COMPILE TIME ANALYSIS OF C AND C^{++} SYSTEMS

BY HEMANT D. PANDE

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

Dr. Barbara G. Ryder

and approved by

____

____

____

____

New Brunswick, New Jersey

May, 1996

ABSTRACT OF THE DISSERTATION

Compile Time Analysis of C and C^{++} Systems

by Hemant D. Pande

Dissertation Director: Dr. Barbara G. Ryder

We present a new approach using data flow techniques to solve compile time analysis problems for languages with general purpose pointer usage. We solve the def-use associations for C and type determination for C^{++} as representative problems. A study of the close interaction of aliasing with these problems has led us to the development of a unified approach to solve them simultaneously with aliasing, as against a factored approach which may lead to significant loss of precision. These problems are fundamental for analysis because they provide important semantic information whose precision can greatly affect the quality and utility of almost all other compile time analyses. Our polynomial algorithms are approximate, which is expected since we have shown the NP-hardness of the problems we are solving. The robust empirical results on actual programs validate our analysis approach and demonstrate its utility.

Def-use analysis links possible value setting statements for a variable (i.e., definitions) to potential value-fetches (i.e., uses) of that value. Def-use associations are thus, compile time calculable data dependences, necessary for software engineering applications such as data flow testing coverage, static slicing techniques and integrating non-interfering versions of programs. Ours is the first interprocedural def-use associations algorithm which accounts for pointer usage and yields reasonable accuracy.

Type determination calculates at compile time, the possible types of objects to which a pointer may point during some execution of the program. In C^{++}, the type of object pointed to by the receiver at a virtual call site dynamically determines the function to be invoked. Type determination enables us to replace this late binding with a direct call to an appropriate function or with inlined code in suitable circumstances, thereby eliminating the late binding overhead and improving execution efficiency. We show how our work particularly benefits architectures which use deep pipelining and branch prediction. We also bring out its utility in building a more precise call graph and improving the efficacy of subsequent analyses for C^{++}. We present a combined algorithm for aliasing and type determination for C^{++}: the first data flow technique for the problem for an object-oriented language.

Acknowledgements

I thank my adviser Barbara Ryder for the support and encouragement throughout my stay at Rutgers. I thank the committee (Bill Landi, Don Smith and Miles Murdocca), the members of the PROLANGS group (particularly Sean Zhang, Javier Elices, Jyh-Shiarn Yur, Vince Sgro and Phil Stocks), and my colleagues at TRDDC (Ashok Sreenivas, Rakesh Ghiya, Ulka Shrotri and R. Venkatesh), who were ever willingly available for active help, comments and feedback. Special thanks are due to Bill Landi and Tom Marlowe for the countless discussions and invaluable feedback on this research. I also thank TRDDC (particularly E. C. Subbarao and K. V. Nori) and Siemens Corporate Research for the use of their resources. I thank all my family and friends who helped me in various ways to reach this milestone. And finally a special thanks to my wife Varsha; I simply would not have been able to do it without her relentless support and patience.

Dedication

To the memory of my father.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Historically, compile time analysis has been used in an intraprocedural context for code optimizations. In the past decade, the emphasis has shifted towards extending compile time analysis techniques to obtain interprocedural information [BC92, BCCH94, Ban79, Bar78, Bur90, Cal88, CBC93, CK88, CK89, EGH94, HS94, HRB90, LRZ93, MLR+93, LR92, Lom77, Mye81]. This information is being employed for applications which go beyond code optimizations, such as construction of effective debugging, testing and maintenance tools [AG96, FW85, HS89, HRB90, OW91, RW85, Wei84, YHR92]. These applications were largely successful in the *Fortran* model of interprocedural communication. Nevertheless, until recently software tools either have not performed interprocedural static analysis or have employed grossly approximate techniques for languages with pointers. Pointers are difficult to analyze because the *address* of a storage location can become the *value* of a pointer variable, and therefore, can be transferred by an assignment statement. An *alias* occurs at a program point when more than one name exists for the same storage location. In the presence of pointers, as in languages like C and C^{++}, aliases are potentially created and destroyed at every pointer assignment. The *Fortran* model of aliasing fails in this context in two ways: (i) aliases cannot be created intraprocedurally during execution of a *Fortran* procedure, (ii) aliases in the calling procedure in *Fortran* cannot be affected by activity in the called procedure. Both (i) and (ii) can occur in C and C^{++} programs.

The research in this thesis was inspired by Landi's work on aliasing for C [Lan92a], which enjoys the distinction of being the first practically successful flow sensitive pointer aliasing algorithm. He showed the theoretical difficulty of intraprocedural aliasing and introduced a new technique, called *conditional analysis*, for precise interprocedural alias analysis for C programs restricted to single level pointers[1] [LR91]. Informally, we state the idea of performing conditional alias analysis in the presence of pointers as: to analyze the code in a procedure under certain input assumptions and then combine the results of these conditional analyses for those assumptions that actually may occur during program execution. The conditional analysis technique answers the question: *If there is a path to the entry of the procedure containing node n, on which condition P holds, can fact Q hold at n?* Landi and Ryder also developed a safe, approximate algorithm to solve the *may aliasing* problem for general purpose pointer usage in a subset of C, one which included most language constructs contributing to the difficulty of the problem [LR92, Lan92a].

In this thesis, we show how to apply the conditional analysis technique to solve the def-use associations problem in C and a combined type determination and aliasing problem for C^{++}. This thesis is part of the first comprehensive attempt to solve these problems in a uniform manner. Our polynomial time algorithms are approximate, which is expected since we have also shown the *NP*-hardness of the problems we are solving. Nevertheless, because the aliasing information we are using is specific to a program point (rather than having the same aliases hold throughout a procedure as in *Fortran*), we are confident that sufficient accuracy is obtained to ensure the utility of our analysis, as demonstrated by our empirical results.

Determining data dependences precisely in the presence of pointers is crucial to the construction of effective software tools mentioned above. Our interprocedural def-use analysis for C programs is an important first step in practically providing accurate, compile time semantic information. Def-use analysis links possible value setting statements for a variable (i.e., definitions) to potential value-fetches (i.e., uses) of that value. Def-use associations are thus, compile time calculable data dependences. Informally speaking, the analysis proceeds by calculating those definitions of variable x, which may have assigned the current value of x when program point n is executed. If x is used at n,

[1]By limiting programs to one level of indirection we mean, for example, that *int ** variables can occur but not *int *** variables, and recursive structures (e.g. linked lists) are not allowed.

then each definition is linked to this use, forming a def-use association. The calculation described involves solution of the interprocedural reaching definitions problem, which is complicated by assignments through dereferenced pointers (e.g., `*p = ...`) and procedure calls. Ours is the first interprocedural reaching definitions algorithm that accounts for pointer usage and yields reasonable accuracy as demonstrated by our implementation results.

Encouraged by the results of analyzing C, we then concentrated on how to extend interprocedural static analysis techniques beneficially to the programming paradigm of object-orientedness. For us, C^{++} was the natural choice because of its proximity to C and its popularity as a representative object-oriented language, and last but not the least, a total absence of effective compile time analysis techniques for C^{++}.

Our efforts are focused on developing new techniques to handle those features distinguishing C^{++} from C, such as inheritance and virtual functions (i.e., object-orientedness), subtyping and overloading (i.e., polymorphism). Most significant are virtual functions, because the type of receiver at a virtual call site dynamically determines the function to be invoked. *Type determination* enables us to replace late binding with a direct call to an appropriate function, or with inlined code in suitable circumstances, thereby eliminating the overhead of late binding and improving execution efficiency.

Recent empirical studies of dynamic behavior of C^{++} programs indicate that there is opportunity to avoid late bindings in many cases, which is particularly significant for architectures which employ deep pipelining [CG94]. A uniquely resolved call site would eliminate pipeline stalls, as the target of the call is unambiguously known. A related empirical study of behavioral differences between C and C^{++} programs reveals that C^{++} functions are usually small (in terms of lines of code) and C^{++} programs contain more calls than C programs [CGZ95]. The higher calling frequency usually degrades analysis performance because of the inherent difficulty of interprocedural analysis contrasted with that of traditional intraprocedural analysis. Therefore, these findings suggest that inlining a uniquely resolved function may be a crucial optimization. Even when unique resolution is not possible, short-listing the functions may allow the compiler to replace the late binding mechanism of virtual call with appropriate function calls within a decision statement. Branch prediction techniques may then be applied to improve the execution performance of the program. Additionally, short-listing can focus further analyses on selected functions, rather than on the entire pool of functions with the same name, potentially saving analysis time and allowing building of a more precise call graph. Also, exclusion of the statically non-invocable functions from analysis can eliminate spurious side effects, thereby improving the precision of subsequent analyses.

Static type determination is significant in analyzing C^{++} also because it represents important semantic information whose precision can greatly affect the quality and utility of various other static analyses. First, we developed a static type determination algorithm for C^{++} programs restricted to using only single level pointers [PR94a]. In this context, type determination can be solved independent of aliasing. However, in the presence of multiple level pointers, these two problems cannot be solved separately, but are interdependent. Secondly, we have designed a combined approximation algorithm that solves type determination and aliasing simultaneously for C^{++} programs. Ours is the first data flow technique for this combined problem in an object-oriented language.

Since all C^{++} programs can be source-to-source transformed into C programs, if we claim to be able to analyze C, should we not be able to analyze C^{++} programs in their C incarnation? Actually, this is not desirable because the distinguishing C^{++} constructs map to C constructs so general that gross approximations in analysis would be inevitable. In particular, the virtual function mechanism can be expressed in terms of function calls through arrays of function pointers. Algorithms which attempt precise analysis in the presence of function pointers and procedure variables handle only a limited usage of such constructs or resort to possibly worst case exponential analyses [BCCH94, EGH94, HK93, Lak93, Ryd79, WL95]. In general, precise compile time analysis in the presence of function pointers is an intractable problem [ZR94]. This motivated us to develop new techniques to analyze virtual functions in the C^{++} domain itself. When there is no increase in generality, we reduce a C^{++} construct to a semantically equivalent C construct. For example, we treat a *class constructor* as a `new` followed by appropriate initializations and we express the principle of encapsulation using the concepts of scope and visibility in C.

Major contributions of the thesis:

1. extension of the conditional alias analysis technique to solve fundamental compile time analysis problems such as def-use associations and type determination; a study of the close interaction of aliasing with these problems led to the development of a unified approach to solve each problem simultaneously with aliasing as against a factored approach which leads to loss of precision.
2. the first polynomial time approximation algorithm for def-use associations in C programs.
3. a combined approximation algorithm for aliasing and type determination for C^{++}: the first data flow technique for the problem for an object-oriented language.
4. robust empirical results on actual programs to validate our analyses approaches and demonstrate their utility,
5. theoretical proofs of the NP-hardness of these problems.
6. monotone data flow framework definition for type determination and aliasing for C^{++}: the first successful effort to fit into the standard lattice theoretical framework a practical, approximation algorithm for an intractable interprocedural data flow analysis problem involving pointers.

Limitations on the language constructs handled: Our research for C assumes a restricted subset of the language which excludes pointers to functions, casting[2], union types, exception handling, *setjump* and *longjump*. We allow arrays and pointer arithmetic; however, we simply treat arrays as aggregates and assume pointer arithmetic stays within array bounds. The C^{++} programs we analyze have all the restrictions of our C analysis, except we do handle type casting within class hierarchy. At this time, our C^{++} algorithm does not handle class copies (when a class object is assigned or passed as parameter to another class object) and constructs which create circular data structures (such as circularly linked lists). We also assume that any template usage

[2]However we do handle simple casting for `p = malloc()`.

(recently added to C^{++} language definition) has been expanded before we analyze the program.

We represent an array element as a dereference of the array treated as a pointer. For example, we represent a[i] as *a. As a result, we cannot distinguish between different elements of an array. For the sake of safety, we must assume that an assignment to *a updates an indeterminate element of a and preserves the others. Also, we transform the *reference* variables in C^{++} programs to pointers and treat the usage of such variables accordingly. Having reduced these language constructs to pointers in this manner, we will omit the special cases pertaining to them from the algorithm description.

1.1 Thesis Overview

In Chapter 2, we describe our program representation and define terminology used throughout this thesis. The main body of this thesis is divided into two parts. In Chapters 3 and 4, we concentrate on the def-use associations analysis of C. In Chapters 5 and 6, we discuss our compile time analysis approach for C^{++} programs.

Chapter 3 provides details of our approach to calculate interprocedural def-use associations for C: In Section 3.1, we introduce additional terminology specific to our C analysis. Here, we also show the theoretical difficulty of the problem. Section 3.2 includes a discussion of the *may-hold* predicate and the *must-hold* function, and how they relate to alias analysis. In Section 3.3, we describe our interprocedural reaching definitions algorithms first without and then with aliasing effects due to single level pointers. In Section 3.4, we outline how this information is used to calculate def-use associations. A straightforward extension to account for multiple level pointers follows immediately in Section 3.5. We describe our implementation techniques and report empirical results in Chapter 4.

In Chapter 5, we present our combined type determination and aliasing analysis for C^{++} programs. In Section 5.1, we establish the intractability of the problem. Section 5.2 includes a precise but intractable formulation, followed by an approximation formulation for the combined type determination and aliasing problem. We use the latter formulation as the basis of our algorithms. Section 5.3 exploits the separability of aliasing and type determination in a scenario where only single level pointers are allowed and includes a description of type determination aspects of our algorithm. In Section 5.4, we describe our combined approximation algorithm for type determination and aliasing for C^{++} programs with general purpose pointer usage. Chapter 6 includes a discussion on theoretical aspects of our algorithm and empirical results of our prototype implementation.

In Chapter 7, we examine the significance of def-use analysis in various phases of software life cycle, and explore relevant prior work. We discuss how the precision of def-use information directly impacts the efficacy of software tools. Following that, we provide an overview of related work on optimizing in the presence of dynamically dispatched calls in object-oriented programming languages. We also discuss relevant literature on analysis in the presence of pointers as it represents an important aspect of our work. The summary of this thesis and our plans for future work appear in Chapter 8.

Throughout this thesis, we have chosen to describe our algorithms in detail for the restricted case of single level pointers first, and then extend them to the general pointer usage. This decision reflects our experience that major theoretical difficulties in solving these problems for programs with multiple level pointers are inherent for programs with single level pointers. Indeed, the proofs of NP-hardness of the problems under consideration only require the presence of single level pointers.

Chapter 2

Problem Representation

This chapter defines concepts such as intermediate program representation and object names, which are common to our analyses of both C and C^{++}. We also explain the different nuances of some of these concepts in the context of the two analyses.

2.1 Interprocedural Control Flow Graph

A *control flow graph* (CFG) for a function consists of nodes which represent single-entry, single-exit regions of executable code, and edges which represent possible execution branches between code regions. We represent a program with an interprocedural control flow graph (ICFG), which intuitively is the collection of CFGs for the individual functions comprising the program, connected with edges representing function calls [LR91, PR94a]. Formally, an ICFG is a triple ($\mathcal{N}$, $\mathcal{E}$, ρ) where $\mathcal{N}$ is the set of nodes, $\mathcal{E}$ is the set of edges and ρ is the *entry* node for `main`. $\mathcal{N}$ contains a node for each simple statement in the program, an *entry* and *exit* node for each function, and a *call* and *return* node for each invocation site. An intraprocedural edge into a *call* node represents the execution flow into an invocation site, while an intraprocedural edge out of a *return* node represents control flow from an invocation site once the invoked[3] function has returned. For a non-virtual function call, we represent the control flow into the called function by an interprocedural edge from a *call* node to the corresponding *entry* node. Similarly, we represent the return of control from the called function by an interprocedural edge from the *exit* node to the *return* node[4]. However, virtual function

[3] We use the terms *call* and *invocation* interchangeably.

[4] This representation of a program is similar to those of [Mye81], although we use it differently during analysis.

invocation makes it impossible to determine the correspondence between a *call* and *entry* before analysis, since the function invoked depends on the type of the receiver at the call site. Establishing the interprocedural edge(s) from a *call* node representing a virtual function invocation to appropriate *entry* node(s) and from *exit* node(s) to the corresponding *return* node(s), (i.e., resolving virtual functions), is a major goal of our algorithms for C^{++}.

We create a unique global variable for each value-returning function. The semantic effect of a function returning some value at a call site is captured by (i) *assigning* the return value to this global variable at an immediate predecessor of the exit node of the called function, and (ii) appropriately *using* the variable at the immediate successor of the corresponding return node. All implementations of a value-returning virtual function, say `f`, share a single global variable. Since a virtual call to `f` may actually invoke any implementation of `f`, this variable facilitates using the returned value of all the different implementations via a common mechanism.

A path in a flow graph is used to represent a program execution. Unfortunately, paths in the graph sometimes do not correspond to possible executions. Intraprocedurally, this can result from interactions of conditionals. For example, in:

n_1:	`if (a > 0)`
n_2:	`    b = 1;`
n_3:	`if (a < 0)`
n_4:	`    b = 2;`

the path $n_1 n_2 n_3 n_4$ can never be executed. The problem of determining which paths are executable can be shown equivalent to solving the Turing machine halting problem and thus is undecidable (i.e., the set of executable paths is not recursive). As is traditional with data flow analysis [Hec77], we will overestimate, assuming all intraprocedural paths are executable. During execution when a procedure finishes, it returns to the call site that invoked it, but paths in an ICFG do not guarantee this property.

We introduce some ICFG related terminology below.

- An ICFG path is *realizable* if, whenever a called function on this path returns, it returns to the corresponding *return* node of the call site which invoked it [LR91].

Clearly, not all ICFG paths are realizable. We refer to paths that are not realizable as *unrealizable*.

- A realizable path is *balanced* if for each intermediate *call* node, the path contains a corresponding *return* node representing the return of control from the called function[5]. Intuitively, the first and the last node on a balanced path belong to the same function. Moreover, they are in the same incarnation of that function since every called function on the path (perhaps recursively) must return control before the path terminates.

- A realizable path from ρ is called *consistent* if, for every edge $\ll call, entry \gg$ on the path, where *call* represents a virtual call with receiver *rec*, the execution defined by the subpath from ρ to *call* implies a pointer-type pair $<rec \Rightarrow C>$ at *call* such that the virtual function represented by *entry* is invocable from *call*. We refer to ICFG paths that are not consistent as *inconsistent*. Inconsistent paths do not correspond to any possible execution sequence; hence, we have designed our algorithms to usually avoid propagating information along inconsistent paths. Note that in the context of C without function pointers and virtual functions, each realizable path is consistent.

If we consider the sub-sequence of a consistent path consisting of those call nodes without corresponding return nodes, we have the *call stack*, which can be used to obtain the precise interprocedural connections in an ICFG path. With full knowledge of the call stack, a compile-time algorithm can avoid propagating information on inconsistent paths. For efficiency, in our algorithm we do not maintain the full call stack, but rather use an abstraction of the call stack that is sufficiently powerful to preserve the consistency of paths (Call stacks are discussed again in Section 3.2).

A useful call stack abstraction, must allow the following:

1. Given an abstraction and a call site (call node), compute the abstractions, possibly

[5]We defined the terms *realizable* and *balanced* paths independently [LR91, PR94b], and later found that the ideas already existed in literature, referred to as *valid* and *complete* respectively in [SP81]. A more recent paper [RHS95] also addresses these issues.

not unique, for the called procedure.

2. Given an abstraction and an exit node, compute to which call sites (return nodes) information is propagated and the abstraction of call stack on return to the calling procedure.

2.2 Object Names

While *objects* correspond to locations that can store information, *object names* provide ways to refer to them at a program point. We begin by naming each object with a unique *fixed-location name* [LRZ93]. The fixed-location name for a statically allocated (on stack) object is the same as its variable name from the declaration statement. We create a fixed-location name new_{pp} for a dynamically created object (on heap), where pp is the program point representing the creation site of the object. This method of naming heap locations was first proposed by Ruggieri and Murtagh [RM88]. A member access (using the "." operator) of a fixed-location also qualifies as a fixed-location as it uniquely identifies the contained object corresponding to the member. However, an object can be accessed with various names through pointer dereferencing. For example, if pointer p points to new_{pp} at a program point, $*p$ can be used to access new_{pp} at that program point. We use the term *object name* to refer to a fixed-location name or a name with at least one pointer dereference of a fixed-location name. Although these definitions of fixed-locations and object names are sufficient in the context of C (and have been used in our algorithms for C described in this thesis), we need more comprehensive definitions based on types for these terms for C^{++}.

The functionality of a C^{++} program is expressed in terms of interactions between *objects*. Each C^{++} object has a declared *type* which may be either a primitive type (such as *int*) or a user defined *class*[6]. We list the relevant operations needed on these types. We borrow some definitions from [Lan92a], with appropriate modifications for the C^{++} type system.

can_deref(C): returns *true* iff C can be dereferenced without type casting.

[6] Another user defined type *struct* is equivalent to a *class* whose members are public.

Rule	Production	Semantic Actions
(1)	for each $i \in \{1, \ldots, \lvert\text{temp\_tuple}\rvert\}$ $\text{OBJNAME}_i \rightarrow$ $*(\text{OBJNAME}_0)$	if **can_deref**(OBJNAME_0.type) temp_tuple = **type_tuple**(OBJNAME_0.type) OBJNAME_i.type = **deref_type**(temp_tuple[i])
(2)	$\text{OBJNAME}_1 \rightarrow$ (OBJNAME_0).**mem**	if **is_member(mem**,OBJNAME_0.type) OBJNAME_1.type = **member_type(mem)**
(3)	$\text{OBJNAME}_1 \rightarrow$ **var**	OBJNAME_1.type = **var_type(var)**

mem is any declared member of a class in the program.
var is any variable appearing in the program body or a heap variable name.

Table 2.1: Syntax-directed definition of object names

is_member(*member, C***):** returns *true* iff C is a class name **and** *member* is a direct or inherited data member of C.

is_classptr(*C***):** returns *true* iff type C is a pointer to a class[7].

type_tuple(*C***):** if **is_classptr(***C*), returns a tuple of pointers to all classes which an object of type C may point to (considering only type casts within inheritance hierarchy); otherwise, returns a single-element tuple containing C.

member_type(*member***):** returns the type of *member*.

var_type(*v***):** returns the type of variable v.

deref_type(*C***)** : if **can_deref(***C*), returns the type of objects to which an object of type C may point without any type casting.

A syntax directed definition of object names appears in Table 2.1. Our definition is an extension of object names for C defined in [Lan92a]. Note that object names derived only using rules (2) and (3) correspond to fixed-location names[8]. We also extend the set of object names in Table 2.1 with special names obtained by application of address operator to the object names (for example, &p). These names affect the aliasing and type determination calculations, but do not themselves appear in the solution.

[7]**is_classptr(***C***)** implies **can_deref(***C***)**, but not *vice versa*.

[8]In the presence of nested classes, the number of fixed-locations generated by applying a sequence of "." operator to a class object can be exponential in the nesting depth; nevertheless we assume a polynomial bound for realistic programs. Given that the number of variables used in a program is bounded above by the number of ICFG nodes, the number of object names created is polynomial in terms of the ICFG nodes.

Note that the object name $p{\rightarrow}member$ is short for $(*p).member$. Each dereference of an object name is associated with a type: the type which is being dereferenced. For example, $p\overset{B}{\rightarrow}member$ indicates that p points to an object of class B while being dereferenced to $member$[9].

We distinguish an object name from its *access path* representation [Deu94, LH88]. An access path is what typically appears in the program, without the intermediate types. For example, two distinct object names $p\overset{A}{\rightarrow}member$ and $p\overset{B}{\rightarrow}member$ have the same access path representation: `p→member`. Nevertheless, for notational convenience we omit the intermediate types in object names. For example, we write the above object names simply as $p{\rightarrow}member$ and rely on the context for the intermediate type. To avoid confusion, we will consistently write object names in *italics* and access paths in `typewriter` font. Note that for fixed-locations, there is a one-to-one correspondence between access path and object name representation, as fixed-locations involve no dereferences.

2.3 Terminology

Finally, we list the definitions for some key concepts which will be used throughout this thesis.

- An *alias* exists at a program point when two or more object names refer to the same location as a result of program execution to that program point[10]. We represent aliases by unordered pairs of object names (e.g. $<v, *p>$). The order is unimportant since the alias relation is symmetric. *a may be aliased* to b at program point n iff there exists an execution path ending at n on which a and b refer to the same location after n is executed. Because of the nature of pointers, these aliases are defined at a program point, not just at procedure entry as in the *Fortran* analysis [CK89]. *Must be aliased* is defined similarly except a and b must

[9]Throughout this thesis, whenever we refer to an object of class B, we mean an instance of B proper and no other base or derived class of B.

[10]For brevity, we use the phrase *to a program point* to mean *up to and including a program point.*

refer to the same location on all execution paths to n. The solutions to may and must alias are formally defined as:

may alias: The precise[11] solution for *interprocedural may alias* is

$$\left\{ [n, <a, b>] \;\middle|\; \begin{array}{l} \exists \text{ consistent path } \rho n_1 n_2 \ldots n_j n \text{ in} \\ \text{ICFG, on which } a \text{ and } b \text{ refer to} \\ \text{the same location} \end{array} \right\}$$

must alias: The precise solution for *interprocedural must alias* is

$$\left\{ [n, <a, b>] \;\middle|\; \begin{array}{l} \forall \text{ consistent paths } \rho n_1 n_2 \ldots n_j n \text{ in} \\ \text{ICFG, } a \text{ and } b \text{ refer to the same} \\ \text{location} \end{array} \right\}$$

A *safe* [MR90] (called conservative in [ASU86]) approximation to may alias is an overestimate of the precise may alias solution; it is safe to report all may aliases and possibly some spurious ones. For must alias, underestimates of the set of aliases are *safe*.

- At a call site, an object name is *visible* in the called function iff the called function is in the scope of the object name *and* at run time the object name refers to the same object in both the calling and called function. This means that if x is a local variable of function f, then the x at a recursive call site is not visible in f, since at run time there will be a distinct x in the new incarnation of f. We consider heap storage names to be globals.

[11] We are using the usual data flow definition of precise which means *precise up to symbolic execution*; in other words assuming all consistent paths through the program are executable [Bar78].

Chapter 3

Interprocedural Def-Use Associations for C

Informally, *reaching definitions* is the problem of statically determining all the program points where the value of a location (e.g., a variable) was last written. For example, in the code below:

```
n1:  a = x;
n2:  if (a == 0)
n3:     a = 1;
     else
n4:     a = 2;
n5:  printf("%d",a);
```

reaching definitions at n_5 tells us that a was written either at n_3 or n_4 (but **not** at n_1). Of course, this information is particularly interesting at n_5 since a is being read. Such pairings of definitions to uses are called *def-use associations.*

In this thesis, we will be using data flow analysis [ASU86, Hec77, MR90] to solve the reaching definitions problem; this analysis requires programs to be represented by rooted, directed graphs. For analysis within a single procedure (i.e., *intraprocedural analysis*) a *control flow graph* (CFG) [ASU86, Hec77][12] is generally used. A CFG consists of nodes that represent single-entry/single-exit regions of executable code and edges which represent possible execution flow between code regions. Often CFG nodes are *basic blocks*, but we assume that the nodes are individual statements. For analysis across procedure calls (i.e., *interprocedural analysis*), a *call graph* is sometimes used, in which nodes represent procedures and edges represent possible calls. Either a call graph or control flow graph can be referred to by the term *flow graph.* Unfortunately, neither

[12] In [ASU86] it is called simply a *flow graph.*

```
int *p;
int a,b,c;
R(f)
int *f;
{
  *f = 1;
  c = 1;
}

main()
{
  scanf("%d",&a);
  b = 0;
  if (a == 0) {
    p = &a;
    R(p);
    printf("%d",*p);
  } else {
    p = &b;
    R(p);
    printf("%d",b);
  }
}
```

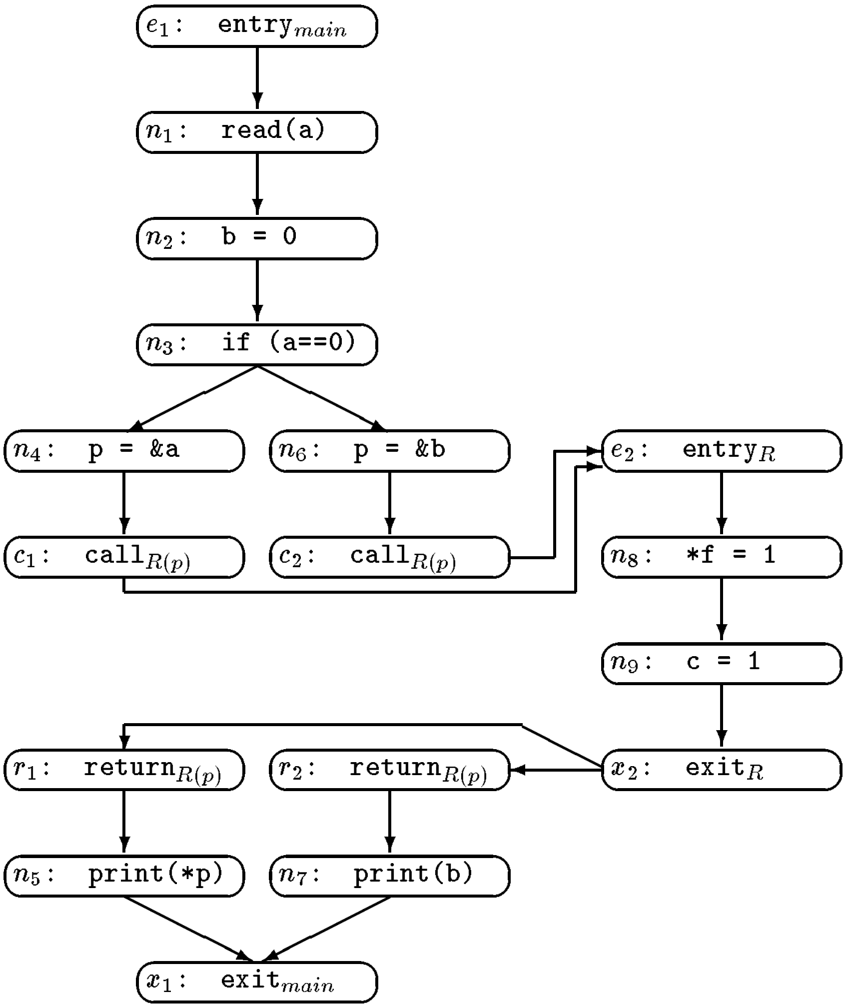


Figure 3.1: Sample Program and its ICFG

of these representations are sufficient for our purposes and we use an *interprocedural control flow graph* (ICFG) which is a hybrid of CFGs and call graphs (see Section 2.1). A sample program and its ICFG appears in Figure 3.1, and will be used as a running example throughout this chapter.

3.1 Problem Definition

3.1.1 Additional Terminology

We introduce the following terminology specific to our analysis of C, in addition to that defined in Section 2.3.

A location is *defined* when its value is set. We are only interested in the definitions of fixed-locations, because the location referred to by a dereferenced pointer ($*p$) can vary on different executions. So in the code below,

```
n1:  if (a > 0)
n2:     p = &v;
     else
n3:     p = malloc(sizeof(int));
n4:  *p = 5;
```

at n_4 we consider v and new_{n_3} defined, but not $*p$. These two definitions are represented by $< n_4 : v >$ and $< n_4 : new_{n_3} >$ respectively. Reaching definitions in the presence of pointers is the problem of statically determining all the program points where the value of a location was last written, perhaps indirectly through an alias. Formally,

reaching definition: A definition $< n : a >$ of a fixed-location a at node n *reaches* node m if there is a realizable path $nn_1n_2\ldots n_j$ such that $n_j = m$ and a is <u>not</u> redefined in any $\{n_i\}_{i=1}^{j}$.

Finally, a location is *used* when its value is read, perhaps indirectly through an alias. A def-use association in the presence of pointers is a pairing of definitions with such uses. Formally,

def-use association: $\ll < n : a >, m \gg$ is said to be a def-use association if the definition $< n : a >$ reaches a predecessor of m and m uses a.

Safe approximations for both reaching definitions and def-use associations are overestimates of the precise solution.

3.1.2 Theoretical Complexity of the Problem

Myers [Mye81] showed that many interprocedural data flow problems are *NP*-hard in the presence of aliasing. Landi and Ryder [LR91] and Larus [Lar89] proved the *NP*-hardness of the intraprocedural alias problem in the presence of multiple level pointers. Landi [Lan92b] later showed that precise intraprocedural may alias in the presence of recursive data structures (e.g., linked lists) is *undecidable* (i.e., recursively enumerable, but not recursive) and that precise intraprocedural must alias also with recursive structures is *uncomputable* (i.e., not recursively enumerable). Ramalingam [Ram94] has also reported similar results about aliasing. We show that precise solution of intraprocedural reaching definitions in the presence of single level pointers is *NP*-hard. Our proof is by reduction of the 3-SAT problem and is a variation of similar proofs in [Lan92a, Lar89, Mye81].

Theorem 3.1.1 *In the presence of single level pointers, the problem of calculating precise intraprocedural reaching definitions is* NP*-hard.*

We proceed by reducing the problem of 3–Satisfiability (3-SAT) to the above problem. Let the formula in conjunctive normal form (cnf) be made of m logic variables $\{v_1, v_2, \ldots, v_m\}$. Let the cnf be represented as

$$(l_{11} \vee l_{12} \vee l_{13}) \wedge (l_{21} \vee l_{22} \vee l_{23}) \wedge \cdots (l_{n1} \vee l_{n2} \vee l_{n3})$$

where each l_{ij} represents a *literal* : logic variable or its negation. The above cnf is satisfiable iff there exists a truth assignment to the logic variables which makes the formula *true*. As a part of our reduction, we construct a program segment polynomial in the size of 3-SAT problem and show that the cnf is satisfiable iff we can solve the intraprocedural reaching definitions problem precisely on the program segment, thereby proving that the problem is *NP*-hard. The program segment appears in Figure 3.2. We have left out the conditions of the `if` statements because we are assuming all paths through the program are executable and thus the actual conditions do not impact this example. We interpret `*v`$_i$ aliased to `true` as meaning the logic variable v_i is *true*. We explicitly assign the complement of the value of `v`$_i$ to `nv`$_i$ in the reduction. This is

```
   int *v_1,*nv_1,*v_2,*nv_2,..., *v_m,*nv_m;
   int true,false;
   const YES, NO;
/* A path through this section of code
   corresponds to a truth assignment */
L1:
   if (-) {v_1 = &true; nv_1 = &false}
       else {v_1 = &false; nv_1 = &true}
   if (-) {v_2 = &true; nv_2 = &false}
       else {v_2 = &false; nv_2 = &true}
                    . . .
   if (-) {v_m = &true; nv_m = &false}
       else {v_m = &false; nv_m = &true}
L2:
d: false = NO;
L3:
   if (-) *l_11† = YES
       else if (-) *l_12 = YES
                else *l_13 = YES;
   if (-) *l_21 = YES
       else if (-) *l_22 = YES
                else *l_23 = YES;
                    . . .
   if (-) *l_n1 = YES
       else if (-) *l_n2 = YES
                else *l_n3 = YES;

L4:
```

† l_{ij} represents $\mathtt{v}_k$ or $\mathtt{nv}_k$ for some k.

Figure 3.2: *NP*-hardness of the problem

consistent with the interpretation in propositional logic that whenever v_i is *true*, $\overline{v_i}$ is *false*, and *vice versa*. Any path from L1 to L2 is a truth assignment to the variables. The propositional formula is represented between program points L3 and L4.

Suppose the formula is satisfiable, then there exists a path from L3 to L4 on which each $*l_{ij}$ is aliased to `true`, which implies that all assignments on that path are in effect "`true = YES`". Thus the definition "`d: false = NO`" reaches L4.

