_1$, by Lemma A.2.3.3, $o \in node_{\text{FA}}(loc)$. Let x be $node_{\text{FA}}(loc)$; both loc and o are in $B_{\text{FA}}(x)$.

(2) If $o_1 \in B_1$ and $<tag_1\ ,\ loc_1> = \mathcal{E}[\![\ o_1\ ]\!]\ (c,h,e,s)$, then there are two nodes, y and z, in G_{FA} such that $loc_1 \in B_{\text{FA}}(y)$, $o_1 \in B_{\text{FA}}(z)$ and there is an edge from z to y annotated with $*$.

Proof: By definition, $\mathcal{E}[\![\ o_1\ ]\!]\ (c,h,e,s) = s(\mathcal{LE}[\![\ o_1\ ]\!]\ (c,h,e,s))$.

Let $<tag\ ,\ loc> = \mathcal{LE}[\![\ o_1\ ]\!]\ (c,h,e,s)$. By (1), there is a node z in G_{FA} such that both loc and o_1 are in $B_{\text{FA}}(x)$. Let y be the node in G_{FA} such that $loc_1 \in B_{\text{FA}}(y)$.

Since $<tag_1\ ,\ loc_1> = s(<tag\ ,\ loc>)$, by definition of store structure graphs, there is an edge in SG_i from $node_{SG_i}(<tag\ ,\ loc>)$ to $node_{SG_i}(<tag_1\ ,\ loc_1>)$ and the edge is annotated with $*$. By the induction hypothesis that $SG_i \sqsubseteq_{\mathcal{M}_1} G_{\text{FA}}$, there is an edge in G_{FA} from $node_{\text{FA}}(loc)$ (i.e., node z) to $node_{\text{FA}}(loc_1)$ (i.e., node y), which is annotated with $*$.

We now show that $SG_{(i+1)} \sqsubseteq_{\mathcal{M}_1} G_{\text{FA}}$ by a case analysis of the statement labeled by l_i:

- l_i: entry of P

  The effect of this statement on SG_i is that a number of nodes are added and edges annotated with member names between the new nodes may be added.

  Each node added corresponds to a tagged location name $<c\ ,\ loc_j>$, where $loc_j \in locations(v)$ and v is a local variable of P. Since $locations(v) \subseteq B_1$, there is a node x in G_{FA} such that $loc_j \in B_{\text{FA}}(x)$.

  An edge annotated with a member name *member* may be added from the node for $<tag\ ,\ loc>$ to the node for $<tag\ ,\ loc.member>$ if the two nodes are just added.

  Since both *loc* and *loc.member* are in B_1, there are nodes, n and m, in G_{PE} such that $loc \in B_{\text{PE}}(n)$ and $loc.member \in B_{\text{PE}}(m)$. By the construction of G_{PE}, there is an edge from n to m annotated with *member*. Because of this edge, by the construction of G_{FA}, there is an edge from $node_{\text{FA}}(loc)$ to $node_{\text{FA}}(loc.member)$ and the edge is annotated with *member*.

  So $SG_{(i+1)} \sqsubseteq_{\mathcal{M}_1} G_{\text{FA}}$.

- l_i: exit of P

  The effect of this statement on SG_i is that a number of nodes are deleted.

  Trivially, $SG_{(i+1)} \sqsubseteq_{\mathcal{M}_1} G_{\text{FA}}$.

- l_i: call P(a_1,...,a_m)

  The effect of this statement on SG_i is as follows:

  - A number of nodes are added.
  - Edges annotated with member names between the new nodes may be added.
  - Edges annotated with $*$ from the new nodes to nodes in SG_i may be added.

  Each node added corresponds to a tagged location name $<c.l_i\ ,\ loc_j>$, where $loc_j \in locations(f)$ and f is a formal parameter of P. Since $locations(f) \subseteq B_1$, it is easy to prove that there is a corresponding node in G_{FA} for each new node and there is a corresponding edge in G_{FA} for each new edge annotated with a member name between new nodes. A similar proof has been given for l_i: entry of P.