Suppose the formula is unsatisfiable, then for every truth assignment there is at least one outermost `if` statement where every $*l_{ij}$ within the statement is aliased to `false`. In effect, we have an entire row of "`false = YES`" no matter which path is taken from L1 to L2. Thus "`d: false = NO`" does not reach L4.

Thus 3-SAT is polynomially reducible to intraprocedural reaching definitions with single level pointers, and we have proved Theorem 3.1.1. □

Corollary 3.1.1 *In the presence of either single or multiple level pointers, the problem of calculating precise interprocedural reaching definitions is* NP*-hard.*

3.2 May and Must Alias Information

It is easy to see how may alias information is needed to ensure safety and must alias information can be used to increase the quality of reaching definitions. This is illustrated below:

n_1: `a = 1;`

n_2: `p = &a;`

n_3: `*p = 2;`

n_4: `printf("%d",a);`

Without may alias information the def-use association $\ll < n_3 : a >, n_4 \gg$ is missed and without must alias information the definition $< n_1 : a >$ is not killed at n_3, resulting in the spurious def-use association $\ll < n_1 : a >, n_4 \gg$.

For aliasing calculations, we use the algorithms in [Lan92a]. These algorithms associate with alias information an abstraction of the call stack for which the alias information is created. This abstraction is created by a call and associated with aliases in the called procedure; it is used at procedure exit to determine to which call site(s), alias information should be propagated. The alias solution with this call stack abstraction proves to be very useful in solving for reaching definitions.

3.2.1 The *may-hold* Predicate

For may alias at a node n, the representation of the call stack is a *reaching alias*[13] that exists at entry of procedure `P` containing n when `P` is invoked. At first this may seem an odd abstraction, but in the presence of single level pointers, this abstraction precisely restricts alias information to realizable paths in the ICFG [LR91]. When multiple levels of indirection are possible, using one reaching alias rather than the set of reaching aliases that are propagated through a call, leads to approximate, but safe alias calculations [LR92].

We use the *may-hold* predicate to associate alias information with the call stack abstraction. Intuitively, *may-hold*(n, RA, $<a, b>$) where n is an ICFG node, RA is a reaching alias and $<a, b>$ is an alias, answers the question: *"If there is a path to the entry node of the procedure containing n, on which the reaching alias RA abstracts the call stack, then may a be aliased to b at n?"*. This question is asking about *conditional aliasing* information. RA can be $\emptyset$, which is not a true reaching alias, but is associated with aliases that exist regardless of the call stack; for example, for the statement "n_1: `p = &v`", we generate *may-hold*(n_1, $\emptyset$, $<*p, v>$).

To explain the use of reaching aliases, assume that all variables are globals and there are no parameters. It is straightforward to introduce name space binding functions, when needed (see Section 3.3.2, p. 32).

- Let c_1 represent a call site and RA a reaching alias at that call site, then every alias $<a, b>$ such that *may-hold*(c_1, RA, $<a, b>$) is *true*, becomes an induced reaching alias in the called procedure from that call site. These reaching aliases are associated only with themselves; i.e., *may-hold*(e_i, $<a, b>$, $<a, b>$) is *true* for e_i, the entry node of the called procedure.

[13] Reaching aliases are referred to as *assumed aliases* in [Lan92a, LR91, LR92, PRL91, PLR94].

- Given an exit node (x_1) and a RA (not $\emptyset$), the aliases associated with RA at x_1 are returned to all return nodes r_i, with reaching alias RA' such that *may-hold*(c_i, RA', RA) is *true*. If RA is $\emptyset$, then the associated alias *alias-pair* is returned to all call sites (i.e., *may-hold*(r_i, $\emptyset$, *alias-pair*) is *true* for all r_i).

The *may-hold* solution for the program in Figure 3.1 appears in Table A.1 of Appendix A.2 and the may alias solution is in Table A.2. In terms of aliasing, it is a simple example, although it does involve parameter passing; however, it does illustrate why *may-hold* information is more useful than may alias information. In particular, at n_8 (`*f = 1`) with may alias information we can correctly conclude that a is defined. However, when `R` returns we are unable to determine to which call sites $< n_8 : a >$ should be returned. If we use *may-hold*, then we can associate the reaching alias ($<*f, a>$) with the definition $< n_8 : a >$. This allows us to correctly return the definition to r_1 while safely avoiding returning it to r_2.

3.2.2 The *must-hold* Function

To represent conditional must alias information, we use the *must-hold* function. The conditional must alias set for a node n_i of the ICFG and an *alias-pair* is the unique minimal set of pairs that must be aliased at the entry node of the procedure containing n_i which ensures that *alias-pair* must be aliased at n_i. If no such set exists, we say *must-hold*(n_i, *alias-pair*) is $\perp$. This implies that *alias-pair* cannot be aliased on all paths to node n_i, regardless of what must alias set exists at entry of the procedure containing n_i. By contrast, *must-hold*(n_i, *alias-pair*) $= \emptyset$ implies that without requiring any assumptions at the entry node, *alias-pair* must be aliased at node n_i.

In Figure 3.3, *must-hold*(x_1, $<*r, s>$) $= \{<*p, s>, <*q, s>\}$, since both pairs must be aliased at entry to ensure $<*r, s>$ must be aliased at x_1, no matter which path is taken. However, *must-hold*(n_1, $<*r, s>$) $= \perp$ since no set at entry can ensure that $<*r, s>$ must be aliased immediately after node n_1 is executed and *must-hold*(n_3, $<*r, *p>$) $= \emptyset$, since this alias always exists immediately after n_3 is executed.

must-hold is computed by setting up a system of equations and solving them with

```
      int *p,*q,*r,s;
e1:   A () {
n1:     r = NULL;
n2:     if (s == 0)
n3:       r = p;
        else
n4:       r = q;
x1:   }
```

Figure 3.3: A program segment

fixed-point iteration. We do not go into the details of this algorithm in this thesis, but *must-hold* is polynomial-time computable in the number of ICFG nodes [Lan92a]. For a must alias problem, an abstraction of the call stacks must embody *all* the call stacks on which the data flow information occurs. An obvious choice is the value of *must-hold*(n_i, *alias-pair*) as the abstraction of all call stacks on which *alias-pair* is a must alias. Thus, the call stack abstraction for *must-hold* is a set of aliases rather than one alias pair.

The *must-hold* solution for the program in Figure 3.1 appears in Table A.3 of Appendix A.2 and the must alias solution is given in Table A.4. The program in Figure 3.1 illustrates why using *must-hold* information is more useful than just using must alias information. In particular, n_8 (`*f = 1`) redefines a or b depending on the call site. Without *must-hold* information we cannot take advantage of this, as there are no fixed-location must aliases of $*f$. If we use *must-hold*, then we capture the fact that n_8 is a kill of definitions of a when called from c_1, and is a kill of definitions of b when called from c_2.

3.3 Reaching Definitions Algorithm for Single Level Pointers

We begin with the algorithm to calculate interprocedural reaching definitions in the absence of pointers, effectively in the absence of mechanisms that cause aliasing, as C only has pass-by-value parameters. In a sense, this is equivalent to *Fortran* without aliasing due to pass-by-reference parameters. A high level overview of our full def-use associations algorithm is shown in Figure 3.4. We first explain the algorithm for programs without aliasing; this involves the starred steps in Figure 3.4. Then, we show

1.* construct the ICFG of the program.

2. compute the conditional may alias solution (*may-hold*).

3. compute the may alias solution (*may alias*) from the conditional may alias solution.

4. compute the conditional must alias solution (*must-hold*).

5. compute the must alias solution (*must alias*) from the conditional must alias solution.

6.* compute a conditional reaching definitions solution (*reaches*) using the conditional may alias, conditional must alias, and must alias solutions, when necessary. (Note: this does not use may alias.)

7.* compute the interprocedural reaching definitions solution (*rdtop*) using conditional reaching definitions.

8.* compute the def-use solution using interprocedural reaching definitions and, when necessary, may alias.

Figure 3.4: Interprocedural def-use associations algorithm

how to adapt the algorithm to use *may-hold* information for single level pointers; this is the version of the algorithm in our prototype implementation.

The conditional analysis approach obviates the need for any special treatment to handle recursive procedures. When conditional analysis assumes a condition at the entry of a procedure, it is regardless of whether the condition holds due to a recursive or non-recursive call, and for a procedure the same condition is never processed twice. Finally, we describe the algorithm to compute def-use associations given the reaching definitions for each ICFG node; we also indicate possible precision improvements through the use of must alias information.

3.3.1 Interprocedural Reaching Definitions without Aliasing

In the case of no aliasing, we use an assumed reaching definition *assumed-rd* as an abstraction of the call stack. We solve the *conditional reaching definitions* problem (*reaches*): *Given that m, n are ICFG nodes, if the definition* assumed-rd *of a fixed-location reaches the entry of the procedure containing node n_i, can definition $< n_k : a >$*

```
    int a, b;
e1: P () {
n1:    a = 1;
c1:    R ();
r1:
x1: }
```

```
e2: Q () {
n2:    a = 2;
c2:    R ();
r2:
x2: }
```

```
e3: R () {
n3:    b = 3;
n4:    printf("%d",b);
x3: }
```

Figure 3.5: Conditional reaching definitions without aliasing

reach node n_i? This section describes how to solve for conditional reaching definitions. Then we show how to use conditional reaching definitions to obtain interprocedural reaching definitions sets (*rdtop*) at the top of every ICFG node. This use of conditional information enables us to restrict our analysis to realizable paths in the ICFG.

Conditional reaching definitions without aliasing

We define the *reaches* predicate to represent conditional reaching definitions information at the bottom of each ICFG node. *reaches*(n_i, *ARD*, rd) is *true* iff assuming the assumed-rd *ARD* reaches the entry of the procedure containing n_i, rd reaches the bottom of n_i. There are two types of rd to consider. First, rd can be generated during the execution of the procedure containing n_i and subsequently reach n_i. In this case, no assumptions are necessary at the entry or equivalently the reaching definition is valid regardless of the calling context; we use the *ARD* of $\emptyset$ to represent this case. For example in Figure 3.5, *reaches*(n_4, $\emptyset$, $< n_3 : b >$) $= true$, since the definition $< n_3 : b >$ reaches node n_4 regardless of calling context. Second, an rd may reach the entry of the procedure and be preserved on some realizable path from entry to n_i, in which case $ARD = rd$ and *ARD* abstracts the call stack. For example in Figure 3.5, *reaches*(n_4, $< n_1 : a >$, $< n_1 : a >$) $= true$, since the definition $< n_1 : a >$ reaches node n_4 contingent on the assumption that $< n_1 : a >$ reaches e_3[14]. From these two cases, it is easy to see that the value of *ARD* is either $\emptyset$ or rd itself.

[14]Note that *reaches*(n_4, $\emptyset$, $< n_1 : a >$) $= false$.

Calculation at each ICFG node

Below we specify the *reaches* equations for each ICFG node n_i. Return nodes offer the most interesting case in the interprocedural analysis.

Assignment node n_i: "a = ..." has three effects on reaching definitions; it (1) creates the definition $< n_i : a >$ regardless of calling context, (2) kills (i.e. does not preserve) all incoming definitions of a, and (3) preserves definitions of everything but a. This naturally implies the equations:

1. $reaches(n_i, \emptyset, < n_i : a >) = true$[15]
2. for all $n_k \neq n_i$, $ARD = \emptyset$ or $< n_k : a >$:

   $reaches(n_i, ARD, < n_k : a >) = false$
3. for all fixed-locations $b \neq a$, all nodes n_k, $ARD = \emptyset$ or $< n_k : b >$:

   $reaches(n_i, ARD, < n_k : b >) =$

$$\bigvee_{n_p \text{ pred of } n_i} (reaches(n_p, ARD, < n_k : b >))$$

Entry node n_i: The start of a procedure has three effects; (1) formal parameters are defined, (2) definitions of globals are preserved across the call and reach the entry of the called procedure, (3) definitions of locals of all procedures (including this one) are uninteresting as they can not be redefined nor used and thus can be ignored. This leads to the following equations:

1. for each formal f:

   $reaches(n_i, \emptyset, < n_i : f >) = true$
2. for all globals b, all nodes n_k:

   $reaches(n_i, < n_k : b >, < n_k : b >) =$

$$\bigvee_{\substack{c_p \text{ calls } n_i \text{ and} \\ ARD = \emptyset \text{ or } < n_k : b >}} (reaches(c_p, ARD, < n_k : b >))$$

[15] For completeness, we need to define $reaches(n_i, < n_i : a >, < n_i : a >)$. Its value is unimportant, but we assume *true*.

3. all other $reaches(n_i,\ ARD,\ < n_k : a >)$ not covered by (1) or (2) are $false$.

Rule 2 indicates that the call stack abstraction of ARD at c_p maps to all $< n_k : b >$ such that $reaches(c_p,\ ARD,\ < n_k : b >)$ is $true$ and b is visible at the call.

Return node n_i: Let c_i be the call node for the site, n_i is the corresponding return node and x_c be the exit node of the procedure called by c_i. There are three ways a reaching definition $< n_k : b >$ can reach the return node of a call site: (i) b is not visible in the called procedure (because of the absence of aliasing this definition must be preserved through the procedure) and $< n_k : b >$ reaches c_i, (ii) $< n_k : b >$ reaches x_c regardless of the calling context (i.e., $reaches(x_c,\ \emptyset,\ < n_k : b >)$ is $true$) and b is visible in the called procedure, (iii) $< n_k : b >$ reaches x_c assuming it reaches e_c (entry of procedure containing x_c), b is visible in the called procedure, **and** $< n_k : b >$ reaches c_i under some abstraction of the call stack.

For Figure 3.5, (ii) allows us to conclude that $< n_3 : b >$ reaches r_1 and r_2, while (iii) allows us to conclude that $< n_2 : a >$ reaches r_2 but not r_1. If there were a local variable `c` in `Q`, such that a definition of `c` appeared at $n_{2'}$ before node c_2, then the definition $< n_{2'} : c >$ would be preserved through the call to procedure `R` and would reach r_2, as in case (i).

1. for all b not visible in the called procedure, all nodes n_k, $ARD = \emptyset$ or $< n_k : b >$:

$$reaches(n_i, ARD, < n_k : b >) = reaches(c_i, ARD, < n_k : b >)$$

2. for all b visible in the called procedure, all nodes n_k, $ARD = \emptyset$ or $< n_k : b >$:
$reaches(n_i,\ ARD,\ < n_k : b >) =$

$$\left(\begin{array}{l} \quad reaches(x_c,\ \emptyset,\ < n_k : b >) \lor \\ \left(\begin{array}{l} reaches(x_c,\ < n_k : b >,\ < n_k : b >) \\ \land\ reaches(c_i,\ ARD,\ < n_k : b >) \end{array} \right) \end{array} \right) \begin{array}{l} \text{:from ii} \\ \text{:from iii} \\ \end{array}$$

Any other node n_i: Because *reaches* at a node includes the effects of that node, other nodes (e.g., *call*, *exit*, *conditional*, ...) simply collect the reaching definitions information. Thus:

$reaches(n_i,\ ARD,\ < n_k : b >) =$

$$\bigvee_{n_p \text{ pred of } n_i} (reaches(n_p,\ ARD,\ < n_k : b >))$$

Thus, the conceptual (although inefficient) algorithm to calculate *reaches* comprises steps 1 and 6 in Figure 3.4 restated here:

1. Construct the ICFG.

2. Compute *reaches*.

   - Initialize $reaches(n_i,\ ARD,\ rd)$ to *false* for all nodes n_i, reaching definitions rd, and $ARD = \emptyset$ or rd.

   - Using the above equations, calculate the fixed point of *reaches* using a standard data flow analysis algorithm.

The *reaches* solution for the program in Figure 3.5 can be found in Table 3.1.

Theorem 3.3.1 *The computation of the* reaches *predicate is a polynomial-time (in the number of ICFG nodes) fixed point calculation.*

Let N denote the set of ICFG nodes and v the number of fixed-locations that get defined in the program. There are $\mathcal{O}(|N|^2 * v)$ *reaches* predicates because a definition is a node/fixed-location pair ($\mathcal{O}(|N| * v)$) and *reaches* is a node/definition pair with a bit to indicate if the assumed-rd is $\emptyset$ or the definition. Since each definition of a fixed-location is represented by an ICFG node, it is clear that $v \leq |N|$. The calculation of each *reaches* takes time $\mathcal{O}(predecessors)$ of the node which is at most $\mathcal{O}(|N|)$. In the fixed point calculation the value of a *reaches* changes from *false* to *true* at most once. Thus the computation of all *reaches* predicates is a polynomial-time calculation. □

Procedure P		
node	*ARD*	*rd*
n_1,c_1	$\emptyset$	$< n_1 : a >$
r_1,x_1	$\emptyset$	$< n_1 : a >$
	$\emptyset$	$< n_3 : b >$

Procedure Q		
node	*ARD*	rd
n_2,c_2	$\emptyset$	$< n_2 : a >$
r_2,x_2	$\emptyset$	$< n_2 : a >$
	$\emptyset$	$< n_3 : b >$

Procedure R		
node	*ARD*	*rd*
e_3	$< n_1 : a >$	$< n_1 : a >$
	$< n_2 : a >$	$< n_2 : a >$
n_3,n_4,x_3	$< n_1 : a >$	$< n_1 : a >$
	$< n_2 : a >$	$< n_2 : a >$
	$\emptyset$	$< n_3 : b >$

This table summarizes the *reaches*(*node*, *ARD*, *rd*) which are *true* for the program in Figure 3.5.

Table 3.1: Summary of *reaches*

Reaching definitions from conditional reaching definitions

We now formulate a data flow problem on the ICFG to compute (non-conditional) reaching definition solution *rdtop* from *reaches*. *reaches* is defined so that the solution associated with a program point (node) includes the effects of executing that program point; we call this an *on-bottom* data flow solution. However, *rdtop* is an *on-top* data flow problem which means the solution does not include the effects of the associated program point. Thus, the equations for *rdtop* must capture this shift from an on-bottom to an on-top solution. This is done by combining *reaches* solutions from predecessor nodes to form an *rdtop* solution at their shared successor node.

Clearly, *rd* is in the reaching solution at n_i (ignoring the top/bottom issue for now) if it conditionally reaches n_i (i.e. *reaches*(n_i, *ARD*, *rd*)) and *ARD* occurs on some path to the entry of the procedure. However, given our equation for *reaches* at entry (on p. 26), *reaches*(e_k, *ARD*, *ARD*) will be true only if there exists an execution on which *ARD* reaches the entry node e_k.

Therefore, for each node n_i in the ICFG, *rdtop* is calculated as follows:

- if n_i is an entry node or return node (these two are special cases as they do not have intraprocedural predecessors).

$$rdtop(n_i) = \left\{ < n_k : b > \;\middle|\; \begin{array}{l} (\exists\ ARD) \text{ such that} \\ reaches(n_i,\ ARD,\ < n_k : b >) \\ \text{is } true \end{array} \right\}$$

- otherwise

$$rdtop(n_i) = \bigcup_{n_p \text{ pred of } n_i} \left\{ < n_k : b > \;\middle|\; \begin{array}{l} (\exists\ ARD) \text{ such that} \\ reaches(n_p,\ ARD,\ < n_k : b >) \\ \text{is } true \end{array} \right\}$$

Theorem 3.3.2 *The computation of* rdtop *given* reaches *is a polynomial-time fixed point calculation.*

$reaches(n_k,\ ARD,\ rd)$ will be accessed exactly once for each intraprocedural successor of n_k unless n_k is a entry or return node. No node has more than two intraprocedural successors[16]. If n_k is an entry or return node, $reaches(n_k,\ ARD,\ rd)$ is accessed for the *rdtop* equation of unique successor of n_k and for $rdtop(n_k)$. Thus the cost of computing *rdtop* is $\mathcal{O}$(size of *reaches* solution) which is clearly polynomial-time (in the size of the ICFG). □

Precision of calculation

Intraprocedural reaching definitions without aliasing is a well-understood problem and precise solution procedures are available in literature. The intraprocedural aspects of our reaching definitions calculation are essentially equivalent to them in obtaining a precise solution. To solve the interprocedural reaching definitions problem precisely, a solution procedure must propagate the intraprocedural reaching definitions only over realizable paths. Using conditional analysis, our algorithm described above restricts propagation to exactly such paths. If a definition *rd* of a local variable reaches a call site, it is preserved through the called function and propagated directly to the corresponding

[16] Switch statements are translated into nested if-then-else statements.

```
      R(f)
      int *f;
e1:   {
        int x;
n1:     x = 1;
n2:     printf("%d",x);
n3:     printf("%d",*f);
n4:     x = 2;
n5:     if (*f == 2)
c1:       R(&x);
r1:
      }
```

Figure 3.6: Example for non-visible object names

return node, as the called function cannot address the variable (case (i) p. 27). If a definition of a global variable is created in a called function and reaches the *exit* of that function, we propagate it to *return* nodes of all call sites invoking the function, as the creation and intraprocedural propagation is completely independent of calling context (see case (ii) on p. 27). Lastly, if a definition *rd* reaches the *entry* of a function, and is preserved through the function to reach its *exit*, it should propagate to only those *return* nodes corresponding to call sites which propagated *rd* to *entry* (case (iii) on p. 27). These simple constraints on interprocedural propagation are satisfied by our algorithm to yield a precise solution of the problem in the absence of aliasing. In terms of the solution calculated, this algorithm is identical to the interprocedural reaching definitions algorithm by Harrold and Soffa [HS90].

3.3.2 Interprocedural Reaching Definitions with Aliasing

In this section, we extend the framework described in Section 3.3.1 to include single level pointers in the analysis. The introduction of pointers results in aliases at various program points. Our analysis must account for the generation of reaching definitions due to aliasing effects. Our reaching definitions algorithm for this problem is polynomial but imprecise, as any polynomial algorithm must be, assuming $P \neq NP$ (see Section 3.1.2).

Before we can give the equations we have to address the issue of non-visible fixed-locations, which intuitively correspond to the local variables of calling procedure propagating into a called procedure. This is especially important in recursive procedures. Consider the program segment in Figure 3.6: we want to ensure that $\ll < n_4 : x >, n_3 \gg$ is reported and avoid reporting $\ll < n_1 : x >, n_3 \gg$. This is not possible if we use x to represent both itself and the "x" from earlier instantiations of `R` on the runtime stack. Thus we let x represent the local variable of the procedure which instantiates it, but refer to it as nv_x when it propagates into a called procedure. Note that in case of a recursive call, nv_x represents the x from an earlier instantiation of the called procedure.

Modeling parameter bindings

We need to model the aliasing effects of pointer parameter bindings for each call site. For this purpose, we use the function back-bind$_{c_P}$ for each call site c_P. $\textit{back-bind}_{c_P}(RA)$ specifies which alias on a path $\rho \ldots c_P$ guarantees that RA exists on the path $\rho \ldots c_P e_P$ where e_P is the entry node of the procedure being invoked. $\textit{back-bind}_{c_P}$ is similar to b_e of [CK88], but instead of mapping a formal parameter to its corresponding actual argument in a call, we are mapping an alias pair in the called procedure to the inducing alias pair in the calling procedure. Assume that `f1`, `f2` are pointer formals of procedure `P`, `q` is a global pointer, c_P is "`P(q,&r)`", and alias $<*q, nv\_l>$ holds at c_P. Then we have:

$$
\begin{aligned}
\textit{back-bind}_{c_P}(\emptyset) = & \quad \emptyset \\
\textit{back-bind}_{c_P}(<*f2, r>) = & \quad \emptyset \\
\textit{back-bind}_{c_P}(<*f1, nv\_l>) = & \quad <*q, nv\_l> \\
\textit{back-bind}_{c_P}(<*q, nv\_l>) = & \quad <*q, nv\_l>
\end{aligned}
$$

We also need the inverse of $\textit{back-bind}_{c_P}$ operation; we use $\textit{bind}_{c_P}$ to model the affects of parameter bindings on aliases[17]. Intuitively, $\textit{bind}_{c_P}(\emptyset)$ is the set of all the aliases on entry of a called procedure that must exist because of parameter bindings,

[17] A detailed discussion of $\textit{back-bind}_{c_P}$ and $\textit{bind}_{c_P}$ appears in [Lan92a].

while $bind_{c_P}(<a, b>)$ is the set of aliases at entry of a called procedure whose existence is implied by a being aliased to b at c_P. For the following example program segment, assume that the aliases $<*g, l>$ and $<*g, nv\_x>$ hold at c_P where g is a global, l is local to procedure R and x is local to the caller of procedure R.

```
      R( )
      {                  P(f)
      ...                int *f;
                         {
c_P:  P(g);              ...
r_P:                     }
      }
```

$$\begin{aligned}
&bind_{c_P}(\emptyset) = \Big\{<*f, *g>\Big\}, \\
&bind_{c_P}(<*g, l>) = \Big\{<*g, nv_l>, <*f, nv_l>\Big\}, and \\
&bind_{c_P}(<*g, nv_x>) = \Big\{<*g, nv_x>, <*f, nv_x>\Big\}
\end{aligned}$$

Conditional reaching definitions with aliasing

The *reaches* predicate for this version of the algorithm has the following interpretation: *reaches*(n_i, (ARD, RA), rd) is *true* if, given that ARD and RA reach the entry of the procedure containing n_i, rd reaches the bottom of n_i. Now the pair (ARD,RA) is our abstraction of the call stack. *reaches* has the nice property that ARD and RA are never both non-$\emptyset$. Intuitively, it is easy to see three possible situations at procedure exit:

1. If ARD is $\emptyset$ and the corresponding reaching definition was created as a result of the call and did not need an alias for its creation, then we have ($\emptyset$,$\emptyset$) as our reaching definition, reaching alias information.

2. If ARD is non-$\emptyset$, then the corresponding reaching definition was passed in from a call and the ARD acts as an abstraction of the call stack. A may alias cannot be used to kill a definition unless it is also a must alias; thus, there is no need for retaining an RA with this definition, because our RA is not an explicit must alias.

3. If *ARD* is $\emptyset$, but *RA* is non-$\emptyset$, then the corresponding definition was created during the call through the alias information. Here *RA* acts as a call stack abstraction and is necessary for the definition through the alias to be identified. In Figure 3.1 (p. 16), $< n_8 : a >$ is created at node n_8 if the alias $<*f, a>$ reaches the entry node of function `R`. Here, ($\emptyset$,$<*f, a>$) abstracts the call stack.

This property limits the number of *reaches* relations by eliminating a multiplicative effect which would arise if relations with both non-$\emptyset$ components occurred in the analysis. To shorten our equations in the remainder of this section, let the notation "(ARD or $RA = \emptyset$)" represent the condition "for $ARD = \emptyset$ or $< n_k : a >$ (n_k and a are determined by context, that is, by the last component of the *reaches*), and for all *RA* such that at least one of *ARD* and *RA* is $\emptyset$".

Calculation at each ICFG node

As in Section 3.3.1, we now present the *reaches* equations differentiated by node type, for programs with single level pointers. Recall from Figure 3.4 that at this point in our algorithm we already have computed *may-hold* information.

Assignment node n_i: There are two classes of assignments; one is assignments to fixed-locations and the other is an assignment through a dereferenced pointer. In either case, the assignment: (i) creates definitions of aliases of the left-hand-side (Reflexive aliases exist everywhere with reaching alias $\emptyset$.), (ii) kills incoming definitions of the left-hand-side (We do not use must alias information in this version of the algorithm.), (iii) preserves definitions of everything other than the left-hand-side.

1. Consider the fixed-location assignment "`a = ...`" (similar to the corresponding case in Section 3.3.1):

   (a) *reaches*(n_i, $(\emptyset, \emptyset)$, $< n_i : a >$) $=$ *true* (It is not possible for two fixed-locations to be aliased; therefore, we do not need to consider aliases of a.)

(b) for all $n_k \neq n_i$, $(ARD$ or $RA = \emptyset)$:

$reaches(n_i,\ (ARD, RA),\ < n_k : a >) = false$

(c) for all fixed-locations $b \neq a$, all nodes n_k, $(ARD$ or $RA = \emptyset)$:

$reaches(n_i,\ (ARD, RA),\ < n_k : b >) =$

$$\bigvee_{n_p \text{ pred of } n_i} (reaches(n_p,\ (ARD, RA),\ < n_k : b >))$$

2. Consider the assignment "`*p = ...`": For all nodes n_k, $(ARD$ or $RA = \emptyset)$, and all fixed-locations b (the first part is from (i) and the second part is from (iii)):

$reaches(n_i,\ (ARD, RA),\ < n_k : b >) =$

$$\left(\begin{array}{l} n_k = n_i \wedge ARD = \emptyset \wedge \\ \displaystyle\bigvee_{n_p \text{ pred of } n_i} (may\text{-}hold(n_p,\ RA,\ <*p, b>)) \end{array} \right)$$

$$\bigvee$$

$$\left(\bigvee_{n_p \text{ pred of } n_i} (reaches(n_p,\ (ARD, RA),\ < n_k : b >)) \right)$$

Entry node n_i: *reaches* at entry nodes is exactly the same as for the case when there are no aliases in the program (see Section 3.3.1) except that there is an associated reaching alias which will always be $\emptyset$. The equations are:

1. for each formal `f`:

   $reaches(n_i,\ (\emptyset, \emptyset),\ < n_i : f >) = true$

2. for all globals `g`, all nodes n_k:

   $reaches(n_i,\ (< n_k : g >, \emptyset),\ < n_k : g >) =$

$$\bigvee_{\substack{c_p \text{ calls } n_i \text{ and} \\ (ARD \text{ or } RA = \emptyset)}} (reaches(c_p,\ (ARD, RA),\ < n_k : g >))$$

3. for all locals `l`, all nodes n_k, and all call sites c_p of n_i (the equation is similar to (2); however, it is complicated by name space mapping):

   $reaches(n_i,\ (< n_k : nv\_l >, \emptyset),\ < n_k : nv\_l >) =$

   $$\bigvee_{\substack{\alpha \in \{l, nv\_l\} \text{ and} \\ (ARD \text{ or } RA = \emptyset)}} (reaches(c_p, (ARD, RA), < n_k : \alpha >))$$

   This accounts for the situations where (i) l is local to the procedure containing c_p or (ii) l is not visible in the procedure containing c_p and appears as nv_l at c_p itself.

4. all other predicates, (ARD or $RA = \emptyset$), $reaches(n_i,\ (ARD, RA),\ < n_k : a >)$ not covered by (1) or (2) are $false$.

Return node n_i: In the following, let the return node be n_i, the corresponding call node be c_i, and the exit node of the called procedure be x_k. (ARD or $RA = \emptyset$), $reaches(n_i,\ (ARD, RA),\ rd)$ is $true$ iff any of the following situations exists:

1. rd was generated during the execution of the called procedure without any assumptions at entry and reached the exit. For example, in Table A.5 of Appendix A.2:

   $$reaches(x_2,\ (\emptyset, \emptyset),\ < n_9 : c >)$$

   $$\Rightarrow reaches(r_1,\ (\emptyset, \emptyset),\ < n_9 : c >)$$

2. rd reached the call site (and thus the entry of the called procedure) and is preserved through the called procedure to reach its exit. For example in Figure 3.1, the definition $< n_4 : p >$ reaches the call node c_1 and is preserved through `R` to reach x_2. As a result in Table A.5:

   $$\left(\begin{array}{l} reaches(c_1,\ (\emptyset, \emptyset),\ < n_4 : p >) \wedge \\ reaches(x_2,\ (< n_4 : p >, \emptyset),\ < n_4 : p >) \end{array} \right)$$

   $$\Rightarrow reaches(r_1,\ (\emptyset, \emptyset),\ < n_4 : p >)$$

3. rd is created because of an alias which exists conditional on some RA at the entry of the called procedure and that rd reaches the exit of that procedure (for example, *reaches*(x_2, $(\emptyset, <*f, a>)$, $< n_8 : a >$) in Table A.5). Also the call site must induce RA in the called procedure (e.g., c_1 induces $<*f, a>$ in R). Thus,

$$\begin{pmatrix} \textit{reaches}(x_2,\ (\emptyset, <*f, a>),\ < n_8 : a >) \wedge \\ \textit{back-bind}_{c_1}(<*f, a>) = <*\text{p,a}> \wedge \\ \textit{may-hold}(c_1,\ \emptyset,\ <*p, a>) \end{pmatrix}$$

$$\Rightarrow \textit{reaches}(r_1,\ (\emptyset, \emptyset),\ < n_8 : a >)$$

The following formula accounts for the three situations described above. A fourth situation where a definition reaches the entry of a procedure and is preserved because of the *absence* of a must alias is safely covered by (2) but is handled better when *reaches* is calculated using *must-hold* information (see Section 3.3.3).

For c_i the call node of the call site of n_i, x_k the exit of the called procedure, (ARD or $RA = \emptyset$), b a global or nv_l for some l which is not visible in the procedure containing c_i:

$\textit{reaches}(n_i,\ (ARD, RA),\ < n_m : b >) =$

$$\begin{array}{ll} 1. & \textit{reaches}(x_k, (\emptyset, \emptyset), < n_m : b >) \vee \\ 2. & \textit{reaches}(c_i, (ARD, RA), < n_m : b >) \wedge \\ & \textit{reaches}(x_k, (< n_m : b >, \emptyset), < n_m : b >) \\ 3. & \displaystyle\bigvee_{\substack{\text{all reaching} \\ \text{aliases } RA'}} \begin{pmatrix} \textit{reaches}(x_k, (\emptyset, RA'), < n_m : b >) \\ \wedge\, \textit{may-hold}(c_i, RA, \textit{back-bind}_{c_i}(RA'))^{18} \end{pmatrix} \end{array}$$

For c_i the call node of the call site of n_i, x_k the exit of the called procedure, (ARD or $RA = \emptyset$), l a local to the procedure containing c_i:

[18] For simplicity, assume *may-hold*(n_m, RA_m, $\emptyset$) is *true* for all n_m and RA_m.

$reaches(n_i,\ (ARD, RA, RMA),\ < n_m : l >) =$

1. /* there is no corresponding case 1 */
2. $reaches(c_i, (ARD, RA), < n_m : l >) \wedge$
   $reaches(x_k, (< n_m : nv\_l >, \emptyset), < n_m : nv\_l >)$
3. $$\bigvee_{\substack{\text{all reaching}\\ \text{aliases } RA'}} \begin{pmatrix} reaches(x_k, (\emptyset, RA'), < n_m : nv\_l >) \\ \wedge may\text{-}hold(c_i, RA, back\text{-}bind'_{c_i}(RA')) \end{pmatrix}$$

Note: $back\text{-}bind'_{c_i}(<a,b>)$ is identical to $back\text{-}bind_{c_i}(<a,b>)$ except whenever $back\text{-}bind_{c_i}(<a,b>) = <*g, nv\_l>$ (any variable g, local l), $<*g, l>$ is returned instead.

Any other node n_i: Because *reaches* at a node includes the effects of that node, other nodes (e.g., *call*, *exit*, *conditional*, ...) simply collect the reaching definitions information. Thus:

$$reaches(n_i,\ (ARD, RA),\ < n_k : b >) = \bigvee_{n_p \text{ pred of } n_i} (reaches(n_p,\ (ARD, RA),\ < n_k : b >))$$

Theorem 3.3.3 *The algorithm to calculate* reaches *is polynomial in the number of ICFG nodes and object names.*

Let N denote the set of ICFG nodes and v the number of object names. Aliases are pairs of object names; thus there are $\mathcal{O}(v^2)$ of them. Definitions are node / fixed-location pairs, so there are $\mathcal{O}(|N| * v)$ of them. *reaches* is a quadruple $reaches(n_i,\ (ARD, RA),\ rd)$ with the restriction that ARD is either $\emptyset$ or rd; thus there are $\mathcal{O}(|N|^2 * v^3)$ *reaches* predicates. For each *reaches* calculation at a return node we may do $\mathcal{O}(v^2)$ work because there may be as many as v^2 RA values (worst case). The work at an arbitrary ICFG node is bounded by the work at a return node. Therefore the total cost of processing for all ICFG nodes, is bounded above by $\mathcal{O}(|N|^2 * v^5)$ since, in the fixed point calculation, the value of a *reaches* changes from $false$ to $true$ at most once. Thus, the computation of *reaches* predicate is a polynomial-time calculation. □

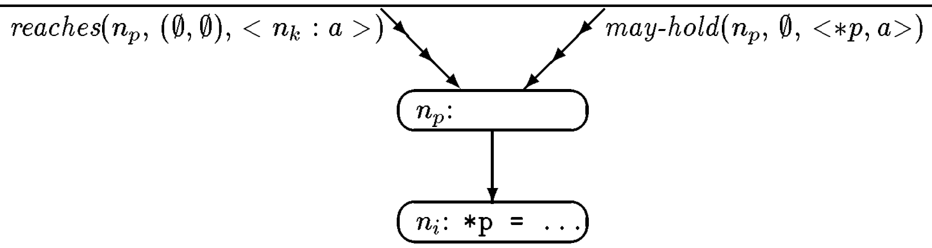


Figure 3.7: Source of approximation

Reaching definitions from conditional reaching definitions

The *rdtop* calculation with aliasing is conceptually identical to the *rdtop* calculation without aliasing (see Section 3.3.1). The *rdtop* solution for the program in Figure 3.1 appears in Tables A.8 and A.9.

For each node n_i in the ICFG, *rdtop* is calculated as follows:

- if n_i is an entry node or return node (special case these two as they do not have intraprocedural predecessors).

$$rdtop(n_i) = \left\{ < n_k : b > \;\middle|\; \begin{array}{l} (\exists\ (ARD,RA)) \text{ such that} \\ reaches(n_i,\ (ARD, RA),\ < n_k : b >) \\ \text{is } true \end{array} \right\}$$

- otherwise

$$rdtop(n_i) = \bigcup_{n_p \text{ pred of } n_i} \left\{ < n_k : b > \;\middle|\; \begin{array}{l} (\exists\ (ARD,RA)) \text{ such that} \\ reaches(n_p,(ARD,RA), \\ < n_k : b >) \text{ is } true \end{array} \right\}$$

Precision of calculation

Beyond the normal imprecision due to nonexecutable intraprocedural paths, there is another source of approximation in our reaching definitions algorithm, which is illustrated by the following scenario. We illustrate the example for *reaches* without must alias information, but must alias information is not enough to avoid this problem in all cases. As shown in Figure 3.7, suppose:

```
         if (i == 0)
            p = &a;
         else
nk:         a = 1;
np:      i = 0;
ni:      *p = 2;
```

reaches(n_i, $(\emptyset, \emptyset)$, $< n_k : a >$)
is precise

Case 1

```
         p = &a;
         if (i == 0)
            p = &b;
         else
nk:         a = 1;
np:      i = 0;
ni:      *p = 2;
```

reaches(n_i, $(\emptyset, \emptyset)$, $< n_k : a >$)
is imprecise (but safe)

Case 2

Figure 3.8: Two possibilities

- n_i is the assignment "`*p = ...`"
- n_p is an immediate predecessor of n_i in the ICFG.
- we know that the definition $< n_k : a >$ reaches the bottom of n_p[19] on some path (for example, the predicate *reaches*(n_p, $(\emptyset, \emptyset)$, $< n_k : a >$) is *true*).
- we also know that $*p$ is aliased to a on some path to the bottom of n_p (for example, the predicate *may-hold*(n_p, $\emptyset$, $<*p, a>$) is *true*).

Our algorithm as described would conclude that the predicate *reaches*(n_i, $(\emptyset, \emptyset)$, $< n_k : a >$) is *true*, but is that precise? If definition $< n_k : a >$ reaches n_p on some path and $<*p, a>$ does not hold on that path then definition $< n_k : a >$ reaches the bottom of n_i (see Case 1 of Figure 3.8). However, if for every path on which definition $< n_k : a >$ reaches n_p, $*p$ is aliased to a, then $< n_k : a >$ does not reach the bottom of n_i and our algorithm is being imprecise (see Case 2 of Figure 3.8).

3.3.3 Using Must Alias for More Precision

To increase the accuracy of the reaching definition calculation, we can use the *must-hold* function and the must alias solution in the *reaches* calculation to kill reaching

[19] and thus the top of n_i

definitions when an assignment is made through a must-aliased object name[20]. The new *reaches* has the form

$$reaches(n_i,\ (ARD, RA, RMA),\ rd)$$

and is *true* iff rd reaches n_i, with the assumed-rd ARD, the reaching alias RA, and the reaching must alias set RMA. For every call site c_i, we define one reaching must alias set (RMA_{c_i}), as the set of must aliases which are propagated to the called procedure from that call. For non-globals, this propagation involves parameter bindings and name space mappings (see Appendix A.1). The number of distinct reaching must alias sets equals the number of calls to the procedure.

Although we are doing a refined conditional reaching definitions calculation, it is still true that during the calculation, a *reaches* can only change its value from $false$ to $true$, thus assuring *monotonicity*. This becomes obvious when one realizes that to *kill* a reaching definition simply means *not propagating* an rd at a node to its successor(s). A complete and formal description of this final version of the algorithm appears in Appendix A.1, but it is similar enough to the algorithm in Section 3.3.2 that we limit our discussion here to an example of killing definitions at an assignment.

In Figure 3.1, $\{<*p, a>\}$ is the set of aliases associated with c_1 in must alias solution (see Table A.4). Thus, $\{< *p, a >, <*p, *f>, <*f, a>\}$ is a reaching must alias set of `R`. Since $< n_1 : a >$ reaches c_1, $reaches(e_2,\ (< n_1 : a >, \emptyset, \{<*p, a>, <*p, *f>, <*f, a>\}),\ < n_1 : a >)$ is $true$ (see Table A.7 for the complete *reaches* solution). The successor of e_2 is n_8 (`*f = 1`). Clearly this assignment kills off definitions of a if $*f$ is must aliased to a. In this case it is, and thus $reaches(n_8,\ (< n_1 : a >, \emptyset, \{<*p, a>, <*p, *f>, <*f, a>\}),\ < n_1 : a >)$ remains $false$. The formal check to see if "`*q = ...`" at n_i kills a $reaches(n_p,\ (ARD, RA, RMA),\ < n_k : x >)$, where n_p is a predecessor of n_i, is:

$$must\text{-}hold(n_p,\ <*q, x>) \subseteq RMA$$

[20]The analysis in this section is not included in our prototype implementation. We do not use *must-hold* information as there are still difficulties that must be surmounted before *must-hold* can be implemented practically.

Now we consider c_2 of Figure 3.1. $\{<*\text{p,b}>\}$ is the set of all aliases associated with c_2 in the must alias solution (see Table A.4). Thus, $\{<*p, b>, <*p, *f>, <*f, b>\}$ is a reaching must alias set of R. Since $< n_1 : a >$ reaches c_2,

$$reaches(e_2, (< n_1 : a >, \emptyset, \{<*p, b>, <*p, *f>, <*f, b>\}), < n_1 : a >)$$

is *true* (see Table A.7). Again consider the assignment at n_8, since

$$must\text{-}hold(e_2, <*p, a>) = \{<*p, a>\} \not\subseteq \{<*p, b>, <*p, *f>, <*f, b>\}$$

we set $reaches(n_8,\ (< n_1 : a >, \emptyset, \{<*p, b>, <*p, *f>, <*f, b>\}),\ < n_1 : a >)$ to *true* implying that $reaches(x_2,\ (< n_1 : a >, \emptyset, \{<*p, b>, <*p, *f>, <*f, b>\}),\ < n_1 : a >)$ is *true* also. *reaches* at return nodes are handled as in Section 3.3.2 with the additional restriction that the call node of the call site must induce the reaching must alias. Thus the above *reaches* causes $reaches(r_2,\ (\emptyset, \emptyset, \emptyset),\ < n_1 : a >)$ to be *true* but leaves $reaches(r_1,\ (\emptyset, \emptyset, \emptyset),\ < n_1 : a >)$ *false*. A comparison of the corresponding entries for r_1 in Tables A.5 and A.6 reveals that the absence of *must-hold* information leads to a spurious $< n_1 : a >$ reaching r_1. Similarly, in the absence of *must-hold* information, a spurious $< n_2 : b >$ reaches r_2 (compare correponding entries in the two tables).

Again, details of how to compute *reaches* using *must-hold* and must alias are in Appendix A.1, and the complete solution for the program in Figure 3.1 is in Tables A.6 and A.7.

Approximating must alias information

As we demonstrated in Section 3.3.3, must alias information is useful in increasing the precision of *reaches*. Unfortunately, little research has been done for computing must alias information in the presence of pointers. Landi [Lan92a] presented a polynomial time algorithm for computing must alias in the presence of single level pointers. As presented, the algorithm is of more theoretical than practical utility; it has never been generalized to handle arbitrary pointers.

However, a relevant and safe alternative to must alias information can be obtained cheaply from the *may-hold* information. Consider the following program segment:

```
n1:  for (i=0;i<2;i++) {
n2:     q = malloc();
n3:     *q = 2;
n4:     if (i == 0)
n5:        p = q;
n6:  }
n7:  *p = 3; /* cannot kill < n3 : new2 > */
```

Figure 3.9: Problem using $<\!*p, new_2\!>$ as a must alias at n_7

```
n1:   v = 2;
n2:   *p = 3;
n3:   printf ("%d", v);
```

and assume the only reported fixed-location may alias for $*p$ immediately prior to the execution of node n_2 is v. Because of safety of the may alias algorithm, this means that whenever n_2 is executed one of three cases must exist: (i) p points to v, (ii) p is uninitialized, or (iii) p is NULL. In case (i), v is redefined and the definition $< n_1 : v >$ is killed. In cases (ii) and (iii), the action of this statement is undefined by the C standard, but for most C compilers, abnormal termination results. Thus, whenever statement n_3 is executed and execution continues, the statement that defined v is always n_2, never n_1. Given our assumptions about abnormal termination in cases (ii) and (iii), we can treat $<\!*p, v\!>$ as a must alias, at the cost of querying the may alias solution and finding all the fixed-location aliases of $*p$ when $*p$ is used/defined.

A few points about this technique are:

- If $*p$ is being used/defined and it has only one *fixed-location* may alias, then it can be treated as a must alias.

- If p points to heap storage, then this fact must be represented by some $<\!*p, new_i\!>$ (Recall we use a unique new_i name for each allocation site i in the program). If the alias $<\!*p, new_i\!>$ is in the may alias solution at n_k, even if it is the only alias involving $*p$, then this technique **cannot** be used as the name new_i possibly represents multiple locations and without further information we cannot safely consider $<\!*p, new_i\!>$ as a must alias (see Figure 3.9).

3.4 Def-Use Associations Algorithm

Given interprocedural reaching definitions information (i.e., *rdtop*) at each ICFG node, it is straightforward to compute the interprocedural def-use associations. If a node n_i uses a fixed-location a either directly or indirectly through a pointer (thus requiring may alias information), and there is a definition $< n_k : a >$ reaching the top of node n_i, then we establish a def-use association $\ll < n_k : a >, n_i \gg$. The def-use association solution for the program in Figure 3.1 appears in Table A.12. Notice that $\ll < n_8 : b >, n_7 \gg$ corresponds to a direct use while $\ll < n_8 : a >, n_5 \gg$ corresponds to a use through a pointer. This algorithm is clearly polynomial in the number of nodes in the ICFG.

In addition to the imprecision in our conditional reaching definitions algorithm, there is another source of approximation inherent in our calculation of def-use associations. Suppose n_i uses a indirectly through a may alias. We establish a def-use $\ll < n_k : a >, n_i \gg$. But the realizable path on which $< n_k : a >$ is created at n_k and subsequently reaches n_i, and the realizable path on which the may alias holds at the top of n_i may be distinct. Nevertheless, following the same reasoning as on p. 39, we must assume, for the sake of safety, that the reaching definition as well as the alias occur on the *same* path, and create a possibly spurious def-use association.

3.5 Extensions to Handle Multiple Level Pointers

When we allow multiple levels of pointer dereferencing, the solution of conditional aliasing is no longer precise, but is safe and approximate [LR92]. Our algorithm for conditional reaching definitions inherits this approximation from the aliasing calculation to create possibly spurious reaching definitions at an assignment node. Although the algorithm description in Section 3.3.2 remains unchanged, we need to study implications of this approximation by revisiting the case where our algorithm creates a definintion at an assignment node using aliases of the left-hand-side. On p. 35, where we consider the assignment "`*p = ...`", there may in fact be no realizable path to node n_p on which $<*p, b>$ holds. Nevertheless, the presence of an approximate *may-hold*(n_p, *RA*, $<*p, b>$) will result in *reaches*(n_i, (ARD, RA), $< n_k : b >$). This is clearly approximate, since there is no realizable path to n_i on which the definition of $*p$ redefines the fixed-location b.

Allowing multiple levels of indirection adds another complication to the aliasing part of our algorithm, *i.e.,* the possibility of infinite object names in the presence of recursive structures. The notion of k-limiting, similar to that introduced by Jones and Muchnick [JM79], may be used to limit the length and number of object names. Intuitively, k-limiting implies that up to k explicit dereferences are maintained in an object name and further dereferences are abstracted into a special dereference. Although the aliasing algorithm must account for k-limiting, it has no impact on our reaching definitions and def-use associations algorithms provided that the value of k is so chosen that the names in the source code do not require k-limiting. We are only interested in the reaching definitions of fixed-locations, which are never k-limited (remember, fixed-locations do not have any dereferences). A fixed-location may be indirectly defined or used through an alias of an object name appearing at an ICFG node, which again is not k-limited. We discuss the notion of k-limiting in the context of our C^{++} analysis in Section 5.4.6.

Chapter 4

Present Status and Empirical Results

We have constructed a prototype implementation of our algorithm to observe its performance on C programs. We use the PTT system from Siemens Corporate Research [PWC91] for our C parser. For efficiency, in our implementation we use a worklist-based algorithm. At an assignment node, we initialize the appropriate *reaches* predicates to *true* depending on the *may-hold* predicates at the node. We then propagate the *reaches* values along the realizable paths.

Our original prototype implementation obtained the reaching definitions and def-use associations for C systems with at most a single level of indirection. For aliasing calculation, it used the implementation described in [Lan92a]. Finding C programs that used only single level pointers was extremely difficult, so our test suite is limited to eleven programs. We present our preliminary implementation timing results on a Sun SPARCstation 10 in Table 4.1. The programs come from the following sources: (i) programming assignments for graduate courses at Rutgers University, (ii) C library functions, and (iii) the test suite for TACTIC [Ost90] which used the def-use associations to perform data-flow-based test coverage of C programs. The timings for various analysis phases[21] (*viz.,* conditional may alias, interprocedural reaching definitions, and def-use analysis) are promising, especially since this is a prototype implementation. The time for an unoptimized compile of each program is included for comparison.

To obtain a lower bound on the empirical precision of our reaching definitions solution we use methods similar to those in [LR92]. We designed our implementation to

[21] The timings are measured separately for each phase, and are not cumulative.

Program Name	Lines	Nodes	*may-hold* Calc. Time	RDs Calc. Time	Def-Use Calc. Time	Total Data Flow Time	Compile time
bcmp-etc	50	53	0.02 sec	0.01 sec	0.01 sec	0.04 sec	0.32 sec
pattern	99	183	0.04 sec	0.13 sec	0.06 sec	0.23 sec	0.23 sec
atol-etc	100	160	0.03 sec	0.03 sec	0.01 sec	0.07 sec	0.39 sec
weather	132	134	0.05 sec	0.07 sec	0.02 sec	0.14 sec	0.22 sec
jacobi	172	357	0.09 sec	0.16 sec	0.22 sec	0.47 sec	0.45 sec
SOR	177	352	0.09 sec	0.19 sec	0.19 sec	0.47 sec	0.48 sec
strings	185	415	0.06 sec	0.13 sec	0.04 sec	0.23 sec	0.28 sec
random	366	247	0.46 sec	0.18 sec	0.05 sec	0.69 sec	0.39 sec
crypt	396	395	0.09 sec	0.51 sec	0.18 sec	0.78 sec	0.44 sec
fixoutput	400	616	0.09 sec	0.85 sec	0.11 sec	1.05 sec	0.49 sec
parser	731	1305	0.79 sec	2.92 sec	0.35 sec	4.06 sec	1.09 sec

Table 4.1: Timing results

keep track of these possibly spurious *reaches*[22], appearing in the solution due to the approximation described in Section 3.3.2 (p. 39). From this we compute *%precision* which is the percentage of the solution generated which is *definitely not spurious.* We consider the precision of our solution to be the ratio of the size of the provably precise[23] reaching definitions solution to the size of the entire solution. Thus, *%precision* is a lower bound on the precision of our solution because it assumes that the *reaches* resulting from *every* such safety consideration was spurious.

In Table 4.2, we present measurements indicating the size and quality of our solutions. *RD's Per Node* is the average number of reaching definitions per ICFG node. *RD % precision* is the measure defined above that gives a lower bound on the precision of our calculated solution. The precision of our algorithm appears to be good; at least 90% of the reaching definitions reported by our algorithm are present in the precise solution. *Def-Uses* is the total number of def-use associations in each program. Perhaps most interesting for data-flow-based testing applications are the last two columns

[22]Reporting *reaches* as *true* is counted as possibly spurious if it is the result of some form of the above case, or if it depends on some possibly spurious *reaches*.

[23]assuming all intraprocedural paths are executable

Program Name	Lines	Nodes	RDs Per Node	RD %precision	Def-Uses	DUs Per Def	DUs Per Use
bcmp-etc	50	53	2.7	100.00	24	1.71	1.33
pattern	99	183	13.1	91.91	124	2.43	1.55
atol-etc	100	160	4.9	98.90	105	2.62	1.57
weather	132	134	6.8	96.93	115	2.21	2.17
jacobi	172	357	7.3	98.56	304	3.75	1.68
SOR	177	352	7.6	91.52	266	3.17	1.57
strings	185	415	8.0	100.00	242	2.95	1.34
random	366	247	15.6	90.26	227	2.99	1.94
crypt	396	395	27.2	93.68	468	3.26	2.00
fixoutput	400	616	35.1	100.00	306	5.88	2.15
parser	731	1305	40.7	93.12	867	6.15	2.29

Table 4.2: Analysis results

which measure the number of def-use associations on average at each definition site (i.e., assignment statement) and at each use site in the program, respectively. There is one definition site per assignment statement; in contrast, there is one use site for each value-fetching part of a statement. For example, `*p = a + *q` has one definition site (for `*p`) and four use sites (for `p,a,q,` and `*q`), regardless of the number of aliases for `*p` and `*q`.

Our prototype implementation of the def-use associations algorithm (at Siemens Corporate Research Inc.) is now capable of analyzing programs with multiple level pointers and uses an improved version of aliasing algorithm[24]. We have obtained some data using this extended implementation. We report the results in Table 4.3. In comparison with Tables 4.1 and 4.2, the number of def-use associations per definition and the analysis time appear to have increased. This is expected since the programs in Table 4.3 are bigger in code size and more complex in pointer usage than those analyzed by our original implementation. Also, we conjecture that further imprecision inherent to analysis for multiple level pointers [LR92] is also responsible for the increase in the number of def-use associations. Unfortunately, we do not have access to such analysis

[24] If our original benchmarks are re-analyzed using this prototype, we should see significantly better timing results than those reported in Table 4.1.

Program Name	Nodes	Def-Uses	Def-Uses Per Def	Total Time (sec)
strings1	279	194	3.34	0.34
allroots	411	429	3.67	0.77
diffh	635	794	4.46	2.31
fixoutput1	621	2967	20.05	2.54
lex315	1306	2432	12.60	6.17
calls	1289	16838	12.92	28.17
pokerd	1886	2776	5.83	11.89
clust	1282	28064	17.33	27.98
loader	1567	2398	5.68	12.56
compress	1917	2007	5.35	8.67
diff	3950	7476	8.27	52.31
tbl	6060	52714	33.17	137.65
assembler	3629	13274	14.97	65.82
simulator	5578	36501	34.80	104.73

Table 4.3: Recent implementation results

figures as the number of reaching definitions or the %*precision* measure for reaching definitions solution of this recent implementation.

Chapter 5

Type Determination and Aliasing for C^{++}

This chapter is divided into three sections. Section 5.1 mainly discusses the theoretical issues involving type determination and aliasing. We show that type determination and aliasing are separable when the pointer usage is restricted to a single level of dereferencing, but are closely interdependent when the restriction is lifted. In Section 5.2, we formulate the combined problem and state our tractable approximation of it. In Section 5.3, we describe a type determination algorithm for this restricted usage. Section 5.4 includes the description of our polynomial time combined algorithm for approximate type determination and aliasing for C^{++} with general purpose pointer dereferencing.

5.1 Problem Definition

5.1.1 Additional Terminology

- *Type determination* involves calculating the type of the object pointed to by a pointer at a program point as a result of an execution to that program point. We represent this information by a pair consisting of a pointer and an associated type (e.g. $<p \Rightarrow C>$), called a *pointer-type pair.*

- A dereference of an object name (using "$\rightarrow$" or "*" dereference operator) at a program point is *valid* if there exists a consistent path to the program point on which the object name points to an object of appropriate type. We only deal with valid dereferences.

- The *precise* solution for static type determination and aliasing at a program point is a set of pointer-type and alias pairs, each of which is a result of an execution on

```
void B::f( ) {              l :  p = &q;
   i :  r = &s;             ...
}                           m :  *p = new B;
void D::f( ) {              ...
   j :  r = &t;             n :  q→f( );
}
```

Figure 5.1: Interdependence of pointer-type and alias pairs

some consistent path to that program point. We denote the type determination solution at a program point n as *May-Pointer-Type*(n), and the aliasing solution as *May-Alias*(n).

5.1.2 Interdependence of Aliasing and Type Determination

In programs restricted to single level pointers, one pointer cannot be aliased to another, as this requires multiple levels of indirection [LR91]. As a result, when a pointer changes its value (to point to an object of another type), it does not affect the value of any other pointers. In this context, type determination impinges on aliasing since the receiver types decide which virtual function is invoked at a call site, and the invoked function can affect aliasing; however, aliasing plays no part in type determination.

Such a separation does not occur when we allow multiple level pointers. We illustrate the close interdependence between aliasing and type determination with the help of Figure 5.1. Assume that class `D` is derived from class `B` and that both `B::f` and `D::f` are virtual functions. The accompanying program segment includes two pointer assignment statements and a virtual function call. In the following cases, as expected, multiple levels of pointer dereferencing is required for aliasing to affect type determination.

unsafe type determination compromising safety of aliasing Suppose an unsafe solution of type determination at the virtual call node $call_n$ does not include $<q \Rightarrow B>$. Thus, $entry_{B::f}$ will not be considered an interprocedural successor of $call_n$. Since the function $B :: f$ is never called in the context of $call_n$, in turn $<*r, s>$ will not hold at $return_n$. Thus, the aliasing solution at $return_n$ will be unsafe.

unsafe aliasing compromising safety of type determination Suppose $<*p, q>$ is not available at an immediate predecessor of m. This will lead to omission of $<q \Rightarrow B>$ in the type determination solution at m, which is unsafe.

imprecise type determination leading to imprecise aliasing The presence of a spurious pointer-type pair $<q \Rightarrow D>$ at $call_n$ will lead to invocation of virtual function $D :: f$ from $call_n$. This will result in a spurious alias $<*r, t>$ to hold at $return_n$, rendering the alias solution at $return_n$ imprecise.

imprecise aliasing leading to imprecise type determination Suppose a spurious alias $<*p, u>$ holds at an immediate predecessor of pointer assignment node m. This will lead to a spurious pointer-type pair $<u \Rightarrow B>$ to hold at m, rendering the type determination solution at m imprecise.

Thus, aliasing affects type determination and *vice versa.* This interdependence motivated us to design a combined algorithm to solve the problems together.

5.1.3 Theoretical Complexity of the Problem

Theorem 5.1.1 *In the presence of single level pointers and virtual functions in C^{++}, precise program-point-specific type determination and aliasing is* NP*-hard.*

Proof outline: We reduce the 3-SAT problem to the above problem. Let the formula in conjunctive normal form (cnf) be made of m logic variables v_1, v_2, ..., v_m. Let the cnf be represented as

$$(l_{11} \vee l_{12} \vee l_{13}) \wedge (l_{21} \vee l_{22} \vee l_{23}) \wedge \cdots (l_{n1} \vee l_{n2} \vee l_{n3})$$

where each l_{jk} represents a *literal*: variable or its negation. The above cnf is satisfiable iff there exists a truth assignment to the logic variables which makes the formula *true.* As a part of our reduction, we construct a program segment polynomial in the size of 3-SAT problem and show that the cnf is satisfiable iff we can solve the point-specific type determination and aliasing problem precisely on the program segment, thereby proving

Define a class `True` and a derived class `False`. Let `true` and `false` be objects of the two classes respectively. Each class has a function called `and( )` and is virtual.

```
True *True::and (True *arg) {          True *False::and (True *arg) {
  return arg;                            return &false;
}                                      }
True *v1, *nv1, *v2, *nv2, ..., *vm, *nvm;
True *c1, *c2, ..., *cn;
L1:
 if(-) {v1 = &true; nv1 = &false;}
 else  {v1 = &false; nv1= &true;}
 ...
 if(-) {vm = &true; nvm = &false;}
 else  {vm = &false; nvm = &true;}
L2:
 if(-) c1=l11; else if(-) c1=l12; else c1=l13;
 if(-) c2=c1→and(l21); else if(-) c2=c1→and(l22); else c2=c1→and(l23);
 ...
 if(-) cn=cn-1→and(ln1); else if(-) cn=cn-1→and(ln2); else cn=cn-1→and(ln3);
L3:
```

Figure 5.2: Reducing 3-SAT to type determination and aliasing

that the problem is NP-hard. We represent each logic variable v_i as a program variable with the name $\mathtt{v}_i$ and its negation with an explicit program variable $\mathtt{nv}_i$. Also, each $\mathtt{l}_{jk}$ in the program segment corresponds to the literal l_{jk} from the cnf and represents a program variable $\mathtt{v}_i$ or $\mathtt{nv}_i$. Figure 5.2 shows the program segment.

A path between L1 and L2 in the program segment represents a truth assignment to the logic variables in the cnf. The code between L2 and L3 represents the cnf with each line as a conjunct and the compound *if* statement on each line represents the disjunction of literals. A program variable pointing to `true` implies that the corresponding logic variable in the cnf is assigned a *true* value.

Claim 1: cnf is satisfiable iff $\mathtt{c}_n$ may point to an object of class `True` at statement L3.

Claim 2: cnf is satisfiable iff $*\mathtt{c}_n$ may be aliased to `true` at statement L3.

Suppose the formula is satisfiable, then there exists a path from L1 to L2 such that in each row between L2 and L3 at least one $\mathtt{l}_{jk}$ points to `true`, defining an all-`true` path from L2 to L3. By construction, the variable $\mathtt{c}_n$ points to an object of class `True` at the end of that path. Equivalently, $*\mathtt{c}_n$ is aliased to `true` on the same path.

Suppose the formula is unsatisfiable. For every path from L1 to L2, there is at least one row between L2 and L3 where every $\mathtt{l}_{jk}$ points to `false`. This row will be responsible for the variable $\mathtt{c}_j$ through $\mathtt{c}_n$ to point to an object of class `False` from that row onwards. Thus at statement L3, $\mathtt{c}_n$ will never point to `True`. Again equivalently, $*\mathtt{c}_n$ will never be aliased to `true`. □

Corollary 5.1.1 *In the presence of multiple level pointers and virtual functions in C^{++}, precise program-point-specific type determination and aliasing is* NP*-hard.*

The theorem involves a subproblem of the problem mentioned in Corollary 5.1.1. □

Corollary 5.1.2 *In the presence of single level pointers and virtual functions in C^{++}, the problem of determining precisely the set of pointer-type and alias pairs which hold on a consistent path is* NP*-hard.*

Each alias and pointer-type pair from the set $\{<*\mathtt{c}_1, \mathtt{true}>, \ldots, <*\mathtt{c}_\mathtt{n}, \mathtt{true}>, <\mathtt{c}_1 \Rightarrow \mathtt{True}>, \ldots, <\mathtt{c}_\mathtt{n} \Rightarrow \mathtt{True}>\}$ holds on a consistent path to L3 iff the cnf is satisfiable, leading to Corollary 5.1.2. □

5.2 Problem Formulations

Precise Problem Formulation:

Conditional analysis [LR91] involves analyzing execution flow in a function, assuming certain information holds at the entry of the function. We formalize this notion as follows:

- A balanced path to an ICFG node n from entry node e of the function containing node n is called *conditionally consistent* with respect to an assumption set S of alias and pointer-type pairs, if for every edge $\ll call, entry \gg$ on the path, where *call* represents a virtual call with receiver *rec*, the following is true:

  Given that all the alias and pointer-type pairs in S hold at e, the execution defined by the subpath from e to *call* implies a pointer-type pair

$<rec \Rightarrow C>$ at *call* such that the virtual function represented by *entry* is invocable from *call*.

We denote such a path by $\mathcal{P}_{n,S}$.

- We define the *conditional type determination problem* as deciding for an ICFG node n, a pointer-type pair PT, and a set of alias and pointer-type pairs S, whether:

  there exists a conditionally consistent path with respect to S to node n on which PT holds, **and** there exists a consistent path from ρ to the entry node of function containing n on which every pair in S holds.

- Similarly, we define the *conditional aliasing problem* as deciding for an ICFG node n, an alias AP, and a set of alias and pointer-type pairs S, whether:

  there exists a conditionally consistent path with respect to S to node n on which AP holds, **and** there exists a consistent path from ρ to the entry node of function containing n on which every pair in S holds.

Approximation Formulation:

Note that the above formulation uses a set of alias and pointer-type pairs at *entry* nodes. Corollary 5.1.2 on p. 54 states the intractability of the problem of finding such a set, rendering this formulation computationally intractable. We approximate the joint solution of these two related problems by considering an assumption set S' containing at most one alias or pointer-type pair from S, chosen arbitrarily[25]. We use this pair

1. to approximate a conditionally consistent path to n, and
2. in subsequent conditional aliasing and type determination calculation, as the only explicitly mentioned pair from some actual assumption set S.

Note that using at most one pair from the assumption set leads to an approximation of consistent paths: any path which imposes a pair $AAPT$ at *entry* now qualifies as a

[25] S' serves as an approximation of the call stack at the invocation, similar to the reaching alias abstraction in [LR92, LRZ93]. An abstraction determines how well the algorithm approximates consistent paths.

consistent path imposing set S at *entry* as long as $AAPT \in S$. This is a conservative, safe approximation since the actual path imposing S will be one of the approximate consistent paths. Note also that in the original formulation, a consistent path which imposed the set S at *entry* was "extended" to node n using a conditionally consistent path which required the same set S to be present at *entry*. Clearly the single-pair formulation is less constrained in that a consistent path which imposes a set S at *entry* can now be extended to node n using any conditionally consistent path which uses an assumption set S_i as long as $AAPT \in S_i$. This is also a safe approximation because the conditionally consistent path with set S will be one of the many extensions for which $S_i = S$.

Approximating consistent paths leads to an approximation in creating object names with dereferences. Our algorithm treats a dereference of an object name as valid at a program point, if the analysis can infer that the object name points to an object of appropriate type on some approximation of a consistent path. In other words, we dereference an object name at a program point n only if it can be shown to be non-NULL on some static path to n. Henceforth, we will not explicitly qualify dereferences as valid, as we never create object names with dereferences which are not valid according to the approximation formulation.

A pointer-type or alias pair at an ICFG node n may be created as a result of interaction between two distinct pairs appearing as a result of executing the same consistent path to a predecessor of n. For example, $<p, *u>$ and $<q \Rightarrow int>$, at the predecessor m of a pointer assignment "`n : p = q`", will interact to produce a pointer-type pair $<*u \Rightarrow int>$ at n only if these pairs hold at m as a result of executing the *same* consistent path. In the approximation formulation, the lack of precise information regarding the equality of two consistent paths necessitates a safe decision that the two pairs may possibly hold on the same consistent path to the predecessor, and hence may interact. Continuing the same reasoning, $<p, *u>$, $<q \Rightarrow int>$ and $<*u \Rightarrow int>$ are considered to hold on the same consistent path to n because they all result from the pairs present at the same predecessor, i.e., m.

We define two predicates, *points-to-type*[26] and *may-hold*[27], with the following interpretation reflecting the approximation formulation. *points-to-type*$(n, AAPT, <p \Rightarrow C>)$ is *true* if there exists a consistent path from ρ to the *entry* node of the procedure containing node n, on which an alias or pointer-type pair $AAPT$ (if any)[28] holds **and** there exists a conditionally consistent path $\mathcal{P}_{n,\{AAPT\}}$ to n on which $<p \Rightarrow C>$ holds. Similarly, *may-hold*$(n, AAPT, <a, b>)$ is *true* if there exists a consistent path from ρ to the *entry* node of the procedure containing node n, on which $AAPT$ (if any) holds **and** there exists a conditionally consistent path $\mathcal{P}_{n,\{AAPT\}}$ to n on which $<a, b>$ holds. For efficiency, we have designed our approximation algorithm so that work is performed only for *true*-valued *may-hold* and *points-to-type* predicates.

Aliasing information is readily computable from the *may-hold* solution as follows:

$$\textit{May-Alias}(n) = \left\{ \ <a, b> \ \middle| \ \exists AAPT(\textit{may-hold}(n, AAPT, <a, b>)) \ \right\}$$

The type determination solution follows directly from the solution of *points-to-type* predicates:

$$\textit{May-Pointer-Type}(n) = \left\{ <p \Rightarrow C> \ \middle| \ \exists AAPT(\textit{points-to-type}(n, AAPT, <p \Rightarrow C>)) \right\}$$

Clearly, the algorithm for *May-Alias* and *May-Pointer-Type* is linear in the size of the *may-hold* and *points-to-type* solutions respectively.

To perform aliasing and type determination calculation in the presence of object names with dereferences, we will need the following "dereference theorem". Intuitively the theorem states that every object name with multiple dereferences is also aliased to another object name with fewer dereferences and a similar dereference sequence.

Theorem 5.2.1 *Given an object name q with $m \geq 1$ valid dereferences on a consistent path to program point n, for each $(1 \leq i \leq m)$ there exists an object name p with $(m-i)$ valid dereferences such that q is aliased to p on the same path* **and** *the dereference sequence of q beyond its i^{th} dereference is identical to the dereference sequence of p.*

---

[26]In our previous papers [PR94a, PR94b], *points-to-type* was called *points-to*.

[27]This predicate is a generalization of *may-hold* for C introduced in Section 3.2.1 (p. 21).

[28]AAPT may be $\emptyset$.

We prove the above theorem by induction on the number of dereferences in q. All aliases and dereferences appearing in the proof are with respect to the same consistent path to node n.

basis: Suppose $m = 1$.

- If q is of the form $r \rightarrow f$, where r is a fixed-location and f contains no dereferences (but may contain multiple member accesses with the "." operator): By definition of valid dereference, r points to an object of some class T such that $r \rightarrow f$ is a legal object name. Let u be the fixed-location name for that object. Clearly, $r \rightarrow f$ is aliased to $u.f$.
- If q is of the form $*r$, where r is a fixed-location: By definition of valid dereference, r points to an object of primitive type; let the fixed-location name for the object be u. In this case, $*r$ is aliased to u.

Note that u, being a fixed-location, has 0 dereferences.

induction hypothesis: Suppose true for object names with $< m$ dereferences.

induction step: Given an object name p with m dereferences, it must be of the form $r \rightarrow f$ or $*r$, where r is an object name with $m - 1$ valid dereferences and f contains no dereferences. We will only consider the former case, as the treatment for latter (i.e. $*r$) is similar. Using the induction hypothesis, for each $i < m$, r is aliased to an object name p_i with $(m - 1 - i)$ valid dereferences and identical dereference sequence after i^{th} dereference. Since r is aliased to p_i, $r \rightarrow f$ is aliased to $p_i \rightarrow f$. Finally, by definition of valid dereference for $r \rightarrow f$, $r \rightarrow f$ is aliased to $u.f$ where u is a fixed-location name for an object of some class T such that $u.f$ is a legal object name.