Each new edge annotated with $*$ is due to a formal-actual parameter binding. Let f be the formal parameter and a be its corresponding actual parameter. The formal-actual binding has the same effect as an assignment: $f = a$. There are three cases.

- $a = \&o_1$.

  Let $<tag_1 , loc_1> = \mathcal{LE}[\![\ o_1 \ ]\!] \ (c,h,e,s)$. The edge is added from the node for $<c.l_i , f>$ to the node for $<tag_1 , loc_1>$. The argument that there is an edge annotated with $*$ from $node_{\text{FA}}(f)$ to $node_{\text{FA}}(loc_1)$ in G_{FA} is similar to the one for a pointer assignment with & on right hand side (l_i: $o = \&o_1$).

- a does not contain & and is of pointer type.

  Let $<tag_1 , loc_1> = \mathcal{E}[\![\ o_1 \ ]\!] \ (c,h,e,s)$. The edge is added from the node for $<c.l_i , f>$ to the node for $<tag_1 , loc_1>$. The argument that there is an edge annotated with $*$ from $node_{\text{FA}}(f)$ to $node_{\text{FA}}(loc_1)$ in G_{FA} is similar to the one for a pointer assignment with object names on both sides (see the argument for l_i: $o = o_1$).

- a does not contain & and is of a structure type.

  Let $<tag_1 , loc_1> = \mathcal{E}[\![\ apply^*(a,F) \ ]\!] \ (c,h,e,s)$, where F is a sequence of member accessor applicable to a and f. The edge is added from the node for $<c.l_i , apply^*(f,F)>$ to the node for $<tag_1 , loc_1>$. The argument that there is an edge annotated with $*$ from $node_{\text{FA}}(apply^*(f,F))$ to $node_{\text{FA}}(loc_1)$ in G_{FA} is similar to the one for a structure assignment (see the argument for l_i: $s = s_1$).

So $SG_{(i+1)} \sqsubseteq_{\mathcal{M}_1} G_{\text{FA}}$.

- l_i: return from P

  The effect of this statement on SG_i is that a number of nodes are deleted.

  Trivially $SG_{(i+1)} \sqsubseteq_{\mathcal{M}_1} G_{\text{FA}}$.

- l_i: $o = e$

  This statement has no effect on SG_i; that is, $SG_{(i+1)}$ is same as SG_i.

  Trivially $SG_{(i+1)} \sqsubseteq_{\mathcal{M}_1} G_{\text{FA}}$.

- l_i: $o =$ NULL

  The effect of this statement on SG_i is that an edge annotated with $*$ may be deleted.

  Trivially $SG_{(i+1)} \sqsubseteq_{\mathcal{M}_1} G_{\text{FA}}$.

- l_i: $o = \&o_1$

  Let $<tag\ ,\ loc> = \mathcal{LE}[\![\ o\ ]\!]\ (c,h,e,s)$ and $<tag_1\ ,\ loc_1> = \mathcal{E}[\![\ \&o_1\ ]\!]\ (c,h,e,s)$. Note that by definition, $\mathcal{E}[\![\ \&o_1\ ]\!]\ (c,h,e,s) = \mathcal{LE}[\![\ o_1\ ]\!]\ (c,h,e,s)$. The effect of this statement on SG_i is as follows.

  - An edge annotated with $*$ from $node_{SG_i}(<tag\ ,\ loc>)$ is deleted if there is one.
  - An edge is added from $node_{SG_i}(<tag\ ,\ loc>)$ to $node_{SG_i}(<tag_1\ ,\ loc_1>)$ and the edge is annotated with $*$.

  We show that there is a corresponding edge in G_{FA} for the new edge.

  Because $<tag\ ,\ loc> = \mathcal{LE}[\![\ o\ ]\!]\ (c,h,e,s)$, by (1), there is a node x in G_{FA} such that both loc and o are in $B_{\text{FA}}(x)$.