Corollary 5.2.1 *An object name with a non-empty sequence of all valid dereferences on a consistent path is aliased to a fixed-location on that path.*

For $i = m$ in Theorem 5.2.1, we have the corollary. □

5.3 Type Determination Algorithm for Single Level Pointers

As we mentioned in Section 5.1.2, type determination and aliasing are separable when the C^{++} programs are restricted to the use of single level pointers. Since a major motivation of this research is virtual function resolution which is closely related to type determination, we exploit this separability to only describe the type determination aspect of the combined problem in full detail. In Section 5.4, we will present the combined algorithm where we will concentrate on the interaction between types and aliases.

A Running Example: Before discussing the algorithm, we examine a program segment in Figure 5.3. We will use it in Section 5.3.1 to illustrate the significance of type determination in practical issues of run-time efficiency and benefits to other optimizations. Throughout Section 5.3.2, we will use it as an illustration for our algorithm description and in Section 5.3.3, to illustrate the sources of approximation for the algorithm. We assume that the boolean conditions in *if* statements are side effect free and hence inconsequential to the analysis.

5.3.1 Practical Issues

At node $n13$ in Figure 5.3, the pointer q is made to point to an object of class $Base$, and then immediately used at node $n14$ as the receiver for a virtual function invocation. Under these circumstances $Base :: foo()$ will be invoked on all executions notwithstanding the virtual nature of the invocation. Since the virtuality of $Base :: foo()$ is not utilized, the invocation can be compiled as a function call, thereby reducing the run time overhead of virtual invocation.

Limiting the scope of invocation to $Base :: foo()$ and eliminating $Derived :: foo()$ from consideration may benefit other analyses. The assignments at nodes $n1$ and $n3$, and printing *hello world* at $n2$ will not appear as possible side effects of the invocation at node $n14$. As another significant implication, our algorithm will be able to determine that $n15$ is a call of $Base :: bar()$ and never $Derived :: bar()$, because the receiver a may only point to an object of type $Base$. Therefore, call site $n15$ can be considered

```
class Base {
public:
        virtual foo ( );
        virtual bar ( );
        virtual baz ( );
} *a, *b, *p, *q;

Base::foo ( ) {
   n0 : a = new Base;
}

Base::bar ( ) {
    ...
}

Base::baz ( ) {
    ...
}
```

```
class Derived : public Base {
public:
        foo ( );
        bar ( );
}  r, *s;

Derived::foo ( ) {
   n1 : a = new Derived;
   n2 : printf ("hello world\n");
   n3 : b = new Derived;
}

Derived::bar ( ) {
    ...
}
```

```
main ( )  {
    if (-) {
        n4 : p = new Base;
        n5 : q = new Derived;
        n6 : b = new Base;
    } else {
        n7 : p = new Derived;
        n8 : q = new Base;
    }
    n9  : s = &r;
    if (-)
        n10 : s->Derived::bar ( );
    n11 : p->foo ( );
    n12 : q->foo ( );
    n13 : q = new Base;
    n14 : q->foo ( );
    n15 : a->bar ( );
    n16 : p->baz ( );
}
```

Figure 5.3: Example for type determination algorithm

non-virtual. Given the potential disparity in side effects of virtual functions which share the same name, type determination can significantly improve the precision of analysis.

Resolving a virtual function invocation to a unique function call may create possibilities for inlining, resulting in elimination of function call overhead. Inlining a function call can also provide opportunities for various intraprocedural optimizations.

A transformation from virtual invocation to function call is sometimes possible without complete resolution of the receiver type. For example at node $n16$, the receiver p may point to an object of class *Base* or *Derived*. Since the receiver type is not unique, a naive approach may result in retaining the invocation as virtual. However, since class *Derived* inherits the function $baz()$ from class *Base* without redefining it, $n16$ may still be safely compiled as a function call to $Base :: baz()$. In general, even if the receiver at the virtual invocation site does not point to a unique class, but all the receiver types utilize the same virtual function, the virtual invocation may be compiled as a function call.

For architectures which use deep pipelining and speculative execution, the issue of accurate control flow prediction assumes significant importance. Using static type determination to replace virtual invocations with function calls, when the target function is known at compile time, would yield benefits comparable to those obtained by profile-based prediction for C^{++} [CG94].

5.3.2 Algorithm Description

The algorithm to perform type determination for C^{++} programs with single level pointers is formally stated in Appendix B.1. In the remainder of this section, we present an equivalent approach, more amenable to understanding and implementation.

To determine the type of an object to which a pointer variable may point at a given program point, we perform a fixed point computation of the data flow equations describing the semantic effects of C^{++} statements. Since type determination is separable from aliasing in this scenario, the equations only involve the *points-to-type* predicates. We present an algorithm which is both *safe* and *approximate* for C^{++} programs with only single level pointers: if there exists a path to node n on which $<ptr \Rightarrow C>$ holds during some execution, our algorithm will report a *true*-valued predicate *points-to-type*(n, APT, $<ptr \Rightarrow C>$) for some APT, guaranteeing the safety of calculation. The assumption APT is either $\emptyset$ or a pointer-type pair, and never involves an alias, reflecting the independence of type determination from aliasing.

The calculation of a fixed point for *points-to-type* is tantamount to the solution of a monotone data flow framework defined on a lattice whose elements are sets of (assumption, pointer-type pair), as described in Appendix B.1. We use a worklist for this calculation. Whenever a predicate *points-to-type*(n, APT, PT) becomes *true* for the first time, it is placed on the worklist. Once marked *true*, a predicate stays *true*, ensuring monotonicity of calculation. A predicate goes on the worklist at most once, guaranteeing the termination of our algorithm. We refer to this action as **make-true** and denote it in the algorithm by "**make-true**(*points-to-type*(n, APT, PT))".

We describe the algorithm in three phases: (i) we *initialize* the information, (ii) during the *introduction* phase we annotate each node appropriately with the information obtained locally at the node itself, and (iii) we *propagate* the information throughout the ICFG until stabilization. All *points-to-type* predicates are assumed *false* initially. For efficiency, we have designed the algorithm in such a way that work is performed only for *points-to-type*(n, APT, PT) which become *true*.

Initialization and introduction phases

Conceptually, we start with no information at any of the ICFG nodes by initializing each possible *points-to-type* predicate to *false*. The time complexity of the initialization of the entire *points-to-type* predicate may appear as proportional to the number of predicates possible, but we have a constant time initialization by following a lazy approach [LR92]. We also initialize the worklist to empty.

The first entries in the worklist come from the introduction phase during which we **make-true** certain predicates at a node by looking at the local information available in the node itself. Figure 5.4 lists the nodes examined in the introduction phase and their associated actions. Note that in item 3 we restrict ourselves to non-virtual function

```
for each node n in the ICFG
   If n is
      1.  n :  p = new T
          make-true(points-to-type(n, ∅, <p ⇒ T>))
      2.  n :  p = &r
          make-true(points-to-type(n, ∅, <p ⇒ type(r)>))
            where type(r) returns the type of object name r.
      3.  n :  foo(param_1, ..., param_k)
          make-true(points-to-type(entry_foo, <p ⇒ type(r)>, <p ⇒ type(r)>))
            for each param_i of the form &r with pointer variable p as
            the corresponding formal, and call is non-virtual.
          make-true(points-to-type(entry_foo, <p ⇒ T>, <p ⇒ T>))
            for each param_i of the form new T with pointer variable
            p as the corresponding formal, and call is non-virtual.
```

Note: p and r represent fixed-locations.

Figure 5.4: Introduction phase

calls, because without the knowledge of the receiver type, we cannot determine which function is invoked from a virtual call site. We handle virtual function calls during the propagation phase.

Using the program segment in Figure 5.3 (p. 60), we list the following examples of type introduction. Using item 1 we have

$$\textbf{make-true}(\textit{points-to-type}(n4, \emptyset, <p \Rightarrow \textit{Base}>))$$

reflecting the fact that $<p \Rightarrow \textit{Base}>$ holds on any balanced path from $entry_{main}$ to $n4$ without assuming any information at $entry_{main}$. Similarly, using item 1 we also have

$$\textbf{make-true}(\textit{points-to-type}(n7, \emptyset, <p \Rightarrow \textit{Derived}>)).$$

At node $n9$, using item 2 and the fact that r is an object of class *Derived*, we have

$$\textbf{make-true}(\textit{points-to-type}(n9, \emptyset, <s \Rightarrow \textit{Derived}>)).$$

Propagation phase

During the propagation phase, the worklist entries are processed one at a time. Processing a worklist entry implies propagating the effects of the pair PT holding at node

```
while worklist is not empty
        remove (m, APT, <ptr ⇒ C>) from worklist
        if m is a call node
                type-implies-type-from-call (m, APT, <ptr ⇒ C>)
        else if m is an exit node
                type-implies-type-from-exit (m, APT, <ptr ⇒ C>)
        else
                type-implies-type-through-other (m, APT, <ptr ⇒ C>)
```

Figure 5.5: Propagation phase

m given the assumption APT, to all the successors of the node m, and then removing the entry from the worklist. New entries which become *true* as a result of this action are placed on the worklist. The computation reaches a fixed point when the worklist becomes empty. We describe this phase as a case analysis on the kind of logical successor of each worklist entry. Figure 5.5 illustrates the propagation phase at a high level with the help of three propagation functions: **type-implies-type-through-other**, **type-implies-type-from-call** and **type-implies-type-from-exit**. In the following discussion, we explicate the high level view by describing each propagation function.

type-implies-type-through-other$(m, APT, <ptr \Rightarrow C>)$**:** This function captures the intraprocedural aspects of type propagation as described in the following cases. **case 1** and **case 3** are mutually exclusive, but either may appear with **case 2**.

case 1: If successor is an assignment to *ptr*, "`n : ptr = ...`", the given *points-to-type* does not propagate through n, as the value of *ptr* is updated as a result of this assignment.

case 2: If successor is an assignment of *ptr* to a pointer variable *aptr*, with or without casting (within inheritance hierarchy): "`n : aptr = ptr`" or "`n : aptr = (E *) ptr`"

$$\textbf{make-true}(\textit{points-to-type}(n, APT, <aptr \Rightarrow C>)).$$

Type casting appears in the latter node so that the assignment is type-correct, but it is unimportant to our analysis as the semantics of a pointer-type pair dictate that we record the type of object to which the pointer *actually* points, and not the type of object to which it is *declared* to point. Since *ptr* points to an object with type C prior to assignment n, *aptr* points to an object with type C at n, and not to an object with type E.

case 3: If successor node n does not define the pointer variable *ptr*, then the type of *ptr* is preserved: **make-true**(*points-to-type*(n, APT, $<ptr \Rightarrow C>$)). This is a case of simple propagation of information without any change. In the example for type introduction, we inferred *points-to-type*($n4$, $\emptyset$, $<p \Rightarrow Base>$) and *points-to-type*($n7$, $\emptyset$, $<p \Rightarrow Derived>$). Propagating this information through the type preserving successors up to node $n9$, we **make-true** the following predicates.

$$\textit{points-to-type}(n9, \emptyset, <p \Rightarrow Base>) \quad \text{and} \quad \textit{points-to-type}(n9, \emptyset, <p \Rightarrow Derived>)$$

Using further applications of **case 3**, the information at $n9$ propagates to its successors with **make-true** for each of the following.

$$\begin{array}{ll} \textit{points-to-type}(call_{n10}, \emptyset, <p \Rightarrow Base>) & \textit{points-to-type}(call_{n11}, \emptyset, <p \Rightarrow Base>) \\ \textit{points-to-type}(call_{n10}, \emptyset, <p \Rightarrow Derived>) & \textit{points-to-type}(call_{n11}, \emptyset, <p \Rightarrow Derived>) \\ \textit{points-to-type}(call_{n10}, \emptyset, <s \Rightarrow Derived>) & \textit{points-to-type}(call_{n11}, \emptyset, <s \Rightarrow Derived>) \end{array}$$

type-implies-type-from-call (*call*, APT, $<ptr \Rightarrow C>$)**:** This function is responsible for propagating a pointer-type pair at the call site to appropriate *entry* and *return* nodes. We consider the following cases.

case 1: Propagation is simpler when the corresponding *entry* is readily known, typically when *call* represents a non-virtual function invocation. As we already saw, *points-to-type*($call_{n10}$, $\emptyset$, $<s \Rightarrow Derived>$) is *true*. Since s is visible in the called function $Derived :: bar()$, we **make-true**

$$\textit{points-to-type}(entry_{Derived::bar}, <s \Rightarrow Derived>, <s \Rightarrow Derived>).$$

We also associate the type of object pointed to by each actual parameter with the corresponding formal parameter. For example, at the call site $n10$, s is the first actual parameter and corresponds to the formal *this* of $Derived :: bar()$. Since *points-to-type*($call_{n10}$, $\emptyset$, $<s \Rightarrow Derived>$) is *true*, we **make-true**

$$\textit{points-to-type}(entry_{Derived::bar}, <this \Rightarrow Derived>, <this \Rightarrow Derived>).$$

If *ptr* is not visible in the called function, the type pointed to by *ptr* cannot change[29]. In this case we propagate the predicate *points-to-type*(*call*, *APT*, <*ptr* ⇒ *C*>) directly to the corresponding *return* node as

$$\textbf{make-true}(\textit{points-to-type}(return, APT, <\!ptr \Rightarrow C\!>)).$$

case 2: If the call is virtual and the call site is: "`m : rec→fun ( )`".

Irrespective of which functions may be invoked by this virtual call, if the pointer variable *ptr* is local to the calling function, the predicate on the worklist propagates directly to the *return* node with

$$\textbf{make-true}(\textit{points-to-type}(return, APT, <\!ptr \Rightarrow C\!>))$$

The entry nodes to which the effects of the given worklist entry have to be propagated depend on the type(s) of objects the receiver *rec* may point to at the call site. Two circumstances are possible: (i) some typing information is already available at the virtual call site before resolving a function to be invocable (**case 2.1**), and (ii) a function is resolved to be invocable before all the typing information to be propagated has reached the virtual call site (**case 2.2**).

case 2.1: $ptr == rec$ (i.e. *ptr* is the same variable as the receiver *rec*)

1. The effect of this *points-to-type* needs to be propagated only to the function invocable when the receiver points to an object of class C. In the example, *points-to-type*($call_{n11}$, $\emptyset$, <$p \Rightarrow Base$>) propagates to $entry_{Base::foo}$ but not to $entry_{Derived::foo}$, with

   $\textbf{make-true}(\textit{points-to-type}(entry_{Base::foo}, <\!p \Rightarrow Base\!>, <\!p \Rightarrow Base\!>))$ and
   $\textbf{make-true}(\textit{points-to-type}(entry_{Base::foo}, <\!this \Rightarrow Base\!>, <\!this \Rightarrow Base\!>)).$

   On the other hand, *points-to-type*($call_{n11}$, $\emptyset$, <$p \Rightarrow Derived$>) propagates to $entry_{Derived::foo}$ (with **make-true** of following predicates),

   $\textit{points-to-type}(entry_{Derived::foo}, <\!p \Rightarrow Derived\!>, <\!p \Rightarrow Derived\!>)$
   $\textit{points-to-type}(entry_{Derived::foo}, <\!this \Rightarrow Derived\!>, <\!this \Rightarrow Derived\!>)$

   but not to $entry_{Base::foo}$.

[29]This is true because we only have single level pointers.

2. The effects of other accumulated information at the call site are propagated through the appropriate function(s) as follows:

   a) If the function call involves passing of an object address $\&r$ as an actual to a pointer formal f: **make-true**(*points-to-type*($entry$, $<f \Rightarrow type(r)>$, $<f \Rightarrow type(r)>$)). Note that this case cannot be handled in the introduction phase, as the invocability of this function from node m would not be known then.

   b) For each *points-to-type*($call$, $APT1$, $<ptr' \Rightarrow E>$) where $ptr' \neq rec$: We determine the corresponding entry and perform actions as in **case 1**. In the example, during **type-implies-type-from-call**($call_{n11}, \emptyset, <p \Rightarrow Base>$) involving the receiver p, we propagate *points-to-type*($call_{n11}$, $\emptyset$, $<b \Rightarrow Base>$) to $entry_{Base::foo}$ with

   $$\textbf{make-true}(\textit{points-to-type}(entry_{Base::foo}, <b \Rightarrow Base>, <b \Rightarrow Base>))$$

case 2.2: ptr and rec are distinct variables:

Suppose the *points-to-type* information currently available about the receiver rec at the given $call$ node is:

$$\textit{points-to-type}(call, APT1, <rec \Rightarrow C1>) \text{ and}$$
$$\textit{points-to-type}(call, APT2, <rec \Rightarrow C2>)$$

According to this information, the receiver rec may point to an object of type $C1$ or $C2$ at the call site depending on the execution path. So the virtual function call $rec \rightarrow fun()$ may lead to the invocation of two distinct virtual functions with name *fun*. Hence the effects of the given worklist entry need to be propagated to the entry nodes of each of these invocable functions. This is done in the same fashion as for **case 1**, considering one *entry* node at a time. In the example, suppose *points-to-type*($call_{n11}$, $\emptyset$, $<s \Rightarrow Derived>$) is the candidate for propagation at $call_{n11}$. We have also seen that *points-to-type*($call_{n11}$, $\emptyset$, $<p \Rightarrow Base>$) and *points-to-type*($call_{n11}$, $\emptyset$, $<p \Rightarrow Derived>$) are *true* at $call_{n11}$ with receiver p. Thus there are two distinct functions $Base :: foo()$ and $Derived :: foo()$ which may be invoked at $call_{n11}$. As a result, we propagate *points-to-type*($call_{n11}$, $\emptyset$,

$<s \Rightarrow Derived>$) to the corresponding *entry* nodes using

make-true(*points-to-type*($entry_{Base::foo}$, $<s \Rightarrow Derived>$, $<s \Rightarrow Derived>$))
make-true(*points-to-type*($entry_{Derived::foo}$, $<s \Rightarrow Derived>$, $<s \Rightarrow Derived>$)).

Handling *memoization***:** A significant property of our implementation of the conditional approach is that we analyze a function only once under each reaching assumption, even if the same assumption may be imposed multiple times from different call sites. This *memoization* necessitates the following additional action by **type-implies-type-from-call**: Suppose *points-to-type*(*call*, $APT1$, $APT2$) results in *points-to-type*(*entry*, $APT3$, $APT3$) being propagated to some *entry* node using any of the various cases in **type-implies-type-from-call**, but that the latter is already *true* there. It is possible that the called function has already been analyzed for assumption $APT3$ and there exists some *points-to-type*(*exit*, $APT3$, PT) at the exit of the function. When **type-implies-type-from-exit** (described below) propagated its effects to the return nodes known at that time, it did not include the *return* of current call. In such cases, **type-implies-type-from-call** utilizes *points-to-type*(*exit*, $APT3$, PT) to **make-true**(*points-to-type*(*return*, $APT1$, PT)).

type-implies-type-from-exit (*exit*, APT, $<ptr \Rightarrow C>$)**:** Lastly we describe how the type information propagates from *exit* node to the corresponding *return* node(s). Let *exit* be the exit node of a function $fun()$ and the return nodes corresponding to *exit* be $r_1, r_2, \cdots, r_k$ at the instant of processing this worklist entry. New return nodes may be added later, when the function is determined to be invocable from other virtual function call sites. We do not consider them at this time. As explained earlier, when a new virtual function is determined to be invocable from a call site we propagate the effects of that call using **case 2.1** of **type-implies-type-from-call**. Let the call nodes corresponding to these return nodes be $c_1, c_2, \cdots, c_k$. We do the following for each return node r_i:

If *ptr* is not visible in the function containing the return node r_i, we take no propagation action. Since the variable itself goes out of scope, we do not need to know its type. However if *ptr* is visible in the function containing the return node r_i, we have the following cases:

case 1: If APT is non-$\emptyset$, this implies that APT holds at *entry* in order that *ptr* points to an object of class C at *exit*. Each call node c_i responsible for imposing APT at *entry* in turn leads to $<ptr \Rightarrow C>$ holding at its corresponding return node r_i. If APT is imposed at *entry* of the invoked function without requiring any *points-to-type* predicate at the call node c_i (i.e. *points-to-type*($entry$, APT, APT) was made *true* during introduction phase), we simply propagate $<ptr \Rightarrow C>$ to r_i. In this case: **make-true**(*points-to-type*($r_i, \emptyset, <ptr \Rightarrow C>$)). On the other hand, suppose it took *points-to-type*(c_i, $APT2$, $PT1$) to impose APT at the *entry*; then we have: **make-true**(*points-to-type*(r_i, $APT2$, $<ptr \Rightarrow C>$)).

In the example, when *points-to-type*($exit_{Base::foo}$, $<p \Rightarrow Base>$, $<p \Rightarrow Base>$) is propagated, there are two possible return nodes *viz.* $return_{n11}$ and $return_{n14}$. Since it takes *points-to-type*($call_{n11}$, $\emptyset$, $<p \Rightarrow Base>$) to impose $<p \Rightarrow Base>$ at $entry_{Base::foo}$, using the information thus available at $call_{n11}$ and $exit_{Base::foo}$:

$$\textbf{make-true}(\textit{points-to-type}(return_{n11}, \emptyset, <p \Rightarrow Base>))$$

As there is no assignment to p on any path from $call_{n11}$ to $call_{n14}$, we have *points-to-type*($call_{n14}$, $\emptyset$, $<p \Rightarrow Base>$). This predicate also imposes $<p \Rightarrow Base>$ at $entry_{Base::foo}$. While propagating *points-to-type*($exit_{Base::foo}$, $<p \Rightarrow Base>$, $<p \Rightarrow Base>$), we use this information available at $call_{n14}$ to

$$\textbf{make-true}(\textit{points-to-type}(return_{n14}, \emptyset, <p \Rightarrow Base>)).$$

case 2: If $APT == \emptyset$, implying that $<ptr \Rightarrow C>$ holds at *exit* without any assumption at *entry* of the function, we directly propagate $<ptr \Rightarrow C>$ to r_i using **make-true**(*points-to-type*(r_i, $\emptyset$, $<ptr \Rightarrow C>$)).

Improving precision at *return* nodes for virtual calls: In all the above cases, there is an opportunity for improving the precision, although we have not incorporated it in our algorithm or the implementation. Suppose a function $E :: foo()$ is invocable from a virtual call site with receiver l due to the presence of *points-to-type*(c_i, $APT3$, $<l \Rightarrow E>$). Suppose further that a predicate is being propagated (using **case 1** or **case 2**) from $exit_{E::foo}$ to the return node r_i as *points-to-type*(r_i, $\emptyset$, $<p \Rightarrow C>$). Since $APT3$ is indirectly responsible for $<p \Rightarrow C>$ to hold at r_i by making $E :: foo()$ invocable from c_i, we replace the $\emptyset$ assumption with $APT3$ to obtain a more precise *points-to-type*(r_i, $APT3$, $<p \Rightarrow C>$).

5.3.3 Sources of Approximation

We illustrate the sources of approximation in our type determination algorithm for single level pointers using two examples from Figure 5.3 (p. 60).

1. At the virtual call node $call_{n11}$, we have the following predicates:

$$\textit{points-to-type}(call_{n11}, \emptyset, <p \Rightarrow Derived>) \text{ and}$$
$$\textit{points-to-type}(call_{n11}, \emptyset, <b \Rightarrow Base>).$$

   The two predicates are *true* on two **distinct** consistent paths through the assignment nodes $n7$ and $n6$ respectively. However, in the absence of complete path specific information, at $call_{n11}$ we must conservatively assume that the predicates may be *true* on the **same** path. The former predicate makes the function $Derived :: foo()$ invocable from $call_{n11}$, and we imprecisely propagate the latter predicate to $entry_{Derived::foo}$ by

$$\textbf{make-true}(\textit{points-to-type}(entry_{Derived::foo}, <b \Rightarrow Base>, <b \Rightarrow Base>))$$

2. The following predicates are *true* on the **same** path to the virtual call node $call_{n11}$:

$$\textit{points-to-type}(call_{n11}, \emptyset, <p \Rightarrow Base>) \text{ and}$$
$$\textit{points-to-type}(call_{n11}, \emptyset, <b \Rightarrow Base>).$$

   Thus we propagate the latter predicate to $Base :: foo()$ with

$$\textbf{make-true}(\textit{points-to-type}(entry_{Base::foo}, <b \Rightarrow Base>, <b \Rightarrow Base>))$$

   As a result of further propagation through the function, we have

$$\textbf{make-true}(\textit{points-to-type}(exit_{Base::foo}, <b \Rightarrow Base>, <b \Rightarrow Base>))$$

When we propagate this predicate to appropriate *return* nodes, the presence of

$$\textit{points-to-type}(call_{n12}, \emptyset, <q \Rightarrow Base>) \text{ and}$$
$$\textit{points-to-type}(call_{n12}, \emptyset, <b \Rightarrow Base>)$$

at $call_{n12}$ implies **make-true**($\textit{points-to-type}(return_{n12}, \emptyset, <b \Rightarrow Base>)$). This is a conservative decision since the above predicates are $true$ on **distinct** paths to $call_{n12}$, so there is no consistent path to $return_{n12}$ on which $<b \Rightarrow Base>$ holds.

In general, the algorithm is rendered imprecise on two counts: 1) while propagating a predicate from a virtual call to the entry of a function, and 2) while propagating a predicate from exit of a function to an associated return node of a virtual call site. In both cases, the imprecision is the outcome of our safe decision to assume that a single consistent path may have made two predicates $true$ at the virtual call site.

5.4 Type Determination and Aliasing Algorithm for Multiple Level Pointers

Having described the type determination algorithm for single level pointer dereferencing in Section 5.3, we now concentrate on handling the interaction between types and aliases in the presence of multiple level pointers. When the calculation only involves types, we proceed as in Section 5.3. When the calculation only involves aliases without using or affecting any type information, the transfer functions are similar to that in [LR92, Lan92a].

5.4.1 Algorithm Overview

Our combined algorithm for aliasing and type determination is a worklist based, fixed point iteration method which is both *safe* and *approximate*: If there exists an execution path to ICFG node n on which a pointer p points to an object of type C, our algorithm will report *points-to-type*(n, $AAPT$, $<p \Rightarrow C>$) for some entry assumption $AAPT$. Similarly, if there exists an execution path to node n on which $<a, b>$ holds, our algorithm will report *may-hold*(n, $AAPT$, $<a, b>$) for some $AAPT$.

```
// initialization of information (Section 5.4.3)
lazily set all possible predicates to false;
set worklist to empty;
// introduction of aliases and pointer-type pairs (Section 5.4.3)
intra-alias-type-introduction ( ); // (Figure 5.8)
for each non-virtual call to entry
     inter-alias-type-introduction (call, entry);
// propagation of aliases and pointer-type pairs
while worklist is not empty
     remove (m, AAPT, APT) from worklist
     if m is a call node // (Section 5.4.5)
         if APT is an alias pair
             alias-implies-from-call (m, AAPT, APT);
         if APT is a pointer-type pair
             type-implies-from-call (m, AAPT, APT);
     elseif m is an exit node // (Section 5.4.5)
         if APT is an alias pair
             alias-implies-alias-from-exit (m, AAPT, APT);
         if APT is a pointer-type pair
             type-implies-type-from-exit (m, AAPT, APT);
     else // intraprocedural propagation (Section 5.4.4)
         for each n ∈ successor (m)
             if n is a pointer assignment // (Table 5.1)
                 if APT is an alias pair
                      alias-implies-alias-thru-assign (n, m, AAPT, APT);
                      alias-implies-type-thru-assign (n, m, AAPT, APT);
                 if APT is a pointer-type pair
                      type-implies-type-thru-assign (n, m, AAPT, APT);
                      type-implies-alias-thru-assign (n, m, AAPT, APT);
             else // propagate directly through n
                 if APT is an alias pair
                      make-true(may-hold (n, AAPT, APT))
                 if APT is a pointer-type pair
                      make-true(points-to-type (n, AAPT, APT))
```

Figure 5.6: A high level description of the algorithm

A high level description of our algorithm appears in Figure 5.6. The algorithm has three main phases which are discussed using examples in Sections 5.4.3–5.4.5. Firstly, the predicates *points-to-type* and *may-hold* and the worklist are *initialized* (see Section 5.4.3). Secondly, we *introduce* certain *true*-valued predicates at pointer assignments (using **intra-alias-type-introduction**) and at parameter binding sites (using **inter-alias-type-introduction**) (see Section 5.4.3). These initial predicates are placed on the worklist. Thirdly, the algorithm performs the usual fixed point iteration, until the worklist is empty. That is, a predicate is removed from the worklist and *propagated* through successor nodes using appropriate functions determined by the node type. This propagation of information occurs in the `while` loop of Figure 5.6. Intraprocedural propagation is explained in Section 5.4.4; the interprocedural propagation functions (e.g., **alias-type-propagation-from-call**) are explained in Section 5.4.5. The propagation functions make additional predicates *true* and put them on the worklist. In this section, we use **make-true** on both *points-to-type* and *may-hold* to set each one to *true* when appropriate and add it to the worklist exactly once.

Our algorithm has a high-level structure that corresponds to a *lazy* evaluation of the interactions between *points-to-type* and *may-hold* predicates. Several of the auxiliary functions used to explain the algorithm may appear to be redundant. In fact, some of them are duals of the others in the following sense: more than one predicate may be necessary at a program point to imply a consequence at a successor. These predicates are propagated to the program point in indeterminate order by our worklist algorithm. The propagation phase must deal with the situation where any one of them is the last one propagated to that program point and use it with the other predicates already propagated to imply the consequence. The cases described in this section include this duality of order of processing both during the intraprocedural (Section 5.4.4) and interprocedural (Section 5.4.5) propagations.

Our calculation of a fixed point for *points-to-type* and *may-hold* is tantamount to the solution of a monotone data flow framework, defined over the ICFG, with a lattice whose elements are sets of (*assumption1, assumption2, alias/pointer-type*) tuples (See Appendix B.2).

```
class B { public:                 class C { public:
  int b1;                           int foo (int *f1, int *f2, int **f3) {
};                                      n6 : p = new int;
                                        n7 : *f3 = &i;
class D { public                    }
  B *d1;                            int bar (D *bar1; B *bar2);
};                                };

C *s;
int *p, i;
main () {
  D *r;
  int *q, *t;
  n1 : p = &i;
  n2 : q = p;
  n3 : s = new C;
  c1 : s->foo (p, q, &t);
  n4 : r = new D;
  n5 : r->d1 = new B;
  c2 : s->bar (r, r->d1);
}
```

Figure 5.7: Example for binding functions

5.4.2 Calculation of Approximate Assumption Sets

In order to perform interprocedural analysis in the presence of recursive functions and local variables, we need the concept of *non-visible* object names, similar to that described in [Lan92a]. A called function can both create or destroy an alias in the calling function containing an object name not visible in the called function. A called function can also create, but not destroy, a pointer-type pair in the calling function containing an object name not visible in the called function. The only way to destroy an incoming pointer-type pair is by re-assigning the pointer directly, not through an alias (remember, aliasing is a *may* problem). Obviously the called function cannot directly assign to a non-visible object name. We illustrate the above circumstances using Figure 5.7. Alias $<*p, *q>$ reaches the function foo via $call_{c1}$. Although $*q$ is not visible in foo, the alias is destroyed in foo by the re-assignment of p at node $n6$. Although t is not visible in function foo, it is aliased to $*f3$ in foo via call site $call_{c1}$. Node $n7$ creates an alias $<*t, i>$ through an assignment to $f3$. The same assignment is also responsible for creating a pointer-type $<t \Rightarrow int>$. In order to represent non-visible object names, we create a special fixed-location name for each type T in the program, we call it nv_T. In the called function, each occurrence a non-visible object name with a (possibly empty) dereference sequence applied to a fixed-location of type T is replaced by an object name with the identical dereference sequence applied to nv_T[30]. In Figure 5.7, we would represent $*q$ as $*nv$ in function foo.

Before providing algorithm details, we describe some auxiliary functions used to capture type and aliasing effects on entry of an invoked function from the types and aliases present at the invocation site. These functions calculate the approximate assumption sets with cardinality of at most one, described earlier. The first two functions, **bind0** and **type-bind0**, are used during the introduction phase (Section 5.4.3) and the rest during interprocedural propagation (Section 5.4.5). In these descriptions, `call` and `entry` represent ICFG nodes whereas *alias* and *pointer-type* are specific pairs.

bind0(`call, entry`) : This function calculates those *aliasing effects* from `call` to `entry` requiring no information from the predecessor(s) of `call`. In other words, only the information local to `call` is utilized. For example, if $\&a$ is visible in the called function and is passed as an actual to the formal $f1$, $<*f, a>$ is created at `entry` regardless of any aliases a may have at `call`. Also, $\&a$ and $\&a.mem$ passed to the formals $f1$ and $f2$ respectively cause the creation of $<*f1 \rightarrow mem, *f2>$ at `entry`. In this example, **bind0**(`call`, `entry`) =

$$\{<*f1, a>, <a.mem, *f2>, <f1 \rightarrow mem, *f2>, <f1 \rightarrow mem, a.mem>\}$$

If a is non-visible, then **bind0**(`call`, `entry`) =

$$\{<*f1, nv>, <nv.mem, *f2>, <f1 \rightarrow mem, *f2>, <f1 \rightarrow mem, nv.mem>\}$$

Note : A pointer a passed to formal f does not create an alias $<*a, *f>$ because it would require information from the predecessor(s) of `call` regarding the type of object

[30]Although the type information is important for the algorithm, for notational convenience we simply use nv, and let the type be implicit.

pointed to by a. This case is handled by Rule 2 of **alias-bind** which we will describe shortly.

type-bind0`(call, entry)` : This function calculates those *type effects* from `call` to `entry` requiring no information from the predecessor(s) of `call`. Like *bind0*, only those cases are relevant where an address of an object name is passed as actual. For example, suppose a is an object of class B, a is visible in the called function, and $\&a$ is passed as actual to the formal f. Then $<f \Rightarrow B> \in$ **type-bind0**(`call`, `entry`).

We examine the details of the remaining functions with respect to the example in Figure 5.7.

bind`(call, entry,` *alias*`)` : This function represents the propagation of *alias* reaching the `call` to the corresponding `entry`. Depending on the actual-formal associations, *alias* may manifest itself at `entry` and/or may give rise to additional alias pairs. In Figure 5.7, $<*p, *q>$ is created at $n2$ and reaches $call_{c1}$. The aliases created at $entry_{C::foo}$ by this pair fall into the following three categories:

1. aliases between dereferences of formals: Since the dereferences of two actuals are aliased at $call_{c1}$, the dereferences of corresponding formals are aliased at $entry_{C::foo}$.

2. aliases between an alias of a dereference of an actual and an appropriate dereference of the corresponding formal: Since $*p$ is aliased to $*q$ at $call_{c1}$, $*f1$ is aliased to $*nv$ at $entry_{C::foo}$. Similarly, $*f2$ is aliased to $*p$ at $entry_{C::foo}$.

3. *alias* itself propagating to `entry`: $<*p, *nv>$ holds at $entry_{C::foo}$.

For this example,

$$\textbf{bind}(call_{c1}, entry_{C::foo}, <*p, *q>) = \{<*f1, *f2>, <*f1, *nv>, <*f2, *p>, <*p, *nv>\}$$

alias-bind`(call, entry,` *pointer-type*`)` : This function calculates the *alias effects* of *pointer-type* present at `call` which fall into the following two categories:

1. aliases between appropriate dereferences of two formals : At $call_{c2}$, actual r is passed to formal $bar1$ and $r{\rightarrow}d1$ is passed to formal $bar2$. Given that *pointer-type* is $<r{\rightarrow}d1 \Rightarrow B>$, the appropriate dereferences of $bar1$ and $bar2$ form aliases comprising the members of class B. Thus

$$\{<*bar1{\rightarrow}d1, *bar2>, <bar1{\rightarrow}d1{\rightarrow}b1, bar2{\rightarrow}b1>\}$$

forms a subset of **alias-bind**($call_{c2}, entry_{C::bar}, <r{\rightarrow}d1 \Rightarrow B>$).

2. aliases between the dereference of an actual and the corresponding formal: Given that *pointer-type* is $<r{\rightarrow}d1 \Rightarrow B>$ and $r{\rightarrow}d1$ is passed to formal $bar2$ at $call_{c2}$,

$$\{<*nv{\rightarrow}d1, *bar2>, <nv{\rightarrow}d1{\rightarrow}b1, bar2{\rightarrow}b1>\}$$

also forms a subset of **alias-bind**($call_{c2}, entry_{C::bar}, <r{\rightarrow}d1 \Rightarrow B>$).

The pair at the call site contains information about which object name was mapped to nv in the called function by **bind** or **alias-bind**. For example,

$$<*p, *nv> \in \textbf{bind}(call_{c1}, entry_{C::foo}, <*p, *q>)$$

where q is mapped to nv. While propagating an alias from the called to the calling function (Section 5.4.5), we will use this association to map the object name involving nv back to the object name local to the calling function.

type-bind(`call, entry,` *pointer-type*) : This function calculates the *type effects* of *pointer-type* present at `call` on `entry`. Depending on the actual-formal bindings at `call`, *pointer-type* may simply propagate to `entry` and/or may create a pointer-type pair involving the corresponding formal. In Figure 5.7,

$$\textbf{type-bind}(call_{c2}, entry_{C::bar}, <s \Rightarrow C>) = \{<s \Rightarrow C>, <this \Rightarrow C>\}$$

Unlike **bind** and **alias-bind**, which may impose aliases involving nv at the entry node, **type-bind** does not impose any pointer-type pairs which involve nv. As we will see in Section 5.4.4, intraprocedural propagation uses a pointer-type pair $<p \Rightarrow C>$ at a pointer assignment "`lhs = rhs`" only if p is a dereference of *rhs*. Since an object name containing nv cannot explicitly appear in a pointer assignment, pointer-type pairs containing nv are irrelevant as entry assumptions in the called function. For example in Figure 5.7, although $<q \Rightarrow int>$ holds at $call_{c1}$,

$$<nv \Rightarrow int> \notin \textbf{type-bind}(call_{c1}, entry_{C::foo}, <q \Rightarrow int>)$$

Note that all the above binding functions iterate over the parameter list at most twice, in particular when two actual parameters are aliased. If the number of parameters for any function in the program is assumed to be bounded by a constant, **bind**, **alias-bind** and **type-bind** have $\mathcal{O}(1)$ complexity.

5.4.3 Initialization and Introduction Phases

The algorithm starts by lazily initializing all the *points-to-type* and *may-hold* predicates to *false*; this enables us to perform initialization of all the predicates in constant time [LR92]. We also initialize the worklist to empty. The intraprocedural aspects of the introduction phase are summarized in Figure 5.8. This introduces pointer-type and alias pairs generated locally at a pointer assignment ICFG node. Note that we only can deal with access paths without dereferences during the introduction phase, because creating the corresponding object name for an access path with dereferences requires knowledge of the type being dereferenced (recall, our analysis restricts itself to generating object names with valid dereferences). This information only becomes available during the propagation phase. Nevertheless, the type of `&r` is uniquely known using symbol table information, which can be utilized in introducing *points-to-type* predicates.

The function **inter-alias-type-introduction(**$call$**,**$entry$**)** has the following tasks:

1. For each AP in **bind0**($call$, $entry$),

$$\textbf{make-true}(\textit{may-hold}(entry, AP, AP))$$

2. for each PT in **type-bind0**($call$, $entry$),

$$\textbf{make-true}(\textit{points-to-type}(entry, PT, PT))$$

```
for each node n in the ICFG
        If n is
              n :  p = new T;
```
$\textbf{make-true}(\textit{points-to-type}(n,\ \emptyset,\ <p \Rightarrow T>))$

$\textbf{make-true}(\textit{may-hold}(n,\ \emptyset,\ <p{\rightarrow}mem_k, new_n.mem_k>))^{\dagger}$

```
              n :  p = &r;
```
$\textbf{make-true}(\textit{points-to-type}(n,\ \emptyset,\ <p \Rightarrow type(r)>))$

if r is a fixed-location

$\textbf{make-true}(\textit{may-hold}(n, \emptyset, <p{\rightarrow}mem_k, r.mem_k>))$

†We use $<p{\rightarrow}mem_k, new_n.mem_k>$ to denote all aliases involving corresponding members. If p is not a class pointer, we denote the resulting alias as $<*p, new_n>$.

Note: p represents a fixed-location.

Figure 5.8: Intraprocedural introduction phase : **intra-alias-type-introduction**

5.4.4 Intraprocedural Propagation

Propagation of information is simple through an intraprocedural successor which is not a pointer assignment; such a node can neither create nor destroy aliases or pointer-types. In this case, a predicate *points-to-type*(m, $AAPT$, PT) propagates through the successor n with **make-true**(*points-to-type*(n, $AAPT$, PT)). Similarly a predicate *may-hold*(m, $AAPT$, AP) propagates through n with **make-true**(*may-hold*(n, $AAPT$, AP)). Therefore, we concentrate on propagation through pointer assignment nodes, calculating the semantic effects (i.e., transfer functions) of the code at these nodes on the information reaching from an ICFG predecessor. In Table 5.1, we categorize various interactions between *may-hold* and *points-to-type* as they propagate through a pointer assignment. As many as two incoming predicates may be needed to infer one predicate at a successor. In the worklist algorithm, they become candidates for propagation in an indeterminate order. The one which reaches last determines which intraprocedural propagation functions from Figure 5.6 (p. 72) is utilized. In Figures 5.9, 5.10 and 5.11, we list the two incoming predicates (i1) and (i2) from a predecessor m to a pointer assignment "`n : lhs = rhs`". We then list the resulting predicates at node n. When two predicates imply a single predicate at the successor, we may use

result	*points-to-type* [and *may-hold*]	*may-hold* [and *may-hold*]
points-to-type	Figure 5.9 Details on p. 80	-
may-hold	Figure 5.10 Details on p. 81	Figure 5.11 Details on p. 82

Table 5.1: Intraprocedural interaction between *may-hold* and *points-to-type*

either of the two assumptions from the incoming predicates, as our approximation formulation accommodates at most one assumption. We prefer the one which is non-$\emptyset$ whenever possible[31]. We denote the resulting choice from *AAPT1* and *AAPT2* as $(AAPT1 \oplus AAPT2)$. Note that aliases alone can never create a pointer-type pair without using another pointer-type pair, which accounts for the blank entry in Table 5.1. Whenever we use two predicates at a node m to derive a predicate at the successor n, we are making a safe, conservative decision, as the two predicates at m need not be *true* on the same consistent path to m. Nevertheless, in the absence of precise path specific information, we must conservatively assume that they are. For example, suppose $<*u, p>$ and $<q \Rightarrow int>$ hold at a node m which is a predecessor of the pointer assignment "n : p = q". Although $<*p, *q>$ clearly holds at n solely on account of $<q \Rightarrow int>$ at m, the transitive alias $<*q, **u>$ should not hold at n unless the two incoming aliases hold as result of execution of the *same* consistent path to m. The following intraprocedural propagation functions account for this possibility by safely creating transitive aliases and pointer-type pairs at the expense of precision.

points-to-type **and** *may-hold* **implying** *points-to-type***:** Figure 5.9 shows the most general case of this category of propagation. If a dereference (if any) of the right hand side (*rhs*) points to an object of class C, then the corresponding dereference of the left hand side (*lhs*) will point to the same object, resulting in (o1). Also, the corresponding dereference of any alias of the *lhs* will also point to the same object, resulting in (o2). The incoming predicate (i1) simply propagates to the successor n as (o3). If (i1)

[31]Unfortunately such a simple choice is not possible when either or both assumptions contain an object name with *nv*; we leave further details for Appendix B.2.

(i1)
points-to-type(m, AAPT1, <rhs[deref_seq] ⇒ C>)

(i2)
may-hold(m, AAPT2, <lhs,lhs_alias>)

```
n : lhs = rhs;
```

(o1)
points-to-type(n, AAPT1 ⊕ AAPT2, <lhs[deref_seq] ⇒ C>)

(o2)
points-to-type(n, AAPT1 ⊕ AAPT2, <lhs_alias[deref_seq] ⇒ C>)

(o3)
points-to-type(n, AAPT1, rhs[deref_seq] ⇒ C>)

Figure 5.9: *points-to-type* using *may-hold* to imply *points-to-type*

becomes a candidate for propagation later than (i2), then **type-implies-type-thru-assign** performs the above actions. On the other hand, if (i2) comes up for propagation later than (i1), **alias-implies-type-thru-assign** performs the same task.

Special cases

- If `lhs` has no dereferences: In this case, we do not need (i2) to obtain the corresponding object name from the access path `lhs` to derive (o1); we can use the fixed-location *lhs*. (i1) alone propagates to (o1) using **type-implies-type-thru-assign**. Since (i2) is not used, $AAPT2$ is $\emptyset$ in (o1).

- If `lhs` is a prefix of `rhs`, clearly (i1) is blocked by the assignment and will not result in (o3).

- *points-to-type*$(m, AAPT, <p \Rightarrow C>)$, where p cannot be obtained by applying a (perhaps empty) dereference sequence to *lhs*, propagates directly to n. Clearly, the assignment cannot break the pointer-type pair in this case.

***points-to-type* and *may-hold* implying *may-hold*:** Figure 5.10 shows the general case of this category of propagation. If a dereference (if any) of the right hand side (*rhs*) points to an object of class C, then the corresponding dereference of the left hand side (*lhs*) will point to the same object. In other words, corresponding dereferences of

(i1)
points-to-type(m, AAPT1, <rhs[deref_seq] ⇒ C>)

(i2)
may-hold(m, AAPT2, <lhs,lhs_alias>)

```
n : lhs = rhs;
```

(o1)
$may\text{-}hold(n, AAPT1 \oplus AAPT2, <lhs[deref\_seq] \xrightarrow{C} mem_k, rhs[deref\_seq] \xrightarrow{C} mem_k>)$

(o2)
$may\text{-}hold(n, AAPT1 \oplus AAPT2, <lhs\_alias[deref\_seq] \xrightarrow{C} mem_k, rhs[deref\_seq] \xrightarrow{C} mem_k>)$

(o3)
$may\text{-}hold(n, AAPT1 \oplus AAPT2, <lhs[deref\_seq] \xrightarrow{C} mem_k, lhs\_alias[deref\_seq] \xrightarrow{C} mem_k>)$

Figure 5.10: *points-to-type* using *may-hold* to imply *may-hold*

these two object names will be aliased, resulting in (o1). Similarly, the corresponding dereference of any alias of the *lhs* will also point to the same object, resulting in (o2). Since *lhs* and *lhs_alias* will point to the same object after the assignment, we have (o3). If (i1) becomes a candidate for propagation later than (i2), then **type-implies-alias-thru-assign** performs the above actions. On the other hand, if (i2) comes up for propagation later than (i1), **alias-implies-alias-thru-assign** performs the same task.

Special cases

- If `lhs` has no dereferences: In this case, we do not need (i2) to obtain the corresponding object name from the access path `lhs` to derive (o1); we can use the fixed-location *lhs*. If `lhs` is not a prefix of `rhs`, (i1) alone propagates to (o1) using **type-implies-alias-thru-assign**. Since (i2) is not used, *AAPT2* is $\emptyset$ in (o1).
- If `lhs` is a prefix of `rhs`, (i1) does not propagate to (o1) or (o2).

Note that both special cases always require (i1). Whether or not (i2) is needed depends on the particular case.

may-hold **and** *may-hold* **implying** *may-hold***:** If a strict dereference of the right hand side (*rhs*) is aliased to another object name (*objname*), the corresponding dereference

(i1)
may-hold(m, AAPT1, <rhs{deref_seq}, objname>)

(i2)
may-hold(m, AAPT2, <lhs, lhs_alias>)

```
n : lhs = rhs;
```

(o1)
may-hold(n, AAPT1 $\oplus$ *AAPT2, <lhs{deref_seq}, objname>)*

(o2)
may-hold(n, AAPT1 $\oplus$ *AAPT2, <lhs_alias{deref_seq}, objname>)*

Figure 5.11: *may-hold* using *may-hold* to imply *may-hold*

of the left hand side (*lhs*) will be aliased to *objname* after the assignment. The corresponding dereference of any object names aliased to *lhs* will also be aliased to *objname*. Figure 5.11 captures this category of intraprocedural propagation. This task is performed by **alias-implies-alias-thru-assign**.

Special cases

- `lhs` has no dereferences; only (i1) is sufficient to derive (o1). We consider *AAPT2* as $\emptyset$ in (o1).

- *may-hold*(m, $AAPT$, $<p, q>$), where neither p nor q can be obtained by applying a dereference sequence to *lhs*, propagates directly to n. Clearly, the assignment cannot break the alias in this case.

5.4.5 Interprocedural Propagation

For ease of description, we need to define another auxiliary function to encapsulate the various steps involved in propagating a predicate from an exit node to a return node. **make-true-at-return**(*mp*(*return*, $AAPT$, APT), APT_c), where *mp* stands for either *may-hold* or *points-to-type*, does the following:

```
if APT contains an object name involving a local variable
       do nothing // variable goes out of scope on exiting the function
elseif APT contains an object name involving nv
```

```
APT₁ = APT with the corresponding original object name
          substituted for object name involving nv, using APT_c
    make-true(mp(return, AAPT, APT₁))
else
    make-true(mp(return, AAPT, APT))
```

In the above function, we assume that APT contains at most one object name which involves *nv*. There are circumstances which create an alias between two object names containing *nv*, which require special treatment. We will not consider those circumstances here, and leave them for Appendix B.2.

Propagation from *call* node

For non-virtual function calls, the corresponding entry node is easily determined. However if *call* represents a virtual call site, the *points-to-type* predicates involving the receiver at *call* determine the possible functions invoked (Section 5.3.2, p. 63). Each class associated with the receiver corresponds to a virtual function. For each *entry* so determined, we propagate information to *entry* and the corresponding *return*, as outlined below.

alias-implies-from-call($call, AAPT, alias$) : The aliases imposed at *entry* due to the presence of *alias* at *call* are available through the set **bind**($call, entry, alias$). Each alias pair AA in this set causes **make-true**(*may-hold*(*entry*, AA, AA)). In some previous iteration of the worklist algorithm, the presence of AA at *entry* (imposed by another call) may have already made some *may-hold* predicates *true* at the *exit* of the function. Let a representative predicate be *may-hold*(*exit*, AA, APT). Associating this information at *call* and *exit*, we **make-true-at-return**(*may-hold*(*return*, $AAPT$, APT), *alias*). Similarly, for each *points-to-type*(*exit*, AA, APT), **make-true-at-return**(*points-to-type*(*return*, $AAPT$, APT), *alias*).

If both the object names in *alias* are non-visible in the called function, the alias cannot be destroyed by the call. We propagate the predicate directly to the return node with **make-true**(*may-hold*(*return*, $AAPT$, *alias*)).

type-implies-from-call(*call*,$AAPT$,*pointer-type*): This function accounts for the special case of type propagation when *pointer-type* involves the receiver of a virtual *call*. When a receiver pointing to an instance of a class is a candidate for propagation, it implies a new function invocable from *call*. Since the introduction phase cannot perform type and alias introduction for virtual function invocations, **alias-intros-by-call(call,entry)** and **types-intro-by-call(call,entry)** are performed at this stage for *entry* node corresponding to the receiver type. Also, all the information (in the form of *points-to-type* and *may-hold*) already existing at *call* is propagated to the new *entry*.

Once the *entry* is so determined, the routine task of propagation continues as follows. Let *pointer-type* be $<p \Rightarrow B>$ without loss of generality.

The set of pointer-type pairs created by $<p \Rightarrow B>$ at *entry* is available as **type-bind**($call, entry, <p \Rightarrow B>$). Each $PT \in$ **type-bind**($call, entry, <p \Rightarrow B>$) causes **make-true**(*may-hold*($entry$, PT, PT)). If some pair PT from this set has already created a *true* predicate of the form *may-hold*($exit$, PT, APT), we **make-true-at-return**(*may-hold*($return$, $AAPT$, APT), $<p \Rightarrow B>$). Similarly, if some pair PT has already created a *true* predicate of the form *points-to-type*($exit$, PT, APT), we **make-true-at-return**(*points-to-type*($return$, $AAPT$, APT), $<p \Rightarrow B>$).

Each $AA \in$ **alias-bind**($call, entry, <p \Rightarrow B>$) causes **make-true**(*may-hold*($entry$, AA, AA)). Some pairs from this set may have already yielded a *true* valued predicate *may-hold*($exit$, AA, APT) at the exit of called function, which results in **make-true-at-return**(*may-hold*($return$, $AAPT$, APT), $<p \Rightarrow B>$). Similarly, if some pair AA has already created a *true* predicate of the form *points-to-type*($exit$, AA, APT), we **make-true-at-return**(*points-to-type*($return$, $AAPT$, APT), $<p \Rightarrow B>$).

If p is non-visible in the called function, the call has no effect on $<p \Rightarrow B>$. Therefore, we also propagate the predicate directly to the return node using **make-true**(*points-to-type*($return, AAPT, <p \Rightarrow B>$)).

Propagation from *exit* node

Suppose a pair APT holds at *exit* with assumption $AAPT$ at *entry*. Using the parameter binding functions described in Section 5.4.2 we determine the *call*(s) responsible for imposing $AAPT$ at *entry*, and propagate APT only to the corresponding *return*(s). This pivotal role played by the binding functions allows us to propagate information along a good approximation of consistent paths.

alias-implies-alias-from-exit(*exit*,$AAPT$,*alias*):

1. If the entry assumption $AAPT$ is $\emptyset$, *alias* may hold at *exit* no matter which *call* invokes the function containing *exit*, as this alias pair is created solely due to the execution of the function and not due to the information reaching *entry* from a call site. As a result, we **make-true-at-return**(*may-hold*(*return*, $\emptyset$, *alias*), $\emptyset$) for all *returns* corresponding to virtual or non-virtual *calls* invoking this function.

2. If $AAPT$ is non-$\emptyset$, it implies that *alias* holds at *exit* if a *call* imposes $AAPT$ at *entry*. Suppose *may-hold*(*call*, $AAPT'$, AP) imposes $AAPT$ at *entry* through **bind**(*call*, *entry*, AP). Using this association, we propagate *alias* to *return* with **make-true-at-return** (*may-hold*(*return*, $AAPT'$, *alias*), AP). Similarly, for each *points-to-type*(*call*, $AAPT'$, PT) imposing $AAPT$ at *entry* through either **type-bind**(*call*, *entry*, PT) or **alias-bind**(*call*, *entry*, PT), we **make-true-at-return**(*may-hold*(*return*, $AAPT'$, *alias*), PT).

type-implies-type-from-exit(*exit*,$AAPT$,*pointer-type*):

1. If the entry assumption $AAPT$ is $\emptyset$, *pointer-type* may hold at *exit* of a function regardless of which call invokes it. This pointer-type pair is created solely due to the execution of the function, and not due to the information reaching *entry*. As a result, we **make-true-at-return**(*points-to-type*(*return*, $\emptyset$, *pointer-type*), $\emptyset$) for all *return* nodes corresponding to *call* nodes invoking this function.

2. If $AAPT$ is non-$\emptyset$, it implies that *pointer-type* holds at *exit* if a call site imposes $AAPT$ at *entry*. Suppose *may-hold*(*call*, $AAPT'$, AP) imposes $AAPT$ at *entry* through **bind**(*call*, *entry*, AP). Using this association at *call*, we **make-true-at-return**(*points-to-type*(*return*, $AAPT'$, *pointer-type*), AP). Similarly, suppose

```
class A {        class B : public A {
public:             ...
 int i,j,k;      };
 ...
 A *next;
} *p, *q;
A u, v, w;
```

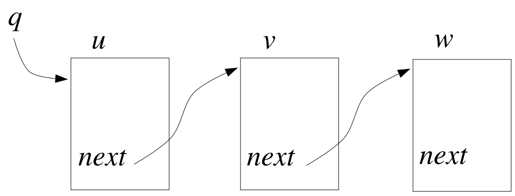


Figure 5.12: Example class hierarchy and storage layout

points-to-type(*call*, $AAPT'$, PT) imposes $AAPT$ at *entry* through either **type-bind**(*call*, *entry*, PT) or **alias-bind**(*call*, *entry*, PT). Here, we **make-true-at-return**(*points-to-type*(*return*, $AAPT'$, *pointer-type*), PT).

5.4.6 Analysis in the Presence of Recursive Data Structures

Recursive structures give rise to potentially infinite object names. For example, using the declarations in Figure 5.12, arbitrarily many object names (represented by the regular expression $p(\rightarrow next)^*$) are possible. To reduce the number and length of object names (and therefore, of aliases and pointer-type pairs) to a finite number, we use the notion of *k-limiting*, similar to that introduced by Jones and Muchnick [JM79]. Intuitively, k-limiting implies that up to k explicit dereferences are maintained in an object name and further dereferences are abstracted into a special # dereference.

For example, we represent $q \rightarrow next \rightarrow next \rightarrow next \rightarrow i$ by $q \rightarrow next \rightarrow next\#$, where the value of k is 2. Note that k-limiting involves loss of information due to a combination of the following basic approximations in object names.

1. *last-member approximation*: A single k-limited object name may represent distinct object names with different last dereferences. For example in Figure 5.12, 1-limited object name $q \rightarrow next\#$ represents both $q \rightarrow next \rightarrow i$ and $q \rightarrow next \rightarrow j$.

2. *dereference approximation*: A single k-limited object name may represent object names with distinct numbers of dereferences. For example in Figure 5.12, a single 1-limited object name $q \rightarrow next\#$ represents both $q \rightarrow next \rightarrow next$ and $q \rightarrow next \rightarrow next \rightarrow next$.

source code	$k = 2$	$k = 1$
`m:   p = q`	$<p \to next \to i, q \to next \to i>$ $<p \to next \to j, q \to next \to j>$ $<p \to next \to k, q \to next \to k>$	$<p \to next\#, q \to next\#>$

Table 5.2: smaller value of k resulting in fewer aliases

source code	$k = 2$	$k = 1$
`m:   p→next = q`	$<p \to next \to next \Rightarrow A>$ $<p \to next \to next\# \Rightarrow A>$	$<p \to next\# \Rightarrow A>$

Table 5.3: smaller value of k resulting in fewer pointer-type pairs

The approximations in object names due to k-limiting have two opposite effects on the number of alias and pointer-type pairs. We use the example class declaration and an initial storage layout in Figure 5.12 to illustrate these effects. In Tables 5.2 and 5.3, we show how the number of aliases and pointer-type pairs *decreases* for a smaller value of k. The first column shows a pointer assignment at m, and the remaining columns list for two values of k, the relevant pairs resulting from this assignment. Due to the last-member approximation, multiple alias pairs map to a single alias pair for $k = 1$ reducing the number of aliases. Also with $k = 1$, the dereference approximation maps two distinct pointer-type pairs to $<p \to next\# \Rightarrow A>$.

Assuming the same initial storage layout, we show how the number of alias pairs at a program point *increases* as a result of selecting a lower value of k. In the first row of Table 5.4, we list relevant alias pairs at some predecessor l of node m. For $k = 2$, we can distinguish the object names aliased to $v.next$ and $w.next$. However for $k = 1$, owing to the dereference approximation, $q \to next\#$ represents both these object names. In the second row, we propagate these aliases through the pointer assignment at m. Since the information is lost as to how many dereferences are abstracted by # in $q \to next\#$, spurious aliases (marked with †) are generated for $k = 1$, leading to an increase in the number of aliases[32]. Suppose the pointer assignment at m is followed by

[32]In our prototype implementation, we use @ to abstract a single k-limited dereference and # for more than one dereference. As a result, our implementation can distinguish between $q \to next@$ and

source code	$k = 2$	$k = 1$
`l:...`	$<q{\rightarrow}next{\rightarrow}next, v.next>$ $<q{\rightarrow}next{\rightarrow}next\#, w.next>$	$<q{\rightarrow}next\#, v.next>$ $<q{\rightarrow}next\#, w.next>$
`m:  p = q→next`	$<p{\rightarrow}next, v.next>$ $<p{\rightarrow}next{\rightarrow}next, w.next>$	$<p{\rightarrow}next, v.next>$ $<p{\rightarrow}next\#, v.next>^{\dagger}$ $<p{\rightarrow}next, w.next>^{\dagger}$ $<p{\rightarrow}next\#, w.next>$

†Spurious aliases

Table 5.4: smaller value of k resulting in spurious aliases

"`n:  p→next = new B`". For $k = 1$, the presence of spurious alias $<p{\rightarrow}next, w.next>$ creates a spurious pointer-type pair $<w.next \Rightarrow B>$ at node `n`, increasing the number of pointer-type pairs created.

In the remainder of this section, we provide an intuitive overview of type determination and aliasing in the presence of k-limited object names through examples, assuming that the value of k is 2.

Intraprocedural analysis in the presence of k-limiting

First, we show how k-limited pointer-type pairs and aliases are initially created from those which are not k-limited. In general, if any of the cases during intraprocedural introduction and propagation (from Sections 5.4.3 and 5.4.4) create an object name with $> k$ dereferences using those with $\leq k$ dereferences, that object name would be replaced by the corresponding k-limited object name.

Initial creation of a k-limited pointer-type pair: $<p{\rightarrow}g{\rightarrow}h \Rightarrow C>$ propagating through a pointer assignment "`n :  q→f = p;`" creates $<q{\rightarrow}f{\rightarrow}g\# \Rightarrow C>$ at `n`.

Initial creation of a k-limited alias: $<p{\rightarrow}g{\rightarrow}h \Rightarrow C>$ propagating through a pointer assignment "`n :  q→f = p;`" creates the alias $<q{\rightarrow}f{\rightarrow}g\#, p{\rightarrow}g{\rightarrow}h\#>$.

Now, we show how to propagate k-limited pointer-type pairs and aliases through

---

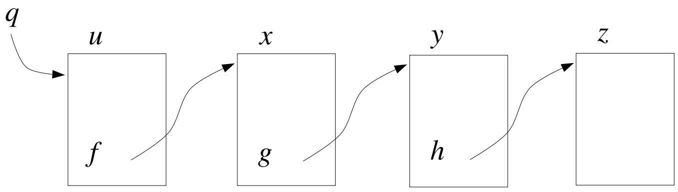
$q{\rightarrow}next\#$. However the example can be generalized using an additional level of dereferencing, so that spurious aliases are generated.

pointer assignments. The following examples describe how we recover the explicit dereferences abstracted into $\#$. Once the dereferences are recovered, intraprocedural propagation is similar to that summarized in Table 5.1 (p. 80), except that each object name created with $> k$ dereferences would be replaced by the corresponding k-limited object name.

Propagating a k-limited pointer-type pair: A k-limited pointer-type pair is shown below as a candidate for propagation through a pointer assignment. One possible storage layout for this scenario appears alongside.

$<q \rightarrow f \rightarrow g\# \Rightarrow C>$

```
n:  p = q→f;
```



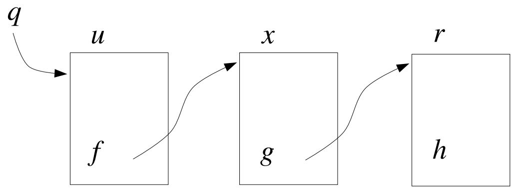
Since the right hand side (rhs) of the assignment is more dereferenced than the left hand side (lhs), a dereference sequence of the rhs may be k-limited while the corresponding dereference sequence of the lhs may not. Since the difference in dereferences between lhs and rhs is 1, we need to recover one explicit dereference that may be abstracted by $q \rightarrow f \rightarrow g\#$. For example, if the object name were to represent $q \rightarrow f \rightarrow g \rightarrow h$, using Theorem 5.2.1 (see p. 57) there must exist an alias $<q \rightarrow f \rightarrow g \rightarrow h, (u.f) \rightarrow g \rightarrow h>$. With k-limiting, it would appear as $<q \rightarrow f \rightarrow g\#, (u.f) \rightarrow g \rightarrow h>$. We use this alias to recover the required explicit dereference, *viz.* h, to obtain $<q \rightarrow f \rightarrow g \rightarrow h \Rightarrow C>$. It is now straightforward to infer $<p \rightarrow g \rightarrow h \Rightarrow C>$ at n. Propagation of $<q \rightarrow f \rightarrow g \rightarrow h \Rightarrow C>$ also results in aliases $<p \rightarrow g \rightarrow h \rightarrow mem_i, q \rightarrow f \rightarrow g \rightarrow h \rightarrow mem_i>$ for each member mem_i of class C. These aliases are represented as $<p \rightarrow g \rightarrow h\#, q \rightarrow f \rightarrow g\#>$ for each distinct type corresponding to member_type(mem_i)[33].

Propagating a k-limited alias: Suppose an alias involving k-limited object name as shown below is a candidate for propagation through pointer assignment n. Again, we show one possible storage layout corresponding to the program segment.

[33]Recall that each object name has an associated type (from Figure 2.1, p. 12). For notational convenience we normally omit it. However, $p \rightarrow g \rightarrow h\#$ representing $p \rightarrow g \rightarrow h \rightarrow mem_i$ is distinct from $p \rightarrow g \rightarrow h\#$ representing $p \rightarrow g \rightarrow h \rightarrow mem_j$ if members mem_i and mem_j have distinct types.

$<q\to f\to g\#, r.h>$

`n:   p = q->f;`



If the object name $q\to f\to g\#$ in the alias were to represent $q\to f\to g\to h$, it must be the case that we have the following aliases:

1. $<q\to f\to g\to h, (u.f)\to g\to h>$ for some u : using Theorem 5.2.1 (p. 57).
2. $<(u.f)\to g\to h, r.h>$: since $q\to f\to g\to h$ is given as aliased to $r.h$.

With k-limiting, the above aliases would appear as $<q\to f\to g\#, (u.f)\to g\to h>$ and $<(u.f)\to g\to h, r.h>$[34]. The existence of these aliases serves as evidence of the fact that the explicit dereference h is abstracted as $\#$ in $q\to f\to g\to h$. Having inferred this, it is easy to derive that $<p\to g\to h, r\to h>$ holds at n.

Interprocedural analysis in the presence of k-limiting

The dereference recovery at a call site is similar to that described above, where rhs is the actual parameter and lhs is the corresponding formal parameter. We use appropriate alias or pointer-type pairs at the call site to recover the required explicit dereferences and proceed with parameter bindings as in Section 5.7.

---

[34] For efficiency, we do not explicitly check for this alias from Rule 2, but safely and approximately assume that it exists.

Chapter 6

Theoretical and Empirical Results

We have shown that the problem of type determination and aliasing for C^{++} with single level pointers, and hence for general purpose pointer usage, is NP-hard. We have presented an algorithm to solve the above problem safely and approximately. In this chapter we derive the theoretical worst case polynomial complexity of the algorithm. In the absence of a good mechanism for calculating usages in an *average* C^{++} program, a theoretical average case analysis would be impossible. Therefore, we resorted to empirical analysis of algorithm performance using a prototype implementation. In this chapter, we show that our algorithm is linear in the size of the solution, with empirical corroboration of our claim using a modest test suite of C^{++} programs. We show that the direct application of our type determination solution yields highly effective results for virtual function resolution. Nevertheless, definite claims about the precision or effectiveness of aliasing solution are beyond the purview of this thesis, as we have not yet used the aliasing solution in an application to gauge its utility.

6.1 Worst Case Complexity of Algorithm

The lazy approach for calculation makes it difficult to obtain a tight bound on the theoretical complexity of our algorithm. However we can show that the worst case complexity of our algorithm is polynomial in the number of ICFG nodes. We define the following sets:

- $\mathcal{N}$ = set of ICFG nodes.
- $\mathcal{O}$ = set of object names.
- $\mathcal{T}$ = set of types.

We make the following reasonable assumptions:

1. The maximum number of parameters for a function is bounded by a constant over the entire program [CK89]. Given this assumption the parameter binding functions are constant time (Section 5.4.2).

2. The value of k, the maximum number of dereferences maintained explicitly by our algorithm, is a small constant. The maximum k used by our algorithm is 5.

3. The data structures and encodings used by the algorithm provide constant time functions for initialization, existence test and insertion for any predicate (using lazy initialization and hashing).

4. All alias and pointer-type pairs relevant to the algorithm at an ICFG node can be obtained in time linear with respect to the number of these relevant pairs. For example, with each assignment node of the form "`lhs = rhs`", we explicitly maintain a list of all *points-to-type* and *may-hold* predicates which contain pairs involving dereferences of `rhs`.

5. The number of types used in the program is bounded above by $O(|\mathcal{N}|)$. In other words, $|\mathcal{T}| = O(|\mathcal{N}|)$.

6. The number of fixed-locations appearing in the program body[35] is bounded above by $O(|\mathcal{N}|)$.

7. The maximum number of data members of a class (including inherited members and the members of any class object contained within the class) is bounded above by $O(|\mathcal{N}|)$.

8. For realistic programs, the maximum number of different implementations of a virtual function in a hierarchy is bounded by a constant.

Let fl be a fixed-location. Then the object names derived from fl can be expressed using a regular expression $fl(\stackrel{type}{\rightarrow}\ member)^*$. Since we use only up to k dereferences,

---

[35]We do not consider the declarations to be part of the program body.

clearly the number of object names is polynomial in terms of the number of ICFG nodes. Since a pointer-type is represented as <*object_name* $\Rightarrow$ *type*> we have a polynomial number of pointer-type pairs. Similarly, an alias is a pair of object names, implying that the number of possible aliases is polynomial. Given that each *may-hold* is a tuple of the form (*node, alias/pointer-type, alias*) and each *points-to-type* is a tuple of the form (*node, alias/pointer-type, pointer-type*), the total number of possible predicates is polynomial in $|\mathcal{N}|$.

As we saw in Section 5.4, each predicate is either introduced (in constant time) at a node or created as a result of propagation from the predicates present at the predecessors. Each propagation function uses a fixed number of predicates at a time to infer a single predicate. The worst case occurs while inferring a predicate at the *return* node of a virtual call site where 3 predicates, *viz. points-to-type* involving the receiver and a predicate each from the call and the exit node, yield a single predicate at the return node. For example, the predicates *points-to-type*($call$, $AAPT$, $<rec \Rightarrow C>$), *points-to-type*($call$, $AAPT1$, $PT1$) and *points-to-type*($exit$, $PT2$, $PT3$) contribute to **make-true-at-return**(*points-to-type*($return$, $AAPT$, $PT3$)), where $PT2 \in$ **type-bind** (*call,entry,PT1*). However, there are multiple ways the same predicate may be inferred at a node. If there are n true-valued predicates present at the predecessors, there are C_3^n ways in which a predicate may be inferred in the worst case, which is polynomial. As a result, the worst case complexity of the entire algorithm is polynomial in $|\mathcal{N}|$.

6.2 Algorithm Performance in Practice

The above worst case complexity does not provide any insight into the practical running complexity of the algorithm. We have designed the algorithm in such a way that the work is performed only for the predicates which become *true*. A better criterion would be to express the performance of the algorithm in terms of the size of *may-hold* and *points-to-type* solution it generates[36]. Using the assumptions in Section 6.1, a constant amount of work is required to make a predicate *true*, except when the same predicate

[36]Since the design of our algorithm is similar to Landi's pointer aliasing analysis for C, the reasoning is similar to that in [Lan92a].

is created in multiple ways. If n is neither a return node nor a pointer assignment, the number of ways a predicate can become *true* is O(number of ICFG predecessors of n). Since the number of intraprocedural edges in the ICFG is of the order of the number of nodes, in an amortized sense the work done per predicate is constant time. Nevertheless, we need a more careful counting argument while dealing with propagation involving return nodes and pointer assignment nodes.