  Because $<tag_1\ ,\ loc_1> = \mathcal{LE}[\![\ o_1\ ]\!]\ (c,h,e,s)$, by (1), there is a node y in G_{FA} such that both loc_1 and o_1 are $B_{\text{FA}}(y)$.

  Since the assignment is a pointer-related one, o and $\&o_1$ will be in an equivalence class of the PE relation; in other words, there is a node n in G_{PE} such that both o and $\&o_1$ are in $B_{\text{PE}}(n)$.

  Because $o \in B_{\text{FA}}(x)$, by Lemma A.2.2.1, $B_{\text{FA}}(x) \subseteq B_{\text{PE}}(n)$; so $loc \in B_{\text{PE}}(n)$.

  For node y in G_{FA}, there is a node m in G_{PE} such that $B_{\text{FA}}(y) \subseteq B_{\text{PE}}(m)$; so loc_1 $\in B_{\text{PE}}(m)$ and $o_1 \in B_{\text{PE}}(m)$.

Because $\&o_1 \in B_{\mathrm{PE}}(n)$ and and $o_1 \in B_{\mathrm{PE}}(m)$, by the construction of G_{PE}, there is an edge from n to m annotated with $*$. Because of this edge in G_{PE}, by the construction of G_{FA}, there is an edge from $node_{\mathrm{FA}}(o)$ (i.e., x) to $node_{\mathrm{FA}}(o_1)$ (i.e., y) annotated with $*$, which is an edge from $node_{\mathrm{FA}}(loc)$ to $node_{\mathrm{FA}}(loc_1)$.

So $SG_{(i+1)} \sqsubseteq_{\mathcal{M}_1} G_{\mathrm{FA}}$.

- l_i: $o = o_1$

  Let $<tag\ ,\ loc> = \mathcal{LE}[\![\ o\ ]\!]\ (c,h,e,s)$ and $<tag_1\ ,\ loc_1> = \mathcal{E}[\![\ o_1\ ]\!]\ (c,h,e,s)$. The effect of this statement is as follows.

  - An edge annotated with $*$ from $node_{SG_i}(<tag\ ,\ loc>)$ is deleted if there is one.
  - An edge is added from $node_{SG_i}(<tag\ ,\ loc>)$ to $node_{SG_i}(<tag_1\ ,\ loc_1>)$ and the edge is annotated with $*$.

  We show that there is a corresponding edge in G_{FA} for the new edge.

  Because $<tag\ ,\ loc> = \mathcal{LE}[\![\ o\ ]\!]\ (c,h,e,s)$, by (1), there is a node x in G_{FA} such that both loc and o are in $B_{\mathrm{FA}}(x)$.

  Because $<tag_1\ ,\ loc_1> = \mathcal{E}[\![\ o_1\ ]\!]\ (c,h,e,s)$, by (2), there are two nodes, y and z, in G_{FA} such that $loc_1 \in B_{\mathrm{FA}}(y)$, $o_1 \in B_{\mathrm{FA}}(z)$ and there is an edge from z to y annotated with $*$.

  Since the assignment is a pointer-related one, o and o_1 will be in an equivalence class of the PE relation; in other words, there is a node n in G_{PE} such that both o and o_1 are in $B_{\mathrm{PE}}(n)$.

  Because $o \in B_{\mathrm{FA}}(x)$, by Lemma A.2.2.1, $B_{\mathrm{FA}}(x) \subseteq B_{\mathrm{PE}}(n)$; so $loc \in B_{\mathrm{PE}}(n)$. Similarly $B_{\mathrm{FA}}(z) \subseteq B_{\mathrm{PE}}(n)$.

  For node y in G_{FA}, there is a node m in G_{PE} such that $B_{\mathrm{FA}}(y) \subseteq B_{\mathrm{PE}}(m)$; so $loc_1 \in B_{\mathrm{PE}}(m)$.

  Due to the edge from z to y in G_{FA}, by Lemma A.2.2.1, there is an edge from n to m annotated with $*$. Because of this edge in G_{PE}, by the construction of G_{FA}, both x (i.e., $node_{\mathrm{FA}}(o)$) and z (i.e., $node_{\mathrm{FA}}(o_1)$) have an edge annotated with $*$ to the same node. Since there is an edge from z to y annotated with $*$, there is an edge in G_{FA} from x to y annotated with $*$. So there is an edge from $node_{\mathrm{FA}}(loc)$ (i.e., x) to $node_{\mathrm{FA}}(loc_1)$ (i.e., y).

So $SG_{(i+1)} \sqsubseteq_{\mathcal{M}_1} G_{\mathrm{FA}}$.

- l_i: $s = s_1$

  Let $<tag\ ,\ loc> = \mathcal{LE}[\![\ s\ ]\!]\ (c,h,e,s)$. Let F_j be a sequence of member accessors applicable to loc and $<tag_j\ ,\ loc_j> = \mathcal{E}[\![\ apply^*(s_1,F_j)\ ]\!]\ (c,h,e,s)$. There may be more than one such F_j. The effect of this statement on SG_i is that for each F_j,

  - an edge annotated with $*$ from $node_{SG_i}(<tag\ ,\ apply^*(loc,F_j)>)$ is deleted if there is one, and
  - an edge is added from $node_{SG_i}(<tag\ ,\ apply^*(loc,F_j)>)$ to $node_{SG_i}(<tag_1\ ,\ loc_1>)$ and the edge is annotated with $*$.

  Note that by definition of B_0 and B_1, $apply^*(loc,F_j)$, $apply^*(s,F_j)$ and $apply^*(s_1,F_j)$ are all in B_1.

  We show that there is a corresponding edge in G_{FA} for each new edge.

  Because $<tag\ ,\ loc> = \mathcal{LE}[\![\ s\ ]\!]\ (c,h,e,s)$, by (1), there is a node x' in G_{FA} such that both loc and s are in $B_{\mathrm{FA}}(x')$.

  Because $<tag_1\ ,\ loc_1> = \mathcal{E}[\![\ apply^*(s_1,F_j)\ ]\!]\ (c,h,e,s)$, by (2), there are two nodes, y and z, in G_{FA} such that $loc_1 \in B_{\mathrm{FA}}(y)$, $apply^*(s_1,F_j) \in B_{\mathrm{FA}}(z)$ and there is an edge from z to y annotated with $*$.

  Because $apply^*(s_1,F_j)$ is of pointer type, the assignment is a pointer-related one; s and s_1 will be in an equivalence class of the PE relation. In other words, there is a node n' in G_{PE} such that $s \in B_{\mathrm{PE}}(n')$ and $s_1 \in B_{\mathrm{PE}}(n')$. Since $s \in B_{\mathrm{FA}}(x')$, by Lemma A.2.2.1, $B_{\mathrm{FA}}(x') \subseteq B_{\mathrm{PE}}(n')$; so $loc \in B_{\mathrm{PE}}(n')$.

  Since the PE relation is weakly right-regular, there is a node n in G_{PE} such that $apply^*(s,F_j)$, $apply^*(s_1,F_j)$, and $apply^*(loc,F_j)$ are all in $B_{\mathrm{PE}}(n)$. Because

$apply^*(s_1,F_j) \in B_{\text{FA}}(z)$, by Lemma A.2.2.1, $B_{\text{FA}}(z) \subseteq B_{\text{PE}}(n)$. For node y, there is a node m in G_{PE} such that $B_{\text{FA}}(y) \subseteq B_{\text{PE}}(m)$; so $loc_1 \in B_{\text{PE}}(m)$.