Propagation from a return node Let n be a return node and *call* be the corresponding call node. Suppose we want to **make-true-at-return**(*may-hold*(n, $AAPT$, APT)).

1. If *call* represents a virtual call site, *may-hold*$(n, AAPT, AP)$ may be a result of propagation from multiple exit nodes, depending on the number of functions invoked by the call. Given our assumption that the maximum number of different implementations of a virtual function in a hierarchy is bounded by a constant, the number of distinct exit nodes associated with n is constant.

2. For each exit node *exit* with corresponding entry node *entry*,

   may-hold$(n, AAPT, AP)$ =

   $$\left(\textit{may-hold}(\textit{exit}, \emptyset, AP) \bigvee_{AAPT2} \left(\begin{array}{l} \textit{may-hold}(\textit{exit}, AAPT2, AP) \wedge \\ \textit{may-hold}(\textit{call}, AAPT, AP_c) \end{array} \right) \right)$$

   where $AAPT2 \in$ **bind**(*call,entry,*AP_c). A similar argument applies to inferring a *points-to-type* at n. Clearly, the number of different ways *may-hold*$(n, AAPT, AP)$ can become *true* depends on (1 + number of distinct $AAPT2$ for AP to hold at *exit*). Unfortunately, the number of distinct predicates which are identical except in their assumption cannot be theoretically bounded by a constant. As a conservative decision, we arbitrarily allowed maximum 5 distinct assumptions for predicates which were otherwise identical and replaced any further assumptions with $\emptyset$[37]. Given this, the number of distinct ways a predicate can become *true* is

[37]We could not replace an $AAPT$ with $\emptyset$ in *may-hold*(n, $AAPT$, AP) or *points-to-type*(n, $AAPT$, PT) if AP or PT contained an object name involving nv. In our implementation, the assumption stores

now a constant.

Propagation through a pointer assignment As we saw in Section 5.4.4, many predicates at an ICFG node may interact (two at a time) to create a new predicate at a pointer assignment successor. For example, suppose for an ICFG node m with pointer assignment successor n, a single predicate *may-hold*$(m,\ AAPT1,\ AP)$ interacts with multiple predicates, identical except in their assumption, to create *may-hold*$(n,\ AAPT1,$ $<a,b>)$. This will happen if *AAPT1* is non-$\emptyset$ and the operation $(AAPT1 \oplus AAPT2)$ favors *AAPT1* over *AAPT2* for any *AAPT2*. Then,

$$\textit{may-hold}(n, AAPT1, <a,b>) = \\ \bigvee_{AAPT2} \left(\textit{points-to-type}(m, AAPT1, PT) \wedge \textit{may-hold}(m, AAPT2, AP) \right)$$

The same decision to allow maximum 5 distinct non-$\emptyset$ assumptions would enable the algorithm to infer the resulting predicate in $O(1)$ time.

Since each *true* predicate is created at a node using constant time work and then remains *true*, the algorithm runs in time linear with respect to the size of *may-hold* and *points-to-type* solution for realistic programs.

6.3 Prototype Implementation

The results presented in the remainder of this chapter represent our efforts to empirically demonstrate the contributions of the algorithm and assess its practicality using a prototype implementation. The prototype is written in C and runs on a Sun SPARC-20 with 64MB main memory. We are using the MasterCraft C^{++} system of Tata Consultancy Services as the front end C^{++} parser for the implementation. Our aliasing and type determination algorithm reuses some code from Landi's aliasing algorithm [Lan92a] with suitable modifications. We present empirical results of analyzing 19 C^{++} programs obtained from various (publicly available) sources such as textbooks, demonstration programs accompanying a C^{++} compiler and undergraduate projects.

crucial information to retrieve the corresponding non-visible fixed-location from *nv*. Fortunately, the number of such distinct assumptions did not exceed 6 in our test suite.

name	LOC	functions	virtual functions	virtual calls	%unreachable virtual calls	aliases per node	ptr-types per node	time min:sec
1. greed	968	47	9	17	35	2	2	0:31
2. garage	143	19	3	10	0	304	34	0:54
3. vcircle	142	16	4	5	0	0	1	0:01
4. office	213	12	4	4	0	49	10	0:04
5. family	109	22	3	3	0	4	3	0:02
6. FSM	98	15	2	1	0	6	2	0:01
7. deriv2	313	34	16	66	45	29	3	1:05
8. shapes	267	34	12	22	23	38	5	0:56
9. deriv1	192	31	13	28	29	20	4	0:14
10. objects	465	59	31	39	85	6	1	0:08
11. simul	339	54	12	7	15	3	1	0:02
12. primes	46	11	4	3	0	48	7	0:16
13. ocean	444	64	10	5	0	81	8	1:58
14. NP	31	7	2	6	0	0	9	0:01
15. city	519	67	2	1	0	322	12	13:27
16. tree	217	26	8	3	33	726	34	8:34
17. employ	894	58	25	4	0	213	14	5:49
18. life	178	21	8	2	0	49	3	0:26
19. chess	392	43	12	1	0	58	16	0:54

Table 6.1: Some characteristics of C^{++} programs analyzed

All these programs exhibit object-oriented programming style through encapsulation, code reuse and modularity.

6.4 Empirical Observations

The test suite of C^{++} programs

Table 6.1 lists some characteristics of programs we analyzed, such as the lines of code (LOC), number of functions, number of virtual functions and number of virtual calls. We also list the number of virtual call sites present in those functions which were found unreachable, number of aliases per ICFG node, number of pointer-types per ICFG node and finally, the program analysis time. Some programs have high analysis times because we have not optimized our prototype for time performance. In the remainder of this section, unless otherwise stated, we normalize the data with respect to virtual call sites in reachable functions. In the charts appearing in this section, we will list the programs by number (from Table 6.1) instead of by name.

Virtual function resolution

Based on the distinct classes of objects pointed to by the receiver at a virtual call site, the algorithm determines which virtual functions in the inheritance hierarchy may be invoked from the call site. In Figure 6.1 we classify the reachable virtual call sites in terms of the number of virtual functions found invocable. Our results support the observation by Calder and Grunwald that although object-oriented libraries support polymorphism through virtual functions, the target of most indirect function calls can be accurately predicted [CG94]. While their observation is based on execution profiles of programs, our results eliminate the dependence on profile data by using compile time analysis which accounts for *all* possible executions of the program. We examined the data and ascertained that the calls for which unique resolution was not possible, would indeed demonstrate polymorphism at run-time, and that the non-uniqueness was never due to approximations in analysis. We also wanted to find out if there was any relation between the program characteristics from Table 6.1 and virtual function resolution results in Figure 6.1. For example, it would have been interesting to find that the resolution results were better for programs with greater number of virtual calls or virtual functions. Table 6.1 is sorted on the decreasing percentage values of virtual function resolution obtained for the programs. There are no corresponding trends in the program characteristics such as lines of code (LOC), number of virtual functions or the number of virtual calls. We conjecture that it is not these program characteristics but the programming style and the functionality of the program which influence the degree of polymorphism demonstrated by the virtual calls. However, a bigger data set is needed to draw definite conclusions about the factors affecting virtual function resolution.

We further classified uniquely resolved virtual calls into those resolved without employing data flow analysis techniques (where the class hierarchy contained only one implementation or the receiver was the address of an object), and those which required the entire functionality of our algorithm (where unique resolution was out of two or more functions). Results in Figure 6.2 suggest that data flow analysis is necessary to obtain good results.

Invocable virtual functions at a call site and call graph construction

Even when unique resolution is not possible, limiting the number of virtual functions invoked at a call site can help subsequent analyses, in that the side effects of the non-invocable functions can be safely eliminated in the context of that call. Given the potential disparity in various implementations of a virtual function, this can translate into improved precision and efficiency of analysis. Limiting the number of invocable virtual functions at a virtual call is relevant to tools which use branch prediction. In Figure 6.3, we report the average percentage of virtual functions found invocable out of those in the class hierarchy of the receiver type at a (reachable or unreachable) virtual call site. We create edges from a virtual call site only to the functions found invocable using resolution information, implicitly building a smaller and more precise call graph than the one built without such information. Therefore, the values in Figure 6.3 can also be interpreted as the size of our call graph *vis à vis* that of a naive approach. Viewing only the relevant aspects of the program (using a call graph browser which avails of the analysis information) can be used for better program understanding. Eliminating non-invocable functions can also be used to prune the class libraries. Such pruning would lead to smaller object code.

As expected, the average percentage of invocable virtual functions was low (implying that many functions were eliminated) for those programs for which

1. a high percentage of virtual calls were found unreachable (in Table 6.1), or
2. we obtained good virtual function resolution (in Figure 6.1), or
3. most of the unique resolution was obtained from two or more candidate functions in the class hierarchy (in Table 6.2).

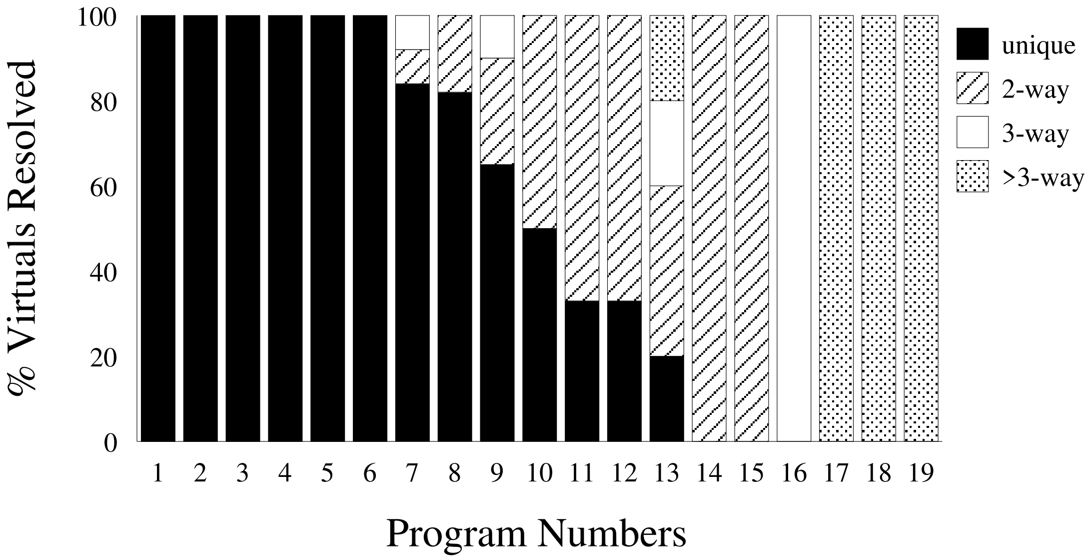


Figure 6.1: Percentage of virtual calls with 1,2,3,>3 invocable functions

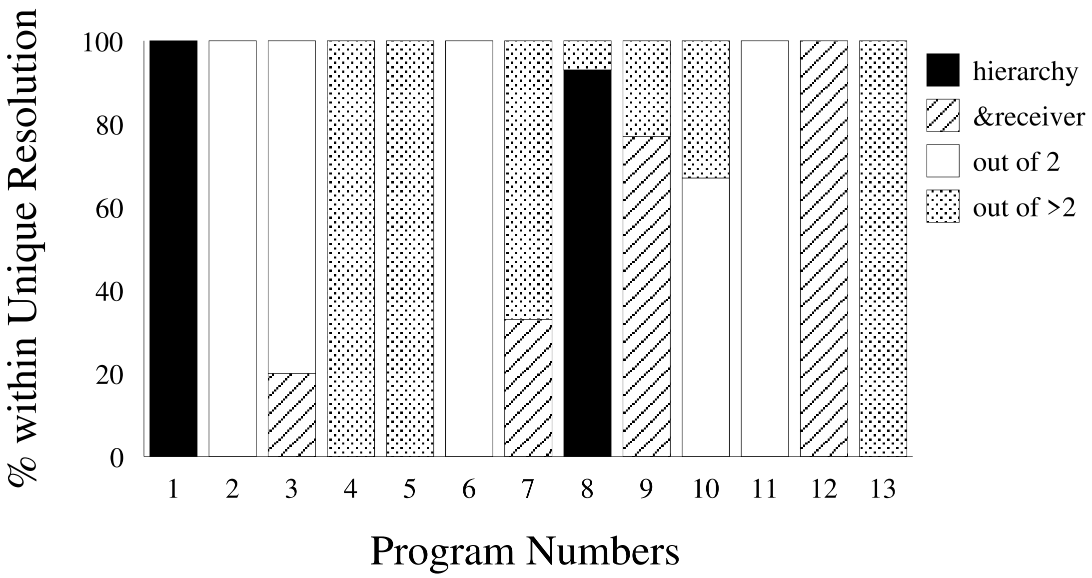


Figure 6.2: Classification of unique resolution for appropriate programs from Figure 6.1

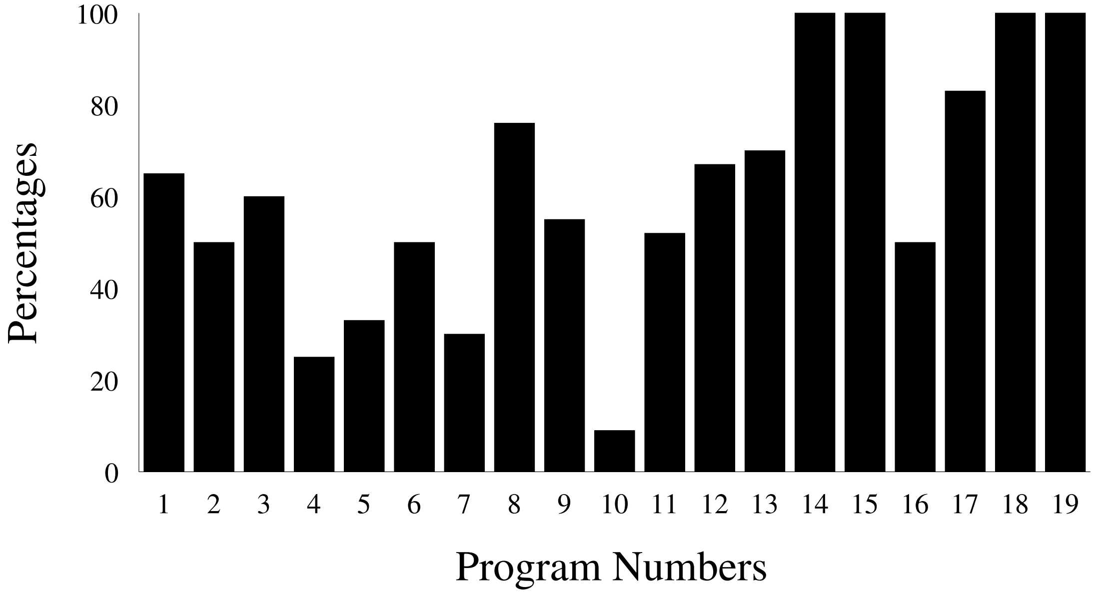


Figure 6.3: Average percentage of invocable virtual functions out of those in class hierarchy at virtual call

name		aliases	pointer-types	virtual resolution (unique, 2-way, 3-way, >3-way)
deriv2	k=3	7064	300	30,3,3,0
	k=2	7064	300	30,3,3,0
	k=1	7098	273	30,2,4,0
deriv1	k=3	2443	208	13,5,2,0
	k=2	2443	208	13,5,2,0
	k=1	2223	176	13,5,2,0
primes	k=4	2986	139	1,2,0,0
	k=3	2164	119	1,2,0,0
	k=2	1466	99	1,2,0,0
ocean	k=4	4623	215	1,2,1,1
	k=3	4255	215	1,2,1,1
	k=2	2971	203	1,2,1,1
chess	k=5	2285	286	0,0,0,1
	k=4	2285	286	0,0,0,1
	k=3	2185	286	0,0,0,1

Table 6.2: Parameterization of k

Parameterization of k

To study the effect of k-limiting on algorithm performance, we parameterized the analysis on the value of k. For each program we picked a minimum value of k, called min_k, such that the object names appearing in the source code would not be k-limited. We analyzed five programs – deriv2, deriv1, primes, ocean and chess – with three values of k, *viz.* $(min_k + 2)$, $(min_k + 1)$ and min_k. These programs were such that we could pick the above three values of k starting with min_k, and analyze the three versions. We had to limit the study to these programs because either the analysis never needed to generate k-limited object names for k values higher than min_k, or the implementation ran out of resources while analyzing the program for the higher k values. For most of the above programs, we found that virtual function resolution was insensitive to the value of k. This is probably because no receiver of a virtual function call was ever k-limited, owing to the selection criterion for min_k.

We also observed that the size of alias and pointer-type solution usually decreased with lower values of k. This was the net result of two opposite effects of k-limiting discussed in Section 5.4.6. On the one hand, multiple object names may map to a single k-limited object name for a lower value of k, leading to reduction in the number of object names and correspondingly alias and pointer-type solution size. On the other hand, further loss of precision caused by a lower value of k leads to increased solution size due to spurious pairs. It was heartening to see that any possible size increase due to loss of precision was almost always overtaken by the reduction due to many-to-one object name mapping. Table 6.2 summarizes our observations. Note that for *deriv2*, the size of alias solution and precision of function resolution suffered for a lower k.

Effect of inlining base class constructors

In C^{++} the derived class constructor calls the constructor of each base class to initialize the inherited base class members. While analyzing the body of the base constructors in certain programs, we realized that the algorithm could not distinguish the calling contexts of distinct derived constructors. These programs were such that distinct derived

name		aliases and pointer-types	*may-hold* per node	*points-to-type* per node	time min:sec
deriv2	original‡	>65000	–	–	–
	inlined	7367	60	3	1:05
shapes	original	4508	105	6	0:54
	inlined	4264	89	5	0:56
deriv1	original	41430	1823	24	25:01
	inlined	2409	42	4	0:14
primes	original	4947	349	12	0:25
	inlined	2291	192	7	0:16

‡ The prototype ran out of resources while analyzing this version.

Table 6.3: Inlining base class constructors

class constructors called the same base class constructor with distinct initialization values for pointer members of the base class. As a result, the information due to one derived constructor went to *all* derived constructors on return from the base constructor. In other words, our approximations led to propagation of information over non-realizable paths. In order to regain the calling context, we inlined the base constructor in each derived constructor. The dramatic improvements in time requirement and precision of analysis are reported in Table 6.3. They underline the need for context sensitive analysis for precision and also suggest a potential technique to improve efficacy of other analyses of object-oriented programs.

Annotating functions as *read-only* or *param-only*

We observed in our test suite that most member functions were small in size. Also, many either were side-effect-free, or only affected outside data through parameter dereferences. This is expected since object-oriented programming style promotes encapsulation and discourages direct references to non-local data. The side-effect-free functions were typically *observers*, which returned the value of a data member. We called such

name		%*r-o*	%*p-o*	*may-hold*	*points-to-type*	#malloc	time min:sec
garage	original			119654	11294	1789065	1:01
	annotated	26	0	104076	9906	1562648	0:54
deriv2	original			111160	2620	2123106	2:31
	annotated	62	29	50416	2620	957134	1:05
shapes	original			46960	3172	1738203	1:48
	annotated	71	15	46960	2939	783060	0:56
deriv1	original			26312	1414	617066	0:26
	annotated	52	32	14253	1414	339094	0:14
simul	original			4868	715	121784	0:03
	annotated	83	0	3120	467	80546	0:02
tree	original			579742	12425	12597024	11:22
	annotated	23	58	479146	12425	8937086	8:34
employ	original			439659	7988	8489510	7:42
	annotated	33	0	317769	7988	6690717	5:49

Table 6.4: Functions annotated as *read-only* or *param-only*

functions *read-only*. The other observable category of functions updated the dereferences of formal parameters, possibly changing the alias or typing relationships in the calling function. We called them *param-only* functions. *Read-only* functions were particularly useful for analysis efficiency because we did not have to propagate the predicates through these functions. We propagated the predicates directly from *call* to *return*. *Parameter-only* functions implied that we needed to propagate through the functions only those predicates involving actual parameters at the call site. All other predicates were propagated directly to the corresponding *return* node. We selected 7 programs where a sizeable proportion of functions could be classified into either category by code inspection[38]. Note that this transformation was not intended to improve the precision of our solution, but to improve the time and storage efficiency of analysis by utilizing an easily observable property of the functions. Indeed, certain *may-hold* or *points-to-type* are absent from the computed solution for annotated functions, since they were known to be preserved through such functions and hence, were never propagated to them. Nevertheless, it should be noted that the above optimization is useful only to applications where the absence of such preserved information does not violate safety. For example, this optimization is not appropriate if one needed information about object names *used* by a function, either directly or indirectly. In Table 6.4, we compare the results of analyzing the annotated program *vis à vis* the original program. We list the percentages of functions annotated *read-only* (column %*r-o*) and *param-only* (column %*p-o*), the size of solution in terms of the number of *may-hold* and *points-to-type* predicates over the entire program, the number of storage allocations (column #malloc) done by our implementation while analyzing the program and finally, the time taken for analysis. As expected, we found that the improvements were proportional to the percentages of functions so annotated, especially if they were marked *read-only*.

[38] A simple flow insensitive algorithm would be able to find similar information, although it may be more conservative than that derived by hand. It remains to be seen whether such an automated process would still give comparable time improvements.

name	LOC	*may-hold*	*points-to-type*	time (sec)	predicates/sec
NP	31	0	675	0.37	1824
primes	46	13398	513	16.07	866
FSM	98	1402	236	0.88	1861
family	109	2435	639	1.70	1808
vcircle	142	372	52	0.31	1367
garage	143	104076	9906	54.28	2100
life	178	46184	1130	25.50	1855
deriv1	192	14253	1414	14.12	1109
office	213	7510	1461	3.91	2238
tree	217	479146	12425	513.76	957
shapes	267	46960	2939	55.53	899
deriv2	313	50416	2620	64.68	819
simul	339	3120	467	2.00	1794
chess	392	66758	11598	54.31	1443
ocean	444	147820	5946	117.61	1307
objects	465	8368	581	8.11	1103
city	519	770868	12162	807.17	970
employ	894	317769	7988	348.76	934
greed	968	22490	2081	30.66	801

Table 6.5: Empirical corroboration: algorithm is linear in solution size - I

name	virtual calls	predicates/sec
FSM	1	1861
chess	1	1443
city	1	970
life	2	1855
primes	3	866
family	3	1808
tree	3	957
office	4	2238
employ	4	934
vcircle	5	1367
ocean	5	1307
NP	6	1824
simul	7	1794
garage	10	2100
greed	17	801
shapes	22	899
deriv1	28	1109
objects	39	1103
deriv2	66	819

Table 6.6: Empirical corroboration: algorithm is linear in solution size - II

Time performance of prototype implementation

In Section 6.2, we claimed that with reasonable assumptions, our algorithm is linear in the size of *may-hold* and *points-to-type* solution. In Tables 6.5 and 6.6, we provide empirical corroboration of this claim using the same programs listed in Table 6.1. We divided the total number of predicates (#*may-hold* + #*points-to-type*) with the time (in seconds) taken to analyze the program. Although the total size of *may-hold* and *points-to-type* solution varies significantly, #predicates per second indeed seems independent of the total solution size. In Table 6.5, we list the programs in increasing order of lines of code (LOC) while in Table 6.6, we list them in increasing order of the number of virtual calls. We do not observe any trends with respect to program size or the number of virtual calls, indicating the value of #predicates per second is independent of these program parameters.

6.5 Limitations of the Implementation

Although our algorithm is capable of analyzing all C^{++} programs subject to restrictions mentioned in Chapter 1 (p. 5), our *proof-of-the-concept* prototype has the following additional limitations.

1. Our front end parser does not handle multiple source files and libraries. We manually merge the multiple source files into a single file before analyzing the program.

2. The implementation does not handle multiple inheritance. Our inability to avoid duplication of members inherited from a virtual base class is the cause of this limitation. To overcome it would require additional information from our *black-box* front end.

3. We use an encoding scheme which assigns a unique ID for each object name generated during analysis. The encoding utilizes a fast growing function of the number of variables, the number of types, the maximum number of members any class in the program and finally, the number of dereferences (i.e., k-limit). This

placed a severe constraint on the programs we could analyze, as the ID would quickly exceed MAXINT. In some cases, the encoding failed when we tried to analyze a program from our test suite for a higher value of k.

4. The object name encoding assumes that there are no nested class declarations. In other words, class A cannot have a member which is a class B object. We hardly encountered nested class declarations in our test data.

5. The object name encoding also assumes that there are no static members in classes.

Our efforts to scale up the analysis to handle bigger programs involve lifting these restrictions, and are described in Chapter 8.

Chapter 7

Applications and Related Work

7.1 Def-Use Associations for C

Def-use associations are necessary for a range of software development environment tools, and are crucial to data-flow-based testing systems [FW88, HS89, HS90, OW91, Ost90, RW85]. Previous efforts to calculate interprocedural def-use associations [HS90, HS91], focused on *Pascal* like programs; their implementation did not account for pointer aliasing. The accuracy of the def-use information determines the efficiency of the test case coverage. If imprecise information is an underestimate of the def-use associations, it can lead to missed test paths; if it is an overestimate of the def-use associations it can lead to generating unnecessary test cases, resulting in more lengthy testing. If we obtain a *safe* approximation to the def-use associations, then only the latter situation can occur.

Debuggers based on static and dynamic slicing methods [AH90, Gor93, HRB90, KL90, OO84, Ven91, Wei84] offer the promise of efficient on-line analyses of programs. The aim of slicing is to concentrate the programmer's attention on those parts of the program significant to the computation under current investigation; the more imprecise the def-use information, the less effectively the slicing method can prune away unrelated computations.

Techniques for merging independently altered versions of programs [HPR89, YHR90] are of great interest to the programming-in-the-large community. In this domain also, the precision of the def-use information greatly impacts the comparison of the program semantics before and after changes; imprecise information may lead to incorrect conclusions of semantic differences.

The classical problem of intraprocedural reaching definitions [Hec77] was extended to handle interprocedural propagation by Harrold and Soffa [HS90]. In the absence of aliasing, our algorithm is equivalent to theirs in that both obtain the precise solution. While our approach extends naturally to account for aliasing induced by pointers as well as parameter bindings, they suggest factoring the aliases into data flow problem solution separately, as in previous work by Lomet [Lom77]. Lomet's approach suggested that an approximation of program semantics could be obtained by analyzing the procedure under different aliasing conditions and then combining them at some loss of precision. This is conceptually similar to our approach, except we are dealing with pointer induced aliasing. By solving a conditional version of the data flow problem, we avoid some of the loss of precision that Lomet incurred. The algorithm by Liu and Taha [LT90] has similar shortcomings in the presence of aliasing. Our work is *qualitatively different* from these analyses, as we use flow sensitive pointer aliasing *during* our reaching definitions calculation, rather than a factored approach.

7.2 Type Determination and Aliasing for C^{++}

Research in optimizations for object-oriented languages has mainly concentrated on two analysis techniques: type feedback (run time) and type inference (compile time). A comparison of the two techniques appears in [AH95] using SELF as the target language. Although type feedback can be (and has been) applied to languages like C^{++} which use pointers extensively, existing type inference techniques for SELF do not extend naturally to handle the additional complications of pointer induced aliasing. Type feedback uses information from previous or current executions of the program to determine the possible receiver classes, while type inference approaches the same goal by analyzing the program's source code at compile time.

Type Feedback Recent work by Dean *et al.* [DCG95] uses a combination of type feedback and static class hierarchy information for *specialization* (compiling multiple versions of a method) based on subsets of possible argument classes of the method. Hölzle and Ungar [HU94] employ type feedback for optimizing dynamically dispatched calls in SELF. Calder and Grunwald [CG94] use their profile based measurements on C^{++} programs to demonstrate that virtual function resolution is applicable and beneficial to C^{++}. Prior work on improving run-time efficiency of SELF uses customization, iterative type analysis and inline caches to replace dynamic binding with procedure calls or inlined code [CU89, CU90, HCU91]. Although type feedback techniques are more suitable for interactive systems and are scalable to handle large and extensible systems, the results of type feedback depend heavily on run time information or representative profiles [AH95]. Although they are effective on *most* executions, their predictions cannot be extrapolated to *all* executions of the program and hence must provide run time support to handle the situations which may deviate from those inferred from previous executions.

In the remainder of this chapter, we concentrate on compile time analyses of object-oriented languages. We discuss in detail the existing type inference techniques based on class hierarchy, type constraints, function pointer analysis and control flow analysis of higher order languages. The type determination aspect of our work can be viewed as type inference using data flow analysis. Prior to our work, the interaction between pointer induced aliasing and type determination has enjoyed little attention. We discuss the approaches which factor out aliasing and type determination for object-oriented languages.

Class Hierarchy Analysis Class hierarchy information is used at link time by Fernandez to replace method invocations with direct calls in Modula-3 programs [Fer95]. Since Modula-3 has *opaque* class hierarchy (where the separate compilation module compiling a derived class may have little information about implementation of the base class), the late binding mechanism is required even for invoking methods inherited from a parent class. This language feature makes hierarchy analysis a very effective link time mechanism to eliminate indirect calls. In contrast, in a C^{++} program, an equivalent method invocation would have already been compiled as a direct function call. Diwan *et al.* have effectively applied hierarchy information and a limited form of static analysis to optimize Modula-3 programs [DMM95]. Nevertheless, such simple techniques proved less effective on our C^{++} benchmarks, as demonstrated empirically in Section 6.4. Kuhn and Binkley use a simple transformation which replaces virtual invocations in C^{++} with direct calls within a decision statement [KB95]. The class hierarchy information is used to obtain the list of virtual functions possibly invocable at each virtual call site. Having obtained an equivalent program without the construct of virtual calls, they propose applying existing optimization techniques to improve program performance. Since hierarchy analysis alone is not shown to be effective in obtaining virtual function resolution, it is not clear to what extent the approach would be useful in practice.

Constraint based Analysis Suzuki attempted eliminating dynamically dispatched calls in Smalltalk programs by using unification based type inference [Suz81]. He performed iterative analysis to determine the receiver type as a set of classes. Although the algorithm failed to type-check common programs, it served as a starting point for other efforts to infer types in object-oriented languages. The algorithm by Palsberg and Schwartzbach infers types of expressions in an object oriented language with inheritance, assignments and late bindings [PS91]. They set up type constraints and compute the least solution in worst case exponential time. The algorithm does not perform control flow analysis nor does it track the values of objects. They suggest type determination using data flow analysis as an orthogonal way to aid their algorithm in performing optimizations and type safety checks. Agesen *et al.* present a polynomial time constraint based type inferencing algorithm for SELF to compute approximate types of all expressions in the program, including the types of receivers at dynamic dispatch sites [APS93]. Kumar *et al.* improve on this technique by utilizing the *Static Single Assignment* form of object-oriented programs [KAI95]. Since each use of an object is reached by at most one definition, their framework can track the program points where a particular receiver may have a Nil value. Plevyak and Chien describe an incremental constraint based type inference technique for Concurrent Aggregates, a concurrent object-oriented language [PC94]. While constraint based inferencing is most suitable for purely dynamic, untyped languages like SELF and ours for typed languages like C^{++}, the two approaches may supplement each other for languages which combine these separate domains. The former can benefit from type declarations and flow analysis to keep track of instance variable values. With this information, it can refine the type solution obtained by satisfying the flow insensitive constraints. The latter can use the constraint based approach to arrive at an initial approximation of types for untyped variables and proceed with flow analysis.

Function Pointer Analysis Since virtual function calls in C^{++} can be modeled using function pointers in C, algorithms which handle them [BCCH94, EGH94, WL95, Wei80] may be applied towards analysis of C^{++}. Nevertheless, these approaches i) have a different emphasis and are ill-tuned to function pointer analysis and/or ii) have impractical worst case complexity, and are unsuitable in C^{++}context where virtual functions are ubiquitous.

Flow Analysis of Higher Order Languages Techniques for flow analysis of higher order languages like Scheme [Har89, JM79, Shi90] may be adapted for analyzing function pointers (and hence C^{++}); however, they do not have acceptable complexity for reasonable precision on real programs. The emphasis seems to be on theoretical issues such as being able to solve the problem. It is not clear whether the theory can lead to a practical implementation. Jagannathan and Wright compare various existing techniques for analysis of such languages and motivate the need for alternative data flow techniques [JW95].

Flow Analysis of Object-Oriented Languages The algorithm by Larcheveque factors out from type determination the side effects of function invocations and aliasing due to parameter bindings as well as pointers [Lar92]. The suggested algorithms for these problems [CK89, Wei80] are grossly approximate and unsuitable in a C^{++} context. Parameswaran has developed an algorithm which performs alias analysis without the knowledge of the receiver type at an invocation site and thus assuming that all corresponding virtual functions are invocable [Par92]. He then uses the precalculated alias information for type determination. We show that aliasing and type determination are inseparable in the general case, therefore a factored approach is not desirable. Suedholt and Steigner use a concept of *representant* virtual function to keep information about all the virtual functions with the same name [SS92]. This approach leads to the loss of context which distinguishes one virtual function from others. Vitek *et al.* present an algorithm which discovers the potential classes of objects for a simple object oriented language as well as a safe approximation to their lifetimes [VHU92].

Pointer Induced Aliasing and Type Determination We have shown that in C^{++}, the problems of aliasing and type determination are practically inseparable in the presence of general purpose pointers. We use the conditional approach proposed by Landi and Ryder [LR91] to solve a combined problem. Since C^{++} is closely related to C (but distinguishes itself from the latter with the significant addition of object-oriented paradigm and polymorphism), our work closely relates to Landi's pointer induced aliasing work for C [Lan92a]. Other techniques for flow-sensitive pointer aliasing include [CBC93, Deu94, EGH94, WL95]. They are disparate in representation, complexity and precision. Nevertheless, we believe it is possible to define the interaction between aliasing and type determination and solve the combined problem using these techniques. Recently, Carini *et al.* [CHS95] have developed a type determination algorithm for C^{++} using the aliasing technique presented in [CBC93]. In contrast to our explicit representation of type information (in terms of pointer-type pairs), they recover in a lazy manner the information about types stored implicitly in their alias representation.

Chapter 8

Conclusions and Future Work

In this thesis, we have presented a unified approach to solve fundamental data flow analysis problems for languages with general purpose pointer usage. The technique of conditional analysis was introduced by Landi to solve the problem of aliasing in the presence of pointers [LR91]; we have extended it to solve other problems which, like aliasing, are crucial to compile time analysis. We have included a comprehensive treatment of two representative problems: def-use associations for C and type determination for C^{++}. In both these language domains, the thesis has contributed to the state of the art at three levels: theoretical analysis of the problems, development of safe and approximate algorithms, and empirical demonstration of the practicality and utility of these algorithms using prototype implementations.

Precise compile time analyses are intractable in the presence of multiple level pointers, and undecidable when recursive data structures are allowed [Lan92b]. We have proved that the def-use associations and type determination problems are NP-hard even when only single level pointers are allowed, providing an insight into the inherent complexity of these problems. This motivated us to develop safe and approximate algorithms which concentrated on the sources of difficulty, before inheriting the additional intricacies of multiple level pointer usage. We have studied the close interaction of each of these problems with aliasing and developed a unified approach to exploit the interaction during calculation. We have shown that there would be further loss of precision if one problem was solved in isolation while making conservative assumptions about the other.