Due to the edge from z to y in G_{FA}, by Lemma A.2.2.1, there is an edge in G_{PE} from n to m annotated with $*$. Because of this edge in G_{PE}, by the construction of G_{FA}, both $node_{\text{FA}}(apply^*(loc,F_j))$ and $node_{\text{FA}}(apply^*(s_1,F_j))$ have an edge annotated with $*$ to the same node. Since there is an edge from $node_{\text{FA}}(apply^*(s_1,F_j))$ (i.e., z) to y annotated with $*$, there is an edge in G_{FA} from $node_{\text{FA}}(apply^*(loc,F_j))$ to y annotated with $*$. So there is is an edge from $node_{\text{FA}}(apply^*(loc,F_j))$ to $node_{\text{FA}}(loc_1)$ (i.e., y).

So $SG_{(i+1)} \sqsubseteq_{\mathcal{M}_1} G_{\text{FA}}$.

- l_i: HeapAlloc(o)

  Let $<tag\ , loc> = \mathcal{LE}[\![\ o\ ]\!]\ (c,h,e,s)$.

  The effect of this statement on SG_i is as follows:

  - A number of nodes are added.
  - Edges annotated with member names between the new nodes may be added.
  - An edge annotated with $*$ from $node_{SG_i}(<tag\ , loc>)$ is deleted if there is one.
  - An edge annotated with $*$ is added from $node_{SG_i}(<tag\ , loc>)$ to $node_{SG_i}(<h\ , heap_{l_i}>)$.

  Each new node corresponds to a tagged location name $<h\ , loc_j>$ such that $loc_j \in locations(heap_{l_i})$. Since $locations(heap_{l_i}) \subseteq B_1$, it is easy to prove that there is a corresponding node in G_{FA} for each new node and there is a corresponding edge in G_{FA} for each new edge annotated with a member name between new nodes. A similar proof has been given for l_i: entry of P.

  Since this statement is considered as a pointer-related assignment: $o = \&heap_{l_i}$ in the construction of G_{FA}, the argument that there is an edge annotated with $*$ from $node_{\text{FA}}(loc)$ to $node_{\text{FA}}(heap_{l_i})$ in G_{FA} is similar to the one for a pointer assignment with & on right hand side (see the argument for l_i: $o = \&o_1$).

So $SG_{(i+1)} \sqsubseteq_{\mathcal{M}_1} G_{\text{FA}}$.

- l_i: HeapDealloc(o)

  The effect of this statement on SG_i is that a number of nodes are deleted.

  Trivially $SG_{(i+1)} \sqsubseteq_{\mathcal{M}_1} G_{\text{FA}}$.

- l_i: if (o) (goto l_{true}) (goto l_{false})

  This statement has no effect on SG_i.

  Trivially $SG_{(i+1)} \sqsubseteq_{\mathcal{M}_1} G_{\text{FA}}$.

- l_i: goto l'

  This statement has no effect on SG_i.

  Trivially $SG_{(i+1)} \sqsubseteq_{\mathcal{M}_1} G_{\text{FA}}$.

We have proved that no matter what statement is at l_i, $SG_{(i+1)} \sqsubseteq_{\mathcal{M}_1} G_{\text{FA}}$.

□

References

[AFL95] Alexander Aiken, Manuel Fahndrich, and Ralph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 174–185, June 1995.

[AL94a] Rita Altucher and William Landi. Call graph analysis in the presence of function pointers. presentation at Siemens Corporate Research, April 1994.

[AL94b] Rita Altucher and William Landi. Personal communication. Feb. 1994.

[AL95] Rita Altucher and William Landi. An extended form of must alias analysis for dynamic allocation. In *Conference Record of the 22nd ACM Symposium on Principles of Programming Languages*, pages 74–84, Jan. 1995.

[And94] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, Department of Computer Science, University of Copenhagen, may 1994.

[APS93] Ole Agesen, Jens Palsberg, and Michael Schwartzback. Type inference of self: Analysis of objects with dynamic and multiple inheritance. In *Proceedings of ECOOP'93*, July 1993.

[Ash96] Michael Ashley. A practical and flexible flow analysis for higher-order languages. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages*, pages 184–194, January 1996.