Ours is the first polynomial-time approximation algorithm for def-use associations for C programs. The algorithm has contributed to further research in the area of software re-engineering: (i) it has been extended to improve the quality of def-uses for dynamically allocated locations [AL95], (ii) it has been adapted to perform data flow testing on classes [HR94] and (iii) it has been used to study the effectiveness of data flow based test coverage [HFGO94]. Our algorithm for type determination and aliasing is the first attempt to solve this combined problem for an object-oriented language. Empirical studies of C^{++} program characteristics have shown that most indirect (virtual) method invocations may be uniquely resolved to direct function calls at run time. In other words, most virtual calls are run time monomorphic. Our algorithm has provided a compile time mechanism to support this result.

We have shown the viability of our approach using prototype implementations for the algorithms presented in this thesis. The empirical results obtained using our def-use associations algorithm for single level pointers have been highly precise, while the results of the recently extended implementation which handles multiple level pointers have been promising. Our C^{++} implementation has yielded impressive virtual function resolution on the test suite comprised of a variety of publicly available C^{++} programs. We have shown that simpler flow-insensitive techniques like class hierarchy information alone are not effective towards this goal. We have also suggested ways to exploit some observable properties of C^{++} programs to improve analysis performance.

We view the research reported in this thesis as an initial attempt to perform practical compile time analysis of C and C^{++}. The necessity of good compile time analysis information to enhance the efficacy of software tools is well established. Although the efforts to obtain and apply such information have been largely successful for *Fortran*, a lot more work must go into the analysis of languages which use pointers and dynamic binding for function calls. We envisage the following future directions for our research.

Short term goals

There are engineering issues which need to be ironed out to make our algorithms more practical. Also, our *proof of the concept* prototype implementations require substantial enhancements so that they efficiently provide useful analysis information for larger programs. The following issues are not independent, but may be viewed as components of a single effort to address these short term goals.

1. Our algorithms have a number of limitations on the language constructs handled (see Chapter 1, p. 5). Although many constructs present open research problems, the analysis should be able to detect such constructs and continue with very conservative and safe assumptions to handle them. For example, when an indirect call involving a function pointer is encounted, the analysis can consider all the functions with the appropriate type signature to be invocable and obtain safe, albeit grossly approximate results in the context of that call.

2. Our implementations need to be optimized for time performance. For some of the test programs, the analysis times are unacceptably high. The size of the alias solution has been a major factor in these *pathological cases*, since our algorithms are linear in the size of the solution. Our experiments to improve the analysis performance using the observable characteristics of the test programs have yielded encouraging results in reducing the solution size (See Table 6.3 and Table 6.4 in Chapter 6). Currently we compute exhaustive solutions for def-use associations, aliasing and type determination. For example, as a result of the assignment "`p = &a`", we create the aliases of the form $<p{\rightarrow}m_k, a.m_k>$ for all data members m_k (direct as well as inherited) of the class of a. By abstracting all such aliases as $<*p, a>$, we can reduce the solution size. The above alias can be expanded for a particular pointer-typed member whenever it participates in further analysis. The alias may not have to be expanded for data members which are not pointers, as they do not individually participate in aliasing. Also, our algorithms use an unoptimized version of aliasing. An independent effort in optimizing this version while solving the modification side effects (MOD) problem for C has been quite successful [LRZA96]. We plan to incorporate similar optimizations to improve our analysis performance as well.

3. We have experienced some bottlenecks with the front end in our attempts to analyze realistic programs. (i) Limitations of the front end (i.e., the parsers are not capable of providing some information we require for better performance of our

back end implementation): For example, the PTT system used in our C implementation stores no information about types of expressions involved in type casts, and the MasterCraft parser for C^{++} requires the entire program under analysis to be contained in a single file. (ii) Inadequate functionality of interface with the front end: For example, although the MasterCraft system handles multiple inheritance, we cannot obtain a list of data members of a derived class in a multiple inheritance hierarchy. We seek a suitable front end in terms of capability and ease of interface with our back end analysis.

4. We need a more powerful and efficient representation for object names.

5. We plan to enhance our algorithms to safely handle the constructs which can create circular data structures. It involves additional case analysis during the intraprocedural propagation phase of the algorithms. Recent work on *shape analysis* for pointer data structures [GH96, SRW96] is relevant in this respect.

6. Our analysis works on the intermediate representation for the whole source program and keeps the entire analysis database in main memory, an approach clearly not suitable for analyzing large programs. We need to re-engineer the algorithms towards modular analysis. A possible approach would be to analyze each module with safe assumptions regarding other modules, and refine the analysis results using an iterative process.

7. Given the above limitations, we were severely restricted in choosing the programs that could be included in the test suite. We need to enlarge our C and C^{++} benchmarks (i) to gauge the utility of our analysis, (ii) to better study the program characteristics affecting our analysis and (iii) to obtain more confidence in our empirical results.

Open research problems

To meet the ultimate goal of this research – to provide useful semantic information leading to powerful software re-engineering and reverse-engineering tools for languages like C and C^{++} – we need to address the following open problems.

1. Popular programming language constructs like function pointers, type casts between structured and scalar types, and exceptions pose difficult problems for compile time analysis. There is a strong need for practical but reasonably precise analysis techniques to solve these problems.

2. Various approaches exist in literature to solve the fundamental analysis problem of pointer aliasing. We have presented a unified framework to solve two practically significant problems, data dependence analysis and virtual function resolution, in combination with the aliasing technique proposed by Landi. We believe that these problems can be combined with the other approaches for aliasing. It would be interesting to see which approaches (or a combination thereof) lead to practical analyses of large programs. There are tradeoffs in cost (time as well as space utilization) *vs.* precision, which may be relevant in this investigation.

3. We would like to further extend this research to solve other data flow problems such as def-use associations and modification side effects (MOD) problems for C^{++}. The def-use associations analysis may be applied to build static slices, and used in turn to increase the effectiveness of software tools such as source level debuggers and semantic browsers.

4. We wish to investigate extending our analysis approach to other object-oriented languages (e.g. Smalltalk and SELF) as well as the languages which use other programming paradigms (e.g. functional languages like Scheme).

5. We would like to quantify the actual impact of the program transformations enabled by our research. Our def-use associations analysis already has been applied to study the data flow based test coverage of C programs. In Section 6.3, we listed various applications of virtual function resolution, such as inlining, call graph generation, and pruning the class libraries. Associated with each application, we mentioned the possible benefits, ranging from execution performance improvement to program understanding. Our analysis results need to be incorporated in appropriate software tools to quantify these benefits.

Appendix A

Def-Use Associations for C

A.1 Reaching Definitions Using May and Must Alias Information

Calculation at each ICFG node

To shorten our equations, let the notation "(ARD or $RA = \emptyset$)" represent the condition "for $ARD = \emptyset$ or $< n_k : a >$ (where n_k and a make up the last component of the *reaches*); for all RA such that at least one of ARD and RA is $\emptyset$".

Assignment node: Consider the fixed-location assignment n_i "`a = ...`":

1. for all reaching must alias sets RMA:

   $reaches(n_i,\ (\emptyset, \emptyset, RMA),\ < n_i : a >) = true$

   $reaches(n_i,\ (< n_i : a >, \emptyset, RMA),\ < n_i : a >) = false$

   $reaches(n_i,\ (\emptyset, RA, RMA),\ < n_i : a >) = false$

   (any RA except $\emptyset$)

2. for all $n_k \neq n_i$, (ARD or $RA = \emptyset$), all reaching must alias sets RMA:

   $reaches(n_i,\ (ARD, RA, RMA),\ < n_k : a >) = false$

3. for all fixed-locations $b \neq a$, all nodes n_k, (ARD or $RA = \emptyset$), all reaching must alias sets RMA:

   $$reaches(n_i,\ (ARD, RA, RMA),\ < n_k : b >) = \bigvee_{n_p \text{ pred of } n_i} (reaches(n_p,\ (ARD, RA, RMA),\ < n_k : b >))$$

Consider the assignment n_i "`*p = ...`":

- for all nodes n_k, (ARD or $RA = \emptyset$), and all fixed-locations b, all reaching must alias sets RMA:

$reaches(n_i,\ (ARD, RA, RMA),\ < n_k : b >) =$

$$\left(\bigvee_{n_p \text{ pred of } n_i} \begin{array}{c} n_k = n_i \wedge ARD = \emptyset \wedge \\ (may\text{-}hold(n_p,\ RA,\ <*p, b>)) \end{array} \right)$$

$$\bigvee$$

$$\left(\begin{array}{ll} false & \text{if } must\text{-}hold(n_p,\ <*p, b>) \subseteq RMA \\ \beta & \text{otherwise} \end{array} \right)$$

$$\beta = \bigvee_{n_p \text{ pred of } n_i} (reaches(n_p,\ (ARD, RA, RMA),\ < n_k : b >))$$

Entry node: For a call node c_p, let

$$RMA_{c_p} = bind_{c_p}(\emptyset) \cup \bigcup_{\substack{\text{all } [c_p,<\text{a,b}>] \text{ in} \\ \text{the must alias solution}}} bind_{c_p}(<a, b>).$$

For procedure `P`, the set of all reaching must alias sets is $\{RMA_{c_p} \mid c_p \text{ invokes } \texttt{P}\}$.

1. for each formal `f`, all reaching must alias sets RMA:

   $reaches(n_i,\ (\emptyset, \emptyset, RMA),\ < n_i : f >) = true$

   $reaches(n_i,\ (< n_i : f >, \emptyset, RMA),\ < n_i : f >) = false$

   $reaches(n_i,\ (\emptyset, RA, RMA),\ < n_i : f >) = false$

   (any RA except $\emptyset$)

2. for all globals g, all nodes n_k, and all call sites c_p of n_i:

$$reaches(n_i,\ (< n_k : g >, \emptyset, RMA_{c_p}),\ < n_k : g >) =$$

$$\bigvee_{\substack{(ARD \text{ or } RA\ =\ \emptyset) \\ \text{and } RMA' \text{ of} \\ \text{calling procedure}}} (reaches(c_p,\ (ARD, RA, RMA'),\ < n_k : g >))$$

$$reaches(n_i,\ (\emptyset, RA, RMA_{c_p}),\ < n_k : g >) = false$$

(any RA)

$$reaches(n_i,\ (< n_k : g >, RA, RMA_{c_p}),\ < n_k : g >) = false$$

(any RA except $\emptyset$)

3. for all locals l, all nodes n_k, and all call sites c_p of n_i:

$$reaches(n_i,\ (< n_k : nv\_l >, \emptyset, RMA_{c_p}),\ < n_k : nv\_l >) =$$

$$\bigvee_{\substack{\alpha\ \in\ \{\text{l,nv\_l}\} \text{ and} \\ (ARD \text{ or } RA\ =\ \emptyset) \\ \text{and } RMA' \text{ of} \\ \text{calling procedure}}} (reaches(c_p, (ARD, RA, RMA'), < n_k : \alpha >))$$

$$reaches(n_i,\ (\emptyset, RA, RMA_{c_p}),\ < n_k : nv\_l >) = false$$

(any RA)

$$reaches(n_i,\ (< n_k : b >, RA, RMA_{c_p}),\ < n_k : nv\_l >) =$$

$false$ (any RA except $\emptyset$)

4. for all other cases (ARD or $RA = \emptyset$):

$$reaches(n_i,\ (ARD, RA, RMA),\ < n_k : a >) = false$$

Return node: In the following, let RMA_{c_p} (any call node c_p) be as defined on p. 122, and for notational simplicity consider $may\text{-}hold(n_i,\ RA,\ \emptyset)$ to be $true$ for all nodes n_i and all reaching aliases RA.

For c_i the call node of the call site of n_i, x_k the exit of the called procedure, (ARD or $RA = \emptyset$), all reaching must alias sets RMA, b a global or nv_l for some local l:

$reaches(n_i,\ (ARD, RA, RMA), < n_m : b >) =$

$$
\begin{array}{ll}
1. & reaches(x_k, (\emptyset, \emptyset, RMA_{c_i}), < n_m : b >) \vee \\
2. & reaches(c_i, (ARD, RA, RMA), < n_m : b >) \wedge \\
 & reaches(x_k, (< n_m : b >, \emptyset, RMA_{c_i}), < n_m : b >) \\
3. & \bigvee\limits_{\substack{\text{all reaching}\\ \text{aliases } RA'}} \begin{pmatrix} reaches(x_k, (\emptyset, RA', RMA_{c_i}), < n_m : b >) \\ \wedge may\text{-}hold(c_i, RA, back\text{-}bind_{c_i}(RA')) \end{pmatrix}
\end{array}
$$

For c_i the call node of the call site of n_i, x_k the exit of the called procedure, (ARD or $RA = \emptyset$), all reaching must alias sets RMA, l a local:

$reaches(n_i,\ (ARD, RA, RMA), < n_m : l >) =$

$$
\begin{array}{ll}
1. & /* \text{ there is no corresponding case 1 } */ \\
2. & reaches(c_i, (ARD, RA, RMA), < n_m : l >) \wedge \\
 & reaches(x_k, (< n_m : nv\_l >, \emptyset, RMA_{c_i}), < n_m : nv\_l >) \\
3. & \bigvee\limits_{\substack{\text{all reaching}\\ \text{aliases } RA'}} \begin{pmatrix} reaches(x_k, (\emptyset, RA', RMA_{c_i}), < n_m : nv\_l >) \\ \wedge may\text{-}hold(c_i, RA, back\text{-}bind'_{c_i}(RA')) \end{pmatrix}
\end{array}
$$

Any other node: Because *reaches* at a node includes the effects of that node, other nodes (e.g., *call*, *exit*, *conditional*, ...) simply collect the reaching definitions information. Thus:

$reaches(n_i,\ (ARD, RA, RMA), < n_k : b >) =$

$$
\bigvee_{n_p \text{ pred of } n_i} (reaches(n_p,\ (ARD, RA, RMA), < n_k : b >))
$$

Reaching definitions from conditional reaching definitions

The *rdtop* calculation with aliasing is conceptually identical to the *rdtop* calculation in Section 3.3.2. The *rdtop* solution for the program in Figure 3.1 appears in Tables A.10 and A.11.

For each node n_i in the ICFG, *rdtop* is calculated as follows:

- if n_i is an entry node or return node (special case these two as they do not have intraprocedural predecessors).

  $rdtop(n_i) =$

$$\left\{ < n_k : b > \left| \begin{array}{l} (\exists\ (ARD,RA,RMA))\ \text{such that} \\ reaches(n_i,\ (ARD,\ RA,\ RMA),\ < n_k : b >) \\ \text{is } true \end{array} \right. \right\}$$

- otherwise $rdtop(n_i) =$

$$\bigcup_{\substack{n_p \text{ pred} \\ \text{of } n_i}} \left\{ < n_k : b > \left| \begin{array}{l} (\exists\ (ARD,RA,RMA))\ \text{such that} \\ reaches\begin{pmatrix} n_p, \\ (ARD,RA,RMA), \\ < n_k : b > \end{pmatrix} \\ \text{is } true \end{array} \right. \right\}$$

A.2 Solutions for Running Example

Procedure main		
node	*RA*	alias
n_4,c_1	$\emptyset$	$< *p, a >$
r_1,n_5	$\emptyset$	$< *p, a >$
n_6,c_2	$\emptyset$	$< *p, b >$
r_2,n_7	$\emptyset$	$< *p, b >$
x_1	$\emptyset$	$< *p, a >$
	$\emptyset$	$< *p, b >$

Procedure R		
node	*RA*	alias
e_2,n_8,n_9,x_2	$< *p, a >$	$< *p, a >$
	$< *p, b >$	$< *p, b >$
	$< *p, *f >$	$< *p, *f >$
	$< *f, a >$	$< *f, a >$
	$< *f, b >$	$< *f, b >$

This table summarizes the *may-hold*(*node*, *RA*, *alias*) predicates that are *true* for the program in Figure 3.1. Reflexive cases (i.e., *may-hold*(n, $\emptyset$, $< x, x >$) for all nodes n and all object names x) which are always *true* are omitted for brevity.

Table A.1: Summary of *may-hold*

Procedure main	
node	alias
n_4,c_1	$< *p, a >$
r_1,n_5	$< *p, a >$
n_6,c_2	$< *p, b >$
r_2,n_7	$< *p, b >$
x_1	$< *p, a >$
	$< *p, b >$

Procedure R	
node	alias
e_2,n_8,n_9,x_2	$< *p, a >$
	$< *p, b >$
	$< *p, *f >$
	$< *f, a >$
	$< *f, b >$

This table summarizes the may alias solution for the program in Figure 3.1. Reflexive cases (i.e., [n,$< x, x >$] for all nodes n and all object names x) are omitted for brevity.

Table A.2: Summary of may alias

Procedure main		
node	*MA*	*value*
n_4,c_1	$< *p, a >$	$\emptyset$
r_1,n_5	$< *p, a >$	$\emptyset$
n_6,c_2	$< *p, b >$	$\emptyset$
r_2,n_7	$< *p, b >$	$\emptyset$

Procedure R		
node	*MA*	*value*
e_2,n_8,n_9,x_2	$< *p, a >$	$\{< *p, a >\}$
	$< *p, b >$	$\{< *p, b >\}$
	$< *p, *f >$	$\{< *p, *f >\}$
	$< *f, a >$	$\{< *f, a >\}$
	$< *f, b >$	$\{< *f, b >\}$

This table summarizes the *must-hold*(*node*, *MA*) = *value* functions that are not $\perp$ for the program in Figure 3.1. Reflexive cases (i.e., *must-hold*(n, $< x, x >$) for all nodes n and all object names x) are all equal to $\emptyset$ and are omitted for brevity.

Table A.3: Summary of *must-hold*

Procedure main	
node	alias
n_4,c_1	$< *p, a >$
r_1,n_5	$< *p, a >$
n_6,c_2	$< *p, b >$
r_2,n_7	$< *p, b >$

Procedure R	
node	alias
e_2n_8,n_9,x_2	$< *p, *f >$

This table summarizes the must alias solution for the program in Figure 3.1. Reflexive cases (i.e., $[n, < x, x >]$ for all nodes n and all object names x) are omitted for brevity.

Table A.4: Summary of must alias

Procedure `main`			
node	*ARD*	*RA*	rd
n_1	$\emptyset$	$\emptyset$	$< n_1 : a >$
n_2,n_3	$\emptyset$	$\emptyset$	$< n_1 : a >$
	$\emptyset$	$\emptyset$	$< n_2 : b >$
n_4,c_1	$\emptyset$	$\emptyset$	$< n_1 : a >$
	$\emptyset$	$\emptyset$	$< n_2 : b >$
	$\emptyset$	$\emptyset$	$< n_4 : p >$
r_1,n_5	$\emptyset$	$\emptyset$	$< n_1 : a >$
	$\emptyset$	$\emptyset$	$< n_2 : b >$
	$\emptyset$	$\emptyset$	$< n_4 : p >$
	$\emptyset$	$\emptyset$	$< n_8 : a >$
	$\emptyset$	$\emptyset$	$< n_9 : c >$
n_6,c_2	$\emptyset$	$\emptyset$	$< n_1 : a >$
	$\emptyset$	$\emptyset$	$< n_2 : b >$
	$\emptyset$	$\emptyset$	$< n_6 : p >$
r_2,n_7	$\emptyset$	$\emptyset$	$< n_1 : a >$
	$\emptyset$	$\emptyset$	$< n_2 : b >$
	$\emptyset$	$\emptyset$	$< n_6 : p >$
	$\emptyset$	$\emptyset$	$< n_8 : b >$
	$\emptyset$	$\emptyset$	$< n_9 : c >$
x_1	$\emptyset$	$\emptyset$	$< n_1 : a >$
	$\emptyset$	$\emptyset$	$< n_2 : b >$
	$\emptyset$	$\emptyset$	$< n_4 : p >$
	$\emptyset$	$\emptyset$	$< n_8 : a >$
	$\emptyset$	$\emptyset$	$< n_6 : p >$
	$\emptyset$	$\emptyset$	$< n_8 : b >$
	$\emptyset$	$\emptyset$	$< n_9 : c >$

Procedure `R`			
node	*ARD*	*RA*	rd
e_2	$\emptyset$	$\emptyset$	$< e_2 : f >$
	$< n_1 : a >$	$\emptyset$	$< n_1 : a >$
	$< n_2 : b >$	$\emptyset$	$< n_2 : b >$
	$< n_4 : p >$	$\emptyset$	$< n_4 : p >$
	$< n_6 : p >$	$\emptyset$	$< n_6 : p >$
n_8	$\emptyset$	$\emptyset$	$< e_2 : f >$
	$< n_1 : a >$	$\emptyset$	$< n_1 : a >$
	$< n_2 : b >$	$\emptyset$	$< n_2 : b >$
	$< n_4 : p >$	$\emptyset$	$< n_4 : p >$
	$< n_6 : p >$	$\emptyset$	$< n_6 : p >$
	$\emptyset$	$< *f, a >$	$< n_8 : a >$
	$\emptyset$	$< *f, b >$	$< n_8 : b >$
n_9,x_2	$\emptyset$	$\emptyset$	$< e_2 : f >$
	$< n_1 : a >$	$\emptyset$	$< n_1 : a >$
	$< n_2 : b >$	$\emptyset$	$< n_2 : b >$
	$< n_4 : p >$	$\emptyset$	$< n_4 : p >$
	$< n_6 : p >$	$\emptyset$	$< n_6 : p >$
	$\emptyset$	$< *f, a >$	$< n_8 : a >$
	$\emptyset$	$< *f, b >$	$< n_8 : b >$
	$\emptyset$	$\emptyset$	$< n_9 : c >$

This table summarizes the *reaches*(*node*, (*ARD*, *RA*), *rd*) which are *true* for the program in Figure 3.1.

Table A.5: Summary of *reaches* without *must-hold*

Procedure main				
node	*ARD*	*RA*	*RMA*	rd
n_1	$\emptyset$	$\emptyset$	$\emptyset$	$< n_1 : a >$
n_2,n_3	$\emptyset$	$\emptyset$	$\emptyset$	$< n_1 : a >$
	$\emptyset$	$\emptyset$	$\emptyset$	$< n_2 : b >$
n_4,c_1	$\emptyset$	$\emptyset$	$\emptyset$	$< n_1 : a >$
	$\emptyset$	$\emptyset$	$\emptyset$	$< n_2 : b >$
	$\emptyset$	$\emptyset$	$\emptyset$	$< n_4 : p >$
r_1,n_5	$\emptyset$	$\emptyset$	$\emptyset$	$< n_2 : b >$
	$\emptyset$	$\emptyset$	$\emptyset$	$< n_4 : p >$
	$\emptyset$	$\emptyset$	$\emptyset$	$< n_8 : a >$
	$\emptyset$	$\emptyset$	$\emptyset$	$< n_9 : c >$
n_6,c_2	$\emptyset$	$\emptyset$	$\emptyset$	$< n_1 : a >$
	$\emptyset$	$\emptyset$	$\emptyset$	$< n_2 : b >$
	$\emptyset$	$\emptyset$	$\emptyset$	$< n_6 : p >$
r_2,n_7	$\emptyset$	$\emptyset$	$\emptyset$	$< n_1 : a >$
	$\emptyset$	$\emptyset$	$\emptyset$	$< n_6 : p >$
	$\emptyset$	$\emptyset$	$\emptyset$	$< n_8 : b >$
	$\emptyset$	$\emptyset$	$\emptyset$	$< n_9 : c >$
x_1	$\emptyset$	$\emptyset$	$\emptyset$	$< n_1 : a >$
	$\emptyset$	$\emptyset$	$\emptyset$	$< n_2 : b >$
	$\emptyset$	$\emptyset$	$\emptyset$	$< n_4 : p >$
	$\emptyset$	$\emptyset$	$\emptyset$	$< n_8 : a >$
	$\emptyset$	$\emptyset$	$\emptyset$	$< n_6 : p >$
	$\emptyset$	$\emptyset$	$\emptyset$	$< n_8 : b >$
	$\emptyset$	$\emptyset$	$\emptyset$	$< n_9 : c >$

This table summarizes the *reaches*(*node*, (ARD, RA, RMA), *rd*) which are *true* for the procedure `main` in Figure 3.1.

Table A.6: Summary of *reaches* with *must-hold* - I

Procedure R				
node	ARD	RA	RMA	rd
e_2	$\emptyset$	$\emptyset$	RMA_{c_1},RMA_{c_2}	$< e_2 : f >$
	$< n_1 : a >$	$\emptyset$	RMA_{c_1},RMA_{c_2}	$< n_1 : a >$
	$< n_2 : b >$	$\emptyset$	RMA_{c_1},RMA_{c_2}	$< n_2 : b >$
	$< n_4 : p >$	$\emptyset$	RMA_{c_1}	$< n_4 : p >$
	$< n_6 : p >$	$\emptyset$	RMA_{c_2}	$< n_6 : p >$
n_8	$\emptyset$	$\emptyset$	RMA_{c_1},RMA_{c_2}	$< e_2 : f >$
	$< n_1 : a >$	$\emptyset$	RMA_{c_2}	$< n_1 : a >$
	$< n_2 : b >$	$\emptyset$	RMA_{c_1}	$< n_2 : b >$
	$< n_4 : p >$	$\emptyset$	RMA_{c_1}	$< n_4 : p >$
	$< n_6 : p >$	$\emptyset$	RMA_{c_2}	$< n_6 : p >$
	$\emptyset$	$< *f, a >$	RMA_{c_1},RMA_{c_2}	$< n_8 : a >$
	$\emptyset$	$< *f, b >$	RMA_{c_1},RMA_{c_2}	$< n_8 : b >$
n_9,x_2	$\emptyset$	$\emptyset$	RMA_{c_1},RMA_{c_2}	$< e_2 : f >$
	$< n_1 : a >$	$\emptyset$	RMA_{c_2}	$< n_1 : a >$
	$< n_2 : b >$	$\emptyset$	RMA_{c_1}	$< n_2 : b >$
	$< n_4 : p >$	$\emptyset$	RMA_{c_1}	$< n_4 : p >$
	$< n_6 : p >$	$\emptyset$	RMA_{c_2}	$< n_6 : p >$
	$\emptyset$	$< *f, a >$	RMA_{c_1},RMA_{c_2}	$< n_8 : a >$
	$\emptyset$	$< *f, b >$	RMA_{c_1},RMA_{c_2}	$< n_8 : b >$
	$\emptyset$	$\emptyset$	RMA_{c_1},RMA_{c_2}	$< n_9 : c >$

This table summarizes the *reaches*(*node*, (ARD, RA, RMA), rd) which are *true* for the procedure R in Figure 3.1. To make the summary more readable, define:

- $RMA_{c_1} = \left\{ < *p, a >, < *p, *f >, < *f, a > \right\}$
- $RMA_{c_2} = \left\{ < *p, b >, < *p, *f >, < *f, b > \right\}$

Table A.7: Summary of *reaches* with *must-hold* - II

Procedure `main`	
node	rdtop(node)
n_2	$\{ < n_1 : a > \}$
n_3,n_4	$\{ < n_1 : a >, < n_2 : b > \}$
c_1	$\{ < n_1 : a >, < n_2 : b >, < n_4 : p > \}$
r_1,n_5	$\left\{ \begin{array}{c} < n_1 : a >, < n_2 : b >, < n_4 : p >, < n_8 : a >, \\ < n_9 : c > \end{array} \right\}$
n_6	$\{ < n_1 : a >, < n_2 : b > \}$
c_2	$\{ < n_1 : a >, < n_2 : b >, < n_6 : p > \}$
r_2,n_7	$\left\{ \begin{array}{c} < n_1 : a >, < n_2 : b >, < n_6 : p >, < n_8 : b >, \\ < n_9 : c > \end{array} \right\}$
x_1	$\left\{ \begin{array}{c} < n_1 : a >, < n_2 : b >, < n_4 : p >, < n_6 : p >, \\ < n_8 : a >, < n_8 : b >, < n_9 : c > \end{array} \right\}$

This table summarizes the *rdtop* solution for procedure `main` in Figure 3.1.

Table A.8: Summary of *rdtop* - I (computed from *reaches* without *must-hold*)

Procedure `R`	
node	rdtop(node)
e_2,n_8	$\left\{ \begin{array}{c} < e_2 : f >, < n_1 : a >, < n_2 : b >, \\ < n_4 : p >, < n_6 : p > \end{array} \right\}$
n_9	$\left\{ \begin{array}{c} < e_2 : f >, < n_1 : a >, < n_2 : b >, \\ < n_4 : p >, < n_6 : p > < n_8 : a >, \\ < n_8 : b > \end{array} \right\}$
x_2	$\left\{ \begin{array}{c} < e_2 : f >, < n_1 : a >, < n_2 : b >, \\ < n_4 : p >, < n_6 : p > < n_8 : a >, \\ < n_8 : b >, < n_9 : c > \end{array} \right\}$

This table summarizes the *rdtop* solution for procedure `R` in Figure 3.1.

Table A.9: Summary of *rdtop* - II (computed from *reaches* without *must-hold*)

Procedure `main`	
node	rdtop(node)
n_2	$\{ < n_1 : a > \}$
n_3,n_4	$\{ < n_1 : a >, < n_2 : b > \}$
c_1	$\{ < n_1 : a >, < n_2 : b >, < n_4 : p > \}$
r_1,n_5	$\{ < n_2 : b >, < n_4 : p >, < n_8 : a >, < n_9 : c > \}$
n_6	$\{ < n_1 : a >, < n_2 : b > \}$
c_2	$\{ < n_1 : a >, < n_2 : b >, < n_6 : p > \}$
r_2,n_7	$\{ < n_1 : a > < n_6 : p >, < n_8 : b >, < n_9 : c > \}$
x_1	$\left\{ \begin{array}{c} < n_1 : a >, < n_2 : b >, < n_4 : p >, < n_6 : p >, \\ < n_8 : a >, < n_8 : b >, < n_9 : c > \end{array} \right\}$

This table summarizes the *rdtop* solution for procedure `main` in Figure 3.1.

Table A.10: Summary of *rdtop* - I (computed from *reaches* with *must-hold*)

Procedure `R`	
node	rdtop(node)
e_2,n_8	$\left\{ \begin{array}{c} < e_2 : f >, < n_1 : a >, < n_2 : b >, \\ < n_4 : p >, < n_6 : p > \end{array} \right\}$
n_9	$\left\{ \begin{array}{c} < e_2 : f >, < n_1 : a >, < n_2 : b >, \\ < n_4 : p >, < n_6 : p > < n_8 : a >, \\ < n_8 : b > \end{array} \right\}$
x_2	$\left\{ \begin{array}{c} < e_2 : f >, < n_1 : a >, < n_2 : b >, \\ < n_4 : p >, < n_6 : p > < n_8 : a >, \\ < n_8 : b >, < n_9 : c > \end{array} \right\}$

This table summarizes the *rdtop* solution for procedure `R` in Figure 3.1.

Table A.11: Summary of *rdtop* - II (computed from *reaches* with *must-hold*)

Def-use without must-hold
$\ll < n_1 : a >, n_3 \gg$
$\ll < n_4 : p >, c_1 \gg$
$\ll < n_6 : p >, c_2 \gg$
$\ll < e_2 : f >, n_8 \gg$
$\ll < n_4 : p >, n_5 \gg$
$\ll < n_8 : a >, n_5 \gg$
$\ll < n_8 : b >, n_7 \gg$
$\ll < n_1 : a >, n_5 \gg$
$\ll < n_2 : b >, n_7 \gg$

Def-use with must-hold
$\ll < n_1 : a >, n_3 \gg$
$\ll < n_4 : p >, c_1 \gg$
$\ll < n_6 : p >, c_2 \gg$
$\ll < e_2 : f >, n_8 \gg$
$\ll < n_4 : p >, n_5 \gg$
$\ll < n_8 : a >, n_5 \gg$
$\ll < n_8 : b >, n_7 \gg$

This table is the def-use solution for the program in Figure 3.1.

Table A.12: Summary of def-use associations

Appendix B

Compile Time Analysis of C^{++}: Monotone Data Flow Frameworks

This chapter represents the first comprehensive effort to fit a practical approximation algorithm for an intractable interprocedural data flow analysis problem involving pointers in the classical model of monotone data flow frameworks. Previous researches have proposed general theoretical frameworks to solve interprocedural data flow problems precisely [CC77, SP81]. However, they have not led to corresponding practical implementations. Others have proposed data flow frameworks for interprocedural problems with precise, polynomial-time solution procedures [KRS94, Lan92a, RHS95]. We present an algorithm for approximation formulation of interprocedural type determination using a monotone data flow framework based on a finite multisource lattice [MMR95]. We show that the algorithm calculates an approximate and safe solution for the precise type determination problem, and the solution is identical to that computed by the algorithm presented in Section 5.3. Further, we outline a similar framework for the combined problem of type determination and aliasing. We believe that practical approximation algorithms for interprocedural problems using conditional analysis approach can be presented in similar monotone data flow frameworks, providing the algorithms a theoretical foundation to reason about properties such as safety and correctness.

In the following discussion, we assume the same restrictions on language constructs as mentioned in Chapter 1 (see p. 5).

B.1 Type Determination for Single Level Pointers

We define the following sets:

$\mathcal{X}$: set of exit nodes of ICFG with each node in the set indexed by a unique number between 1 and $|\mathcal{X}|$. Let $\tilde{x} = |\mathcal{X}|$. Let d_x be the index of exit node x.

$\mathcal{EN}$: set of entry nodes of ICFG. Since there is a unique *exit* for each *entry*, each node e in $\mathcal{EN}$ is indexed by the same number d_e, between 1 and $\tilde{x}$, as its corresponding *exit*.

$\mathcal{T}$: Set of types in the program.

$\mathcal{LC}_n$: set of pointer-typed object names derived from local variables of the function containing ICFG node n.

$\mathcal{GB}$: set of pointer-typed object names derived from global variables in the program (including heap variables and static members of classes).

$\mathcal{P}$: Set of object names which have a pointer type. Clearly, $\mathcal{P} = \mathcal{GB} \cup (\bigcup_{e \in \mathcal{EN}} \mathcal{LC}_e)$.

$\mathcal{PT} = (\mathcal{P} \times \mathcal{T})$: set of all possible pointer-type pairs (e.g., $<p \Rightarrow C>$).

$$\mathcal{RT}_{c \to e} = \left\{ <rec \Rightarrow T> \;\middle|\; \begin{array}{l} rec \in \mathcal{P} \text{ is the receiver of call node } c \text{, and} \\ c \text{ calls the method with entry node } e \text{ if } rec \\ \text{points to an object of } T \in \mathcal{T} \text{ at } c \end{array} \right\}.$$

$\mathcal{PT}_{\emptyset} = (\mathcal{PT} \cup \{\emptyset\})$: set of all possible assumptions at entry nodes.

$\mathcal{S} = (\mathcal{PT}_{\emptyset} \times \mathcal{PT})$: set of all (*assumption*, *pointer-type*) tuples.

$\mathcal{S}^{\tilde{x}}$: $(\tilde{x}+1)$-ary cross product of $\mathcal{S}$.

$\emptyset^{\tilde{x}}$: $(\tilde{x}+1)$-ary tuple of $\emptyset$s.

Monotone data flow framework: To solve the problem of type determination using the approximation formulation (see Section 5.2), we define an instance of a monotone data flow framework $D = (ICFG_D, L, F, M)$, where

1. $ICFG_D = (\mathcal{N}, \mathcal{E}_D, \rho)$ is a program representation similar to the ICFG described in Section 2.1. $\mathcal{N}$ is the set of nodes, $\mathcal{E}_D$ is the set of edges and ρ is the entry node of `main`.

2. $L = (2^{\mathcal{S}^{\tilde{x}}}, \mathcal{S}^{\tilde{x}}, \emptyset^{\tilde{x}}, \supseteq^{\tilde{x}}, \overset{\tilde{x}}{\cup})$ is a finite meet semilattice with poset $2^{\mathcal{S}^{\tilde{x}}}$, two special elements $\mathcal{S}^{\tilde{x}}$ and $\emptyset^{\tilde{x}}$, a reflexive partial order $\supseteq^{\tilde{x}}$, and a meet operator $\overset{\tilde{x}}{\cup}$.