[Ash97] Michael Ashley. The effectiveness of flow analysis for inlining. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 199–111, June 1997.

[Ban78] J. Banning. *A method for determining the side effects of procedure calls*. PhD thesis, Department of Electrical Engineering, Stanford University, 1978.

[BCCH95] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Lecture Notes in Computer Science*, number No. 892, pages 234–250. Springer-Verlag, 1995. Proceedings from the 7th International Workshop on Languages and Compilers for Parallel Computing.

[BS96] David Bacon and Peter Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th Annual Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA'96)*, October 1996.

[BTV96] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von neumann machines via region representation inference. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages*, pages 171–183, January 1996.

[CBC93] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the 20th ACM Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Jan. 1977.

[CCHK90] D. Callahan, A. Carle, M. W. Hall, and K. Kennedy. Constructing the procedure call multigraph. *IEEE Transaction on Software Engineering*, 16(4):483–487, April 1990.

[CL97] K. Cooper and J. Lu. Register promotion in C programs. In *Proceedings of SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 308–319. ACM, June 1997.

[Coo71] S. A. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[Coo85] K. Cooper. Analyzing aliases of reference formal parameters. In *Conference Record of the 12nd ACM Symposium on Principles of Programming Languages*, pages 281–290, Jan. 1985.

[Coo89] B. Cooper. Ambitious data flow analysis of procedural programs. Master's thesis, University of Minnesota, May 1989.

[Cou86] D. Coutant. Retargetable high-level alias analysis. In *Conference Record of the 13th ACM Symposium on Principles of Programming Languages*, pages 110–118, Jan. 1986.

[CR82] A. 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.

[CU90] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: optimizing dynamically-typed object-oriented programs. In *Proceedings of SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 150–164, June 1990.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 296–310, June 1990.

[Deu90] Alain Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Conference Record of The 17th ACM Symposium on Principles of Programming Languages*, pages pp157–168, Jan. 1990.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 230–241, June 1994.

[DGC98] Greg DeFouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In *Conference Record of the 25th ACM Symposium on Principles of Programming Languages*, pages 222–236, January 1998.

[DMW98] Saumya Debray, Robert Muth, and Mattew Weippert. Alias analysis of executable code. In *Conference Record of the 25th ACM Symposium on Principles of Programming Languages*, pages 12–24, Jan. 1998.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th Annual Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA'97)*, pages 108–124, October 1997.

[GH96] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages*, pages 1–15, Jan. 1996.

[GH98] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Conference Record of the 25th ACM Symposium on Principles of Programming Languages*, pages 121–133, Jan. 1998.

[Ghi92] Rakesh Ghiya. Interprocedural analysis in the presence of function pointers. ACAPS Technical Memo 62, McGill University, School of Computer Science, December 1992.

[Ghi95] Rakesh Ghiya. Practical techniques for interprocedural heap analysis. Master's thesis, McGill University, Montreal, Canada, march 1995.

[Gua88] Vincent A. Guarna. A technique for analyzing pointer and structure references in parallel restructuring compilers. In *Proceedings of the International Conference on Parallel Processing*, pages 2:212–220, 1988.

[Har89] Williams Ludwell Harrison. The interprocedural analysis and automatic parallelization of scheme programs. *LISP and Symbolic Computation*, 2(3-4):179–396, Oct. 1989.

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, 1977.

[HHN92a] Laurie Hendren, Joseph Hummel, and Alexandru Nicolau. Abstractions for recursive pointer data structures: improving the analysis and transformation of imperative programs. In *Proceedings of SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 249–260, June 1992.

[HHN92b] Joseph Hummel, Laurie Hendren, and Alexandru Nicolau. Applying an anstract data structure description approach to parallelizing scientific pointer programs. In *Proceedings of 1992 international conference on parallel processing*, pages 100–104, August 1992.

[HK93] Mary W. Hall and Ken Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, 1993.