3. F is a closed subset of monotone operations from $2^{\mathcal{S}^{\tilde{x}}} \rightarrow 2^{\mathcal{S}^{\tilde{x}}}$.

4. $M : \mathcal{E}_D \rightarrow F$ is a function which maps each edge in $\mathcal{E}_D$ to an *edge function* in F.

We describe the above concepts in detail as follows.

The program representation $ICFG_D$ defined for the above framework is identical to the ICFG in Section 2.1 with the *exit*-to-*return* edges in $\mathcal{E}$ replaced by the following two types of edges.

- $\ll exit, call \gg$ is an edge from *exit* to *call* if the function containing *exit* is possibly invoked from *call*.

- $\ll call, return \gg$ is an edge from *call* to its corresponding *return*.

Note that these edges do not represent any execution flow, but are introduced in order to define appropriate edge functions accounting for interprocedural data flow from *exit* to *return*. The significance of these special edges will become clear when we describe the edge functions.

Each node in $\mathcal{N}$ is associated with a lattice element $\in 2^{\mathcal{S}^{\tilde{x}}}$ (i.e., a subset of $\mathcal{S}^{\tilde{x}}$). Let the lattice element associated with node n be $S_n = [S_n(0), \ldots, S_n(\tilde{x})]$. In this framework, which serves as the basis of our approximation formulation, $(APT, PT) \in S_n(0)$ if the following is true.

> There exists an approximate conditionally consistent path to n from the entry node e of the function containing n on which PT holds assuming APT holds at e, **and** there exists an approximate consistent path to e on which APT holds.

The other components of the lattice elements (i.e., $S_n(1)$ through $S_n(\tilde{x})$) play a crucial role in the propagation of information from an exit node to corresponding return node(s). They are non-$\emptyset$ only at the call nodes, and are used by the edge function associated with the special interprocedural edges introduced above. The edge functions representing interprocedural propagation from a call node to entry node(s) and the edge functions representing intraprocedural propagation effectively ignore these components. The use of this multi-component lattice enables us to fit our formulation in the classical definition of a monotone data flow framework [KU77]. By employing the well-established results of data flow analysis based on this model, we prove certain properties of the formulation, such as safety and non-distributivity.

$\mathcal{S}^{\tilde{x}}$ represents the $\perp$ (bottom) and $\emptyset^{\tilde{x}}$ represents the $\top$ (top) elements of this semi-lattice. The reflexive partial order $\supseteq^{\tilde{x}}$ is defined on $2^{\mathcal{S}^{\tilde{x}}}$ as: if S and T are two elements of $2^{\mathcal{S}^{\tilde{x}}}$, $S \supseteq^{\tilde{x}} T$ iff $S(i) \supseteq T(i)$ for all i, $0 \leq i \leq \tilde{x}$. The irreflexive relation $\supset^{\tilde{x}}$ is similarly defined. The meet operator has the following interpretation: if S and T are two elements of $2^{\mathcal{S}^{\tilde{x}}}$, $(S \overset{\tilde{x}}{\cup} T) = [S(0) \cup T(0), \ldots, S(\tilde{x}) \cup T(\tilde{x})]$.

Monotone operation space: In order to define the operation space F, we need some auxiliary functions. Since the set $\mathcal{S}$ is finite, we can assign to each subset of $\mathcal{S}$ a unique index between 0 and $|2^{\mathcal{S}}| - 1$. Let s_j represent the subset of $\mathcal{S}$ with index j, with s_0 representing $\emptyset$. Define[39],

$$\begin{aligned}
g^i_j(S) &: 2^{\mathcal{S}^{\tilde{x}}} \rightarrow 2^{\mathcal{S}} &&= S(i) - s_j \\
g^i_{j,k}(S) &: 2^{\mathcal{S}^{\tilde{x}}} \rightarrow 2^{\mathcal{S}} &&= s_j \subseteq S(i)\ ?\ s_k : \emptyset \\
\pi_1(s) &: 2^{\mathcal{S}} \rightarrow 2^{\mathcal{PT}_\emptyset} &&= \left\{ APT \;\middle|\; (APT, PT) \in s \right\} \\
\pi_2(s) &: 2^{\mathcal{S}} \rightarrow 2^{\mathcal{PT}} &&= \left\{ PT \;\middle|\; (APT, PT) \in s \right\}
\end{aligned}$$

We inductively define a set of functions, each of which has signature $h : 2^{\mathcal{S}^{\tilde{x}}} \rightarrow 2^{\mathcal{S}}$, as follows:

$$\begin{array}{lcll}
\text{Rule 1.1:} & h & \leftarrow & g^i_j \\
\text{Rule 1.2:} & & \leftarrow & g^i_{j,k} \\
\text{Rule 1.3:} & & \leftarrow & (\pi_1 \circ h_1) \times (\pi_2 \circ h_2) \\
\text{Rule 1.4:} & & \leftarrow & h_1 \cup h_2
\end{array}$$

[39] Note that $g^i_j(S)$ can also be defined as $S(i) \cap s_{j'}$ where $s_{j'} = \mathcal{S} - s_j$.

Equipped with these auxiliary functions, we now define the operation space F to be comprised of the functions obtained by the following inductive definition. Each function in F has type $2^{\mathcal{S}^{\tilde{x}}} \rightarrow 2^{\mathcal{S}^{\tilde{x}}}$ and each component function ($h0$ through $h\tilde{x}$) in Rule 2.1 below satisfies Rule 1.

$$\begin{array}{lcll} \text{Rule 2.1:} & f & \leftarrow & [h0, \ldots, h\tilde{x}] \\ \text{Rule 2.2:} & & \leftarrow & f_1 \overset{\tilde{x}}{\cup} f_2 \\ \text{Rule 2.3:} & & \leftarrow & f_1 \circ f_2 \end{array}$$

We now show that F satisfies the following four conditions [Hec77], making it a *monotone function space* associated with L.

1. Each $f \in F$ is monotonic. That is, $(\forall f \in F)(\forall S, T \in L)[S \supseteq^{\tilde{x}} T \Rightarrow f(S) \supseteq^{\tilde{x}} f(T)]$: We prove this by structural induction, first on the formation of each component function h, and then on the formation of f.

   basis: By inspection, g^i_j and $g^i_{j,k}$ are monotonic. Therefore each component function h defined by Rule 1.1 or Rule 1.2 is monotonic.

   induction hypothesis: Suppose h_1 and h_2 are monotonic.

   induction step: By inspection, π_1 and π_2 are monotonic. Since cross-product, functional composition and set union are monotonic, each h formed using Rule 1.3 or Rule 1.4 is monotonic.

   Therefore, each component function h is monotonic. Finally we show that the f functions are monotonic.

   basis: Using the above result, each component function h defined by Rule 1 is monotonic. Since cross-product is a monotonic operation, each f formed using Rule 2.1 is monotonic.

   induction hypothesis: Suppose f_1 and f_2 are monotonic.

   induction step: f is formed by monotonic operations (meet and composition) on f_1 and f_2 (Rules 2.2 and 2.3).

2. There exists an identity operation f_e in F. That is, $(\exists f_e \in F)(\forall S \in L)[f_e(S) = S]$: Let $f_e = [g^0_0, g^1_0, \ldots, g^{\tilde{x}}_0]$. Clearly f_e is an identity operation.

3. F is closed under meet and composition. That is, $(\forall f1, f2 \in F)[f1 \overset{\tilde{x}}{\cup} f2 \in F]$ and $(\forall f1, f2 \in F)[f1 \circ f2 \in F]$: By construction (Rules 2.2 and 2.3 above). We consider only finite ICFG paths; therefore, showing closure under infinite composition is irrelevant to our formulation. Since we are dealing with a finite domain (set $\mathcal{S}$ is finite), an infinite meet can be expressed as a meet of finitely many sets which are pairwise unrelated with respect to set inclusion. Thus, F is closed under infinite meets.

4. For each $S \in L$, there exists an $f_S \in F$ such that $S = f_S(\mathcal{S}^{\tilde{x}})$: By definition, $S = [S(0), S(1), \ldots, S(\tilde{x})]$. For each $S(i)$, we can find an index k_i such that $S(i) = s_{k_i}$. For each such S, we can define f_S to be $[g^0_{0,k_0}, g^1_{0,k_1}, \ldots, g^{\tilde{x}}_{0,k_{\tilde{x}}}]$. Clearly f_S is a function such that $f_S(\mathcal{S}^{\tilde{x}}) = S$.

Edge functions: First we define an auxiliary edge function $\textit{type-bind}_{c \to e}$ to account for the parameter bindings at call sites.

$$\textit{type-bind}_{c \to e}(\emptyset) = \left\{ \; {<}f \Rightarrow B{>} \;\middle|\; \begin{array}{l} \texttt{f} \text{ is corresponding formal for actual } \&\texttt{b} \text{ (where} \\ \texttt{b} \text{ is an object of type } \texttt{B}) \text{ or } \texttt{new B} \text{ at call } c \end{array} \right\}$$

$$\textit{type-bind}_{c \to e}({<}p \Rightarrow B{>}) = \begin{array}{l} \left\{ \; {<}p \Rightarrow B{>} \;\middle|\; p \in \mathcal{GB} \right\} \cup \\ \left\{ \; {<}f \Rightarrow B{>} \;\middle|\; \begin{array}{l} \texttt{f} \text{ is corresponding formal} \\ \text{for actual } \texttt{p} \text{ at call } c \end{array} \right\} \end{array}$$

Finally, we define the edge functions describing propagation of type information over each class of ICFG edges. In all the following cases, we first list some component functions which facilitate defining the edge function, describe these component functions intuitively, and then define the edge function using them.

1. $\ll m, n \gg$ where n is a pointer assignment with left hand side p and m is an intraprocedural predecessor. Let

$$h0(S) \;=\; S(0) - \left\{ \; (APT, {<}p \Rightarrow C{>}) \;\middle|\; APT \in \mathcal{PT}_{\emptyset} \text{ and } C \in \mathcal{T} \; \right\}$$

$h0(S)$ removes all tuples (APT, PT) where PT involves p, since p is redefined.

$$h1_{APT,\,q,\,C}(S) \;=\; \{(APT, {<}q \Rightarrow C{>})\} \subseteq S(0)\ ?\ \{(APT, {<}p \Rightarrow C{>})\} : \emptyset$$

$$h2_q(S) \;=\; \bigcup_{\substack{APT \in \mathcal{PT}_\emptyset \\ C \in \mathcal{T}}} h1_{APT,\,q,\,C}(S)$$

$h2_q(S)$ returns tuples of the form $(APT, {<}p \Rightarrow C{>})$ where $(APT, {<}q \Rightarrow C{>})$ holds at the predecessor m.

$$h4_B(S) \;=\; \{(\emptyset, {<}p \Rightarrow B{>})\}$$

$$f_{m\to n}(S) \triangleq \begin{cases} [h0(S), g^1_{0,0}(S), \ldots, g^{\tilde{x}}_{0,0}(S)] & \texttt{n} : \texttt{p = NULL} \\ [h0(S) \cup h2_q(S), g^1_{0,0}(S), \ldots, g^{\tilde{x}}_{0,0}(S)] & \texttt{n} : \texttt{p = q} \\ [h0(S) \cup h4_B(S), g^1_{0,0}(S), \ldots, g^{\tilde{x}}_{0,0}(S)] & \texttt{n} : \texttt{p = new B} \\ [h0(S) \cup h4_B(S), g^1_{0,0}(S), \ldots, g^{\tilde{x}}_{0,0}(S)] & \texttt{n} : \texttt{p = \&b}^\dagger \end{cases}$$

†b is an object of type B.

2. $\ll c, e \gg$ where c is a non-virtual call node, e is the corresponding entry node. Let

$$h0_{APT1,\,PT1}(S) \;=\; \{(APT1, PT1)\} \subseteq S(0)\ ?\ \left\{ (APT_e, APT_e) \;\middle|\; APT_e \in \textit{type-bind}_{c\to e}(PT1) \right\} : \emptyset$$

$$h1(S) \;=\; \bigcup_{\substack{APT1 \in \mathcal{PT}_\emptyset \\ PT1 \in \mathcal{PT}}} h0_{APT1,\,PT1}(S)$$

$h1(S)$ returns all tuples (APT_e, APT_e) where (i) $APT_e \in \textit{type-bind}_{c\to e}(PT1)$ and (ii) $PT1$ holds at c.

$$h2(S) \;=\; \left\{ (APT_e, APT_e) \;\middle|\; APT_e \in \textit{type-bind}_{c\to e}(\emptyset) \right\}$$

$h2(S)$ returns all tuples (APT_e, APT_e) where $APT_e \in \textit{type-bind}_{c\to e}(\emptyset)$.

$$f_{c\to e}(S) \triangleq [h1(S) \cup h2(S), g^1_{0,0}(S), \ldots, g^{\tilde{x}}_{0,0}(S)]$$

3. $\ll c, e \gg$ where c is a virtual call node with receiver rec, e is an entry node. Let

$$h0_{APT1,\,PT1,\,APT2,\,PT2}(S) \;=\; \{(APT1, PT1), (APT2, PT2)\} \subseteq S(0) \;?\; \left\{ (APT_e, APT_e) \;\middle|\; APT_e \in \textit{type-bind}_{c \to e}(PT1) \right\} : \emptyset$$

$$h1(S) \;=\; \bigcup_{\substack{APT1, APT2 \in \mathcal{PT}_\emptyset \\ PT1 \in \mathcal{PT} \\ PT2 \in \mathcal{RT}_{c \to e}}} h0_{APT1,\,PT1,\,APT2,\,PT2}(S)$$

$h1(S)$ checks if a pointer-type pair <$rec \Rightarrow E$> holds at c (i.e., $(APT,$<$rec \Rightarrow E$>$) \in S(0)$ for some APT) such that rec pointing to an object of class E makes c invoke e; if so, returns tuples (APT_e, APT_e) where (i) $APT_e \in \textit{type-bind}_{c \to e}(PT1)$ and (ii) $PT1$ holds at c.

$$h2_{APT2,\,PT2}(S) \;=\; \{(APT2, PT2)\} \subseteq S(0) \;?\; \left\{ (APT_e, APT_e) \;\middle|\; APT_e \in \textit{type-bind}_{c \to e}(\emptyset) \right\} : \emptyset$$

$$h3(S) \;=\; \bigcup_{\substack{APT2 \in \mathcal{PT}_\emptyset \\ PT2 \in \mathcal{RT}_{c \to e}}} h2_{APT2,\,PT2}(S)$$

$h3(S)$ checks if a pointer-type pair <$rec \Rightarrow E$> holds at c such that rec pointing to an object of class E makes c invoke e; if so, returns tuples (APT_e, APT_e) where $APT_e \in \textit{type-bind}_{c \to e}(\emptyset)$.

$$f_{c \to e}(S) \stackrel{\triangle}{=} [h1(S) \cup h3(S), g^{1}_{0,0}(S), \ldots, g^{\tilde{x}}_{0,0}(S)]$$

4. $\ll c, r \gg$ where c is a non-virtual call node with corresponding entry node e, r is the corresponding return node. Let

$$h0_{PT1}(S) = S(d_e) \cap (\mathit{type\text{-}bind}_{c \to e}(PT1) \times (\mathcal{GB} \times \mathcal{T}))$$

$$h1_{APT1,\,PT1}(S) = \{(APT1, PT1)\} \subseteq S(0)\ ?\ \{(APT1, PT1)\} : \emptyset$$

$$h2_{PT1}(S) = \bigcup_{APT1 \in \mathcal{PT}_\emptyset} h1_{APT1,\,PT1}(S)$$

$$h3(S) = \bigcup_{PT1 \in \mathcal{PT}} \pi_1(h2_{PT1}(S)) \times \pi_2(h0_{PT1}(S))$$

$h3(S)$ returns tuples $(APT1, PT)$ such that (i) $PT1$ holds at c with assumption $APT1$, (ii) $APT_e \in \mathit{type\text{-}bind}_{c \to e}(\mathrm{PT1})$, (iii) PT holds at corresponding exit node of e with assumption APT_e, and (iv) PT involves a global object name.

$$h4(S) = S(d_e) \cap ((\mathit{type\text{-}bind}_{c \to e}(\emptyset) \cup \{\emptyset\}) \times (\mathcal{GB} \times \mathcal{T}))$$

$$h5(S) = \{\emptyset\} \times \pi_2(h4(S))$$

$h5(S)$ returns tuples $(\emptyset, PT)$ such that (i) $APT_e \in \mathit{type\text{-}bind}_{c \to e}(\emptyset)$ or $APT_e = \emptyset$, (ii) PT holds at corresponding exit node of e with assumption APT_e, and (iii) PT involves a global object name.

$$h6(S) = \bigcup_{\substack{APT1 \in \mathcal{PT}_\emptyset \\ PT1 \in \mathcal{LC}_c \times \mathcal{T}}} h1_{APT1,\,PT1}(S)$$

$h6(S)$ returns tuples $(APT1, PT1) \in S(0)$ where $PT1$ involves an object name local to the function containing call node c.

$$f_{c \to r}(S) \triangleq [h3(S) \cup h5(S) \cup h6(S), g^1_{0,0}(S), \ldots, g^{\tilde{x}}_{0,0}(S)]$$

5. $\ll c, r \gg$ where c is a virtual call node with receiver rec, r is the corresponding return node, and $e \in \mathcal{EN}$. Let

$$h0_{e,\,PT1}(S) \quad = \quad S(d_e) \cap (\mathit{type\text{-}bind}_{c \to e}(PT1) \times (\mathcal{GB} \times \mathcal{T}))$$

$$h1_{APT1,\,PT1,\,APT2,\,PT2}(S) \quad = \quad \{(APT1, PT1), (APT2, PT2)\} \subseteq S(0) \;?\; \{(APT1, PT1)\} : \emptyset$$

$$h2_{e,\,PT1}(S) \quad = \quad \bigcup_{\substack{APT1,\, APT2 \in \mathcal{PT}_\emptyset \\ PT2 \in \mathcal{RT}_{c \to e}}} h1_{APT1,\,PT1,\,APT2,\,PT2}(S)$$

$$h3_e(S) \quad = \quad \bigcup_{PT1 \in \mathcal{PT}} \pi_1(h2_{e,\,PT1}(S)) \times \pi_2(h0_{e,\,PT1}(S))$$

$h3_e(S)$ checks if $S(0)$ contains a tuple $(APT2, <rec \Rightarrow E>)$ such that rec pointing to an object of class E results in c calling e; if so, returns tuples $(APT1, PT)$ such that (i) $PT1$ holds at c with assumption $APT1$, (ii) $APT_e \in \mathit{type\text{-}bind}_{c \to e}(PT1)$, (iii) PT holds at corresponding exit node of e with assumption APT_e, and (iv) PT involves a global object name.

$$h4_e(S) \quad = \quad S(d_e) \cap ((\mathit{type\text{-}bind}_{c \to e}(\emptyset) \cup \{\emptyset\}) \times (\mathcal{GB} \times \mathcal{T}))$$

$$h5_{APT2,\,PT2}(S) \quad = \quad \{(APT2, PT2)\} \subseteq S(0) \;?\; \{(\emptyset, PT2)\} : \emptyset$$

$$h6_e(S) \quad = \quad \bigcup_{\substack{APT2 \in \mathcal{PT}_\emptyset \\ PT2 \in \mathcal{RT}_{c \to e}}} h5_{APT2,\,PT2}(S)$$

$$h7_e(S) \quad = \quad \pi_1(h6_e(S)) \times \pi_2(h4_e(S))$$

$h7_e(S)$ checks if $S(0)$ contains a tuple $(APT2, <rec \Rightarrow E>)$ such that rec pointing to an object of class E results in c calling e; if so, returns tuples $(\emptyset, PT)$ such that (i) $APT_e \in \mathit{type\text{-}bind}_{c \to e}(\emptyset)$ or $APT_e = \emptyset$, (ii) PT holds at corresponding exit node of e with assumption APT_e, and (iii) PT involves a global object name.

$$h8(S) \quad = \bigcup_{\substack{APT1,\, APT2 \,\in\, \mathcal{PT}_\emptyset \\ PT1 \,\in\, \mathcal{LC}_c \times \mathcal{T} \\ PT2 \,\in\, \mathcal{PT}}} h1_{APT1,\, PT1,\, APT2,\, PT2}(S)$$

$h8$ returns tuples $(APT1, PT1) \in S(0)$ where $PT1$ involves an object name local to the function containing call node c.

$$h9(S) \quad = \quad \bigcup_{e \,\in\, \mathcal{EN}} (h3_e(S) \cup h7_e(S)) \cup h8(S)$$

$$f_{c \rightarrow r}(S) \stackrel{\triangle}{=} [h9(S), g^1_{0,0}(S), \ldots, g^{\tilde{x}}_{0,0}(S)]$$

6. $\ll x, c \gg$ where x is an exit node and c is a call node. Let

$$h^i(S) = \begin{cases} g^0_0(S) & i = d_x \\ g^0_{0,0}(S) & \text{otherwise} \end{cases}$$

$$f_{x \rightarrow c}(S) \stackrel{\triangle}{=} [g^0_{0,0}(S), h^1(S), \ldots, h^{\tilde{x}}(S)]$$

Effectively, $S_x(0)$ of exit node x is propagated to $S_c(d_x)$ of the call site c. We saw earlier how $f_{c \rightarrow r}$ uses $S_c(0)$ and $S_c(d_x)$ at call node c to propagate type information to corresponding return node r.

7. $\ll m, n \gg$ where m is not an exit node and n is not an entry node, return node or pointer assignment. This is the case of simple intraprocedural propagation, since the information flows unchanged on an intraprocedural edge to a node which is not a pointer assignment.

$$f_{m \rightarrow n}(S) \stackrel{\triangle}{=} [g^0_0(S), g^0_{0,0}(S), \ldots, g^0_{0,0}(S)]$$

Theoretical properties of the framework: To demonstrate the difficulty of type determination even in the approximation formulation, we evaluate the function space

F with respect to two properties *viz., distributivity* and *k-boundedness*. A distributive function space guarantees that the iteratively obtained Maximal Fixed Point (MFP) solution is identical to the ideally sought Meet-Over-all-Paths (MOP) solution of the problem. k-boundedness of the function space guarantees that the lattice values being carried around a loop in the source code stabilize in at most k iterations of the loop. Unfortunately our function space possesses neither of these properties, providing a measure of inherent difficulty of the approximation formulation.

The closed monotone function space F associated with lattice L is distributive iff

$$(\forall f \in F)(\forall S1, S2 \in L) : [f(S1 \overset{\tilde{x}}{\cup} S2) = f(S1) \overset{\tilde{x}}{\cup} f(S2)]$$

Theorem B.1.1 *F is not a distributive function space.*

Let e be the entry node of a virtual function `f` and c be a virtual call node of the form "`rec→f( )`", such that c invokes `f` when the receiver `rec` points to an object of class `E`. Let p be a global object name and rec be local to the calling function. Assume

$$S1(0) = \{(\emptyset, <p \Rightarrow C>)\} \text{ and } S2(0) = \{(\emptyset, <rec \Rightarrow E>)\}$$

while each $S1(i \neq 0)$ and $S2(i \neq 0)$ is $\emptyset$. Clearly, $\textit{type-bind}_{c \to e}(<p \Rightarrow C>) = \{<p \Rightarrow C>\}$ and $\textit{type-bind}_{c \to e}(<rec \Rightarrow E>) = \{<this \Rightarrow E>\}$. Using the edge function for the interprocedural edge $\ll c, e \gg$,

$$f_{c \to e}(S1) = \emptyset \text{ and } f_{c \to e}(S2) = [\{(<this \Rightarrow E>, <this \Rightarrow E>)\}, \emptyset, \ldots, \emptyset]$$

whereas

$$f_{c \to e}(S1 \overset{\tilde{x}}{\cup} S2) = [\{(<p \Rightarrow C>, <p \Rightarrow C>), (<this \Rightarrow E>, <this \Rightarrow E>)\}, \emptyset, \ldots, \emptyset]$$

Therefore,

$$f_{c \to e}(S1 \overset{\tilde{x}}{\cup} S2) \neq f_{c \to e}(S1) \overset{\tilde{x}}{\cup} f_{c \to e}(S2)$$

□

Let f^i represent i-composition of the function $f \in F$. For example, $f^0(S) = S$, $f^1(S) = f(S)$ and $f^2(S) = f \circ f(S)$. Let $f^{[k]} = \overset{\tilde{x}}{\cup}{}_{i=0}^{k-1} f$. The monotone function space F is k-bounded iff there exists a constant k such that

$$(\forall f \in F)(\forall S \in L) : [f^{[k]}(S) \supseteq^{\tilde{x}} f^k(S)]$$

```
while (-) {
    n0   :  ...
    n1   :  p1 = p2
    n2   :  p2 = p3
    ...
    nm-1:  pm-1 = pm
    nm   :  pm = new B
}
```

Figure B.1: Example for non-k-boundedness of F

In other words, the sequence of repeated applications of any f until stabilization is at most k in length.

Theorem B.1.2 *F is not k-bounded.*

Consider the code segment in Figure B.1. Assume that all variables are initialized to $NULL$ initially. Let $f = f_{n_0 \to n_1} \circ f_{n_1 \to n_2} \circ \ldots \circ f_{n_m \to n_0}$. Each application of this function corresponds to an iteration of the `while` loop. For each $1 \leq i \leq m$, $f^{[i+1]}(\emptyset)$ adds a tuple $(\emptyset, <p_{m-i+1} \Rightarrow B>)$ to the value of $S_{n_0}(0)$ from previous iteration. For example, 1^{st} iteration adds $(\emptyset, <p_m \Rightarrow B>)$ to the solution at node n_0 and m^{th} iteration adds $(\emptyset, <p_1 \Rightarrow B>)$ to the solution at node n_0. This process stabilizes only after m iterations, when each variable eventually points to an object of class `B`. By construction, m may be arbitrarily large. Note that the monotonicity of F and the finiteness of lattice L imply that the function space is bounded. However, as we have just shown, no predetermined k exists to guarantee k-boundedness. □

Algorithm: We use a worklist algorithm to obtain the MFP solution of the system of equations defined by the edge functions. The algorithm appears in Figure B.2.

Every iteration of the `while` loop starts by removing exactly one element from the worklist. If $S_m \supset^{\tilde{x}} S$, nothing is added to the worklist, effectively shrinking the worklist in length by 1. Otherwise, for each successor n, (n, S_n) may be added to the worklist iff $S_{new} \supset^{\tilde{x}} S_n$ (i.e., only if S_n grows). Since $\mathcal{S}^{\tilde{x}}$ is finite and the edge functions are monotonic, this process will terminate at the MFP solution.

```
set worklist to empty;
for each m ∈ 𝒩 // introduction of type information
   switch (m)
       ''p = new B'' :
          S_m = [{(∅, <p ⇒ B>)}, ∅, ..., ∅];
          add (m, S_m) to worklist;
       ''p = &b'' :
          S_m = [{(∅, <p ⇒ B>)}, ∅, ..., ∅]; // b is object of type B
          add (m, S_m) to worklist;
       non-virtual call with corresponding entry e :
          S_e = [{(APT_e, APT_e) | APT_e ∈ type-bind_{m→e}(∅)}, ∅, ..., ∅];
          add (e, S_e) to worklist;
       default:  S_m = ∅;
while worklist is not empty
   remove (m, S) from worklist;
   if S_m ⊃^x̃ S continue;
   switch (m)
       call :
          for each e invocable by m
             S_new = f_{m→e}(S) ∪^x̃ S_e;
             if S_new ⊃^x̃ S_e
                S_e = S_new;
                add (e, S_e) to worklist;
          r = corresponding return node for m;
          S_new = f_{m→r}(S) ∪^x̃ S_r;
          if S_new ⊃^x̃ S_r
             S_r = S_new;
             add (r, S_r) to worklist;
       exit :
          for each call node c such that c may
                   invoke function containing m
             S_new = f_{m→c}(S) ∪^x̃ S_c;
             if S_new ⊃^x̃ S_c
                S_c = S_new;
                add (c, S_c) to worklist;
       default:
          for each n ∈ succ(m)
             S_new = f_{m→n}(S) ∪^x̃ S_n;
             if S_new ⊃^x̃ S_n
                S_n = S_new;
                add (n, S_n) to worklist;
```

Figure B.2: Type determination algorithm

Lemma B.1.1 *For a call node c, the entry node e for the function invoked by c, and a (possibly $\emptyset$) pointer-type pair PT, $type\text{-}bind_{c \to e}(PT)$ is the precise set of pointer-type pairs which hold on a consistent path $\rho\Upsilon ce$ using the fact that PT holds on $\rho\Upsilon c$.*

The pointer-type pairs which hold on $\rho\Upsilon ce$ without requiring any pairs to hold on $\rho\Upsilon c$ are exactly what is calculated by $type\text{-}bind_{c \to e}(\emptyset)$. This is the type information derived solely from the actual parameters at the call site c, and not from any predecessors of c. All circumstances which result in a set of pointer-type pairs holding on $\rho\Upsilon ce$, given that PT holds on $\rho\Upsilon c$, are exactly captured by $type\text{-}bind_{c \to e}(PT)$. If PT involves a global object name, PT itself holds on $\rho\Upsilon ce$. If $PT = <p \Rightarrow B>$ where `p` is an actual parameter of c, the corresponding formal `f` of e must point to an object of type `B` (i.e., $<f \Rightarrow B>$ must hold on $\rho\Upsilon ce$). $\Box$

Proof of safety of the algorithm: Let S_m denote the MFP solution for node m calculated by the algorithm in Figure B.2.

Theorem B.1.3 *Given an execution path $\Sigma = \Upsilon e \Phi$ such that (i) Υ originates at ρ, (ii) e is the entry node of the function containing node n, and (iii) $e\Phi$ is a balanced path to n, if $<p \Rightarrow B>$ holds on Σ then $(APT, <p \Rightarrow B>) \in S_n(0)$ for some assumption APT where APT holds on Υe.*

We prove the theorem using induction on the path length. Let k denote the length of path Σ. Note that Υ and Φ may possibly be empty paths.

basis: $k = 1$. Trivially true since no pointer-types hold at ρ.

induction hypothesis: Theorem B.1.3 holds for paths of length $j < k$.

induction step: We show that Theorem B.1.3 is true for each path Σ of length k. let m be the immediate predecessor of n on Σ. The following are all the cases which make $<p \Rightarrow B>$ hold on Σ.

1. n is pointer assignment "`p = new B`" or "`p = &b`" where `b` has type `B` : $f_{m \to n}(S_m)$ ensures that $(\emptyset, <p \Rightarrow B>) \in S_n(0)$. Clearly, $\emptyset$ always holds on Υe.

2. n is pointer assignment "p = q" and $<q \Rightarrow B>$ holds on the subpath of Σ from ρ to m : Using induction hypothesis, $(APT, <q \Rightarrow B>) \in S_m(0)$ where APT holds on Υe. $f_{m \to n}(S_m)$ ensures that $(APT, <p \Rightarrow B>) \in S_n(0)$ where APT holds on Υe.

3. m is a call node, n is the entry of function invoked by m (i.e., n is the entry node e itself, Φ is an empty path and Υ ends at m), and $<p \Rightarrow B> \in \textit{type-bind}_{m \to n}(PT1)$ for some $PT1 \neq \emptyset$ which holds on Υ (using Lemma B.1.1).

   (a) m is not a virtual call : Using induction hypothesis, $(APT1, PT1) \in S_m(0)$ for an entry assumption $APT1$. Using $f_{m \to n}(S_m)$ and $\textit{type-bind}_{m \to n}(PT1)$ (Lemma B.1.1),

   $$(<p \Rightarrow B>, <p \Rightarrow B>) \in S_n(0)$$

   where $<p \Rightarrow B>$ holds on Υe.

   (b) m is a virtual call : Since Σ is an execution path, the receiver rec of m must point to an object of appropriate type E such that m invokes n. Using induction hypothesis,

   $$\{(APT1, PT1), (APT2, <rec \Rightarrow E>)\} \subseteq S_m(0)$$

   for entry assumptions $APT2$ and $APT1$. Using $f_{m \to n}(S_m)$ and Lemma B.1.1 for $\textit{type-bind}_{m \to n}(PT1)$,

   $$(<p \Rightarrow B>, <p \Rightarrow B>) \in S_n(0)$$

   where $<p \Rightarrow B>$ holds on Υe.

4. m is a call node, n is the entry of function invoked by m (i.e., n is the entry node e itself, Φ is an empty path and Υ ends at m), and $<p \Rightarrow B> \in \textit{type-bind}_{m \to n}(\emptyset)$. By Lemma B.1.1, this case represents all parameter bindings not covered by case 3 above.

   (a) m is not a virtual call : Using $f_{m \to n}(S_m)$ and Lemma B.1.1 for correctness of $\textit{type-bind}_{m \to n}(\emptyset)$,

   $$(<p \Rightarrow B>, <p \Rightarrow B>) \in S_n(0)$$

where $<p \Rightarrow B>$ holds on Υe.

(b) m is a virtual call : Since Σ is an execution path, the receiver rec of m must point to an object of appropriate type E such that m invokes n. Using induction hypothesis, $(APT2, <rec \Rightarrow E>) \in S_m(0)$ for an entry assumption $APT2$. Using $f_{m\to n}(S_m)$ and Lemma B.1.1 for $\textit{type-bind}_{m\to n}(\emptyset)$,

$$(<p \Rightarrow B>, <p \Rightarrow B>) \in S_n(0)$$

where $<p \Rightarrow B>$ holds on Υe.

5. n is a return node with corresponding virtual or non-virtual call c on Σ, m is exit node of the function with entry node e, p is local to the function containing n, and $<p \Rightarrow B>$ holds on the subpath of Σ from ρ to c : Using induction hypothesis, $(APT1, <p \Rightarrow B>) \in S_c(0)$ for some assumption $APT1$ which holds on Υe. Using $f_{c\to n}(S_c)$,

$$(APT1, <p \Rightarrow B>) \in S_n(0)$$

where $APT1$ holds on Υe.

6. n is a return node with corresponding non-virtual call c on Σ, m is exit node of the function with entry node e, p is a global object name and $<p \Rightarrow B>$ holds on the subpath of Σ from ρ to m : Let the path Φ be represented as $\Theta c e' \Psi m$ where $e' \Psi m$ is a balanced path.

(a) Using induction hypothesis, $(APT_{e'}, <p \Rightarrow B>) \in S_m(0)$ for some $APT_{e'}$ holding on $\Upsilon e \Theta c e'$.

(b) Since $APT_{e'}$ holds on $\Upsilon e \Theta c e'$, either $APT_{e'} \in (\textit{type-bind}_{c\to e'}(\emptyset) \cup \{\emptyset\})$ or $APT_{e'} \in \textit{type-bind}_{c\to e'}(PT1)$, using Lemma B.1.1.

(c) If $APT_{e'} \in \textit{type-bind}_{c\to e'}(PT1)$, using induction hypothesis $(APT1, PT1) \in S_c(0)$, where $APT1$ holds on Υe.

Given the above, $f_{m\to c}(S_m)$ ensures that $(APT_{e'}, <p \Rightarrow B>) \in S_c(d_m)$. Subsequently, $f_{c\to n}(S_c)$ makes

$$(\emptyset, <p \Rightarrow B>) \in S_n(0) \text{ or } (APT1, <p \Rightarrow B>) \in S_n(0)$$

as

$$APT_{e'} \in (\textit{type-bind}_{c \to e'}(\emptyset) \cup \{\emptyset\}) \text{ or } APT_{e'} \in \textit{type-bind}_{c \to e'}(PT1)$$

respectively. In the former case, $\emptyset$ always holds at Υe. In the latter, we already saw that $APT1$ holds at Υe.

7. n is a return node with corresponding virtual call c on Σ, m is exit node of the function with entry node e, p is a global object name and $<p \Rightarrow B>$ holds on path Σ ending at m. Let the path Φ be represented as $\Theta c e' \Psi m$ where $e' \Psi m$ is a balanced path.

   (a) Using induction hypothesis, $(APT