[HM97] Nevin Heintze and David McAllester. Linear-time subtransitive control flow analysis. In *Proceedings of SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 261–272, June 1997.

[HN89] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. In *International Conference on Parallel Processing*, pages (II) 49–56, 1989.

[HN90] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Transaction on Parallel and Distributed Systems*, 1(1):35–47, 1990.

[HPR89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 28–40, June 1989.

[JM81] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of lisp-like structures. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. Prentice-Hall, Inc., 1981.

[JM82] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *ACM Symposium on Principles of Programming Languages*, pages 66–74, 1982.

[JW95] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Conference Record of the 22nd ACM Symposium on Principles of Programming Languages*, pages 393–407, January 1995.

[KR88] Brian W. Kernighan and Dennis M. Richie. *The C Programming Language.* Prentice-Hall,Englewood Cliffs,NJ, 2nd edition, 1988.

[Lak93] Arun Lakhotia. Constructing call multigraphs using dependence graphs. In *Conference Record of the 20th ACM Symposium on Principles of Programming Languages*, pages 273–284, January 1993.

[Lan92] William Landi. Undecidability of static analysis. *ACM letters on programming languages and systems*, 1:323–337, 1992.

[Lar91] James R. Larus. Compiling lisp programs for parallel execution. *LISP and Symbolic Computation*, 4:29–99, Jan. 1991.

[LH88] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 21–34, June 1988.

[LM96] C. Luk and T. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Archetectural Support for Programming Languages and Operating Systems*, pages 222–233, Oct. 1996.

[LR91] William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem classification. In *Conference Record of the 18th ACM Symposium on Principles of Programming Languages*, pages 93–103, Jan. 1991.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of 1992 ACM Symposium on Programming Language Design and Implementation*, June 1992.

[LRZ93] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.

[MLR+93] T. J. Marlowe, W. A. Landi, B. G. Ryder, J. D. Choi, M. G. Burke, and P. Carini. Pointer-induced aliasing: a clarification. *ACM SIGPLAN Notices*, 28(9):67–70, 1993.

[Mye81] E. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the 8th ACM Symposium on Principles of Programming Languages*, pages 219–230, Jan. 1981.

[NN97] Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: A collecting semantics for closure analysis. In *Conference Record of the 24th ACM Symposium on Principles of Programming Languages*, pages 332–345, January 1997.

[NPD87] A. Neirynck, P. Panangaden, and A. Demers. Computation of aliases and support sets. In *Conference Record of the 14th ACM Symposium on Principles of Programming Languages*, pages 274–283, Jan. 1987.

[PC94] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the 9th Annual Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA'94)*, pages 324–340, October 1994.

[PR94] Hemant D. Pande and Barbara G. Ryder. Static type determination for C++. In *Proceedings of USENIX 6th C++ Technical Conference*, pages 85–97, April 1994.

[PRL91] Hemant D. Pande, Barbara G. Ryder, and William Landi. Interprocedural Def-Use associations in C programs. In *Proceedings of the Fifth Testing, Analysis, and Verification Symposium*, October 1991.

[Ram94] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, September 1994.

[RM88] C. Ruggieri and T.P. Murtagh. Lifetime analysis of dynamically allocated objects. In *ACM Symposium on Principles of Programming Languages*, pages 285–293, 1988.

[RRH92] Anne Rogers, John Reppy, and Laurie Hendren. Supporting spmd execution for dynamic data structures. In *Conference Records of the 5th Workshop on Languages and Compilers for Parallel Computing*, pages 123–134, August 1992. use continuation-pass-style.

[Ruf95] Erik Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 13–22, june 1995.

[Ruf97] Erik Ruf. Partitioning dataflow analyses using types. In *Conference Record of the 24th ACM Symposium on Principles of Programming Languages*, Jan. 1997.

[Ryd79] Barbara G. Ryder. Constructing the call graph of a program. *IEEE Transaction on Software Engineering*, 5(3):216–226, May 1979.

[SFRW90] S. Sagiv, N. Francez, M. Rodeh, and R. Wilhelm. A logic-based approach to data flow analysis problems. In *LNCS 456,International Workshop on Programming Language Implementation and Logic Programming*, pages 277–292. Springer-Verlag, 1990.

[SH97a] Marc Shapiro and Susan Horwitz. The effects of the precision of pointer analysis. In *Proceedings of the Fourth International Symposium on Static Analysis*, September 1997.

[SH97b] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of the 24th ACM Symposium on Principles of Programming Languages*, pages 1–14, January 1997.

[Shi88] Olin Shivers. Control flow analysis in scheme. In *Proceedings of SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 164–174, June 1988.

[SRLZ98] Philip Stocks, Barbara Ryder, William Landi, and Sean Zhang. Comparing flow and context sensitivity on the modification side effect problem. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 11–20, March 1998.

[SRW96] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages*, pages 16–31, Jan. 1996.

[Ste95] Bjarne Steensgaard. Points-to analysis in almost linear time. Technical Report MSR-TR-95-08, Microsoft Research, March 1995.

[Ste96a] Bjarne Steensgaard. Points-to analysis by type inference of prorams with structures and unions. In *International Conference on Compiler Construction*, number 1060 in Lecture Notes in Computer Science, pages 136–150. Springer-Verlag, April 1996.

[Ste96b] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[SZ94] Dan Stefanescu and Yuli Zhou. An equational framework for the flow analysis of higher-order functional languages. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, June 1994.

[TAFM97] Paolo Tonella, Giuliano Antoniol, Roberto Fiutem, and Ettore Merlo. Flow-insensitive C++ pointers and polymorphism analysis and its application to slicing. In *Proceedings of the 19th International Conference on Software Engineering (ICSE97)*, pages 433–443, 1997.

[Tar83] Robert E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.

[TT94] Mads Tofte and Jean-Pierre Taplin. Implementation of the typed call-by-value lambda calculus using a stack of regions. In *Conference Record of the 21st ACM Symposium on Principles of Programming Languages*, pages 188–201, January 1994.

[Wei80] W. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Conference Record of the 7th ACM Symposium on Principles of Programming Languages*, pages 83–94, Jan. 1980.

[WL95] Robert Wilson and Monica Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 1–12, june 1995.

[ZRL96] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 81–92, October 1996.

Vita

Xiang-xiang Sean Zhang

1986	B.S. in Computer Science, Tsinghua University, Beijing, China.
1986-89	Graduate Student, Department of Computer Science, Tsinghua University, Beijing, China.
1989-92	Teaching Assistant, Department of Computer Science, Rutgers, The State University of New Jersey, New Brunswick, New Jersey.
1992	M.S. in Computer Science, Rutgers, The State University of New Jersey.
1992-96	Research Assistant, Department of Computer Science, Rutgers, The State University of New Jersey.
1996-98	Graduate Student, Department of Computer Science, Rutgers, The State University of New Jersey.
1993	W. Landi, B. G. Ryder, and S. Zhang. Interprocedural Modification Side Effect Analysis with C Programs. In *Proceedings of SIGPLAN'93 Conference on Programming Language Design and Implementation*, June 1993.
1996	S. Zhang, B. G. Ryder, and W. Landi. Program Decomposition for Pointer Aliasing: A Step toward Practical Analyses. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October 1996.
1998a	P. Stocks, B. G. Ryder, W. Landi, and S. Zhang. Comparing Flow and Context Sensitivity on the Modification Side Effect Problem. In *Proceedings of the International Symposium on Software Testing and Analysis*, March 1998.
1998b	S. Zhang, B. G. Ryder, and W. Landi. Experiments with Combined Analysis for Pointer Aliasing. In *Proceedings of the ACM SIGPLAN Workshop on Program Analysis for Software Tools and Engineering*, June 1998.
1998	Ph.D. in Computer Science, Rutgers, The State University of New Jersey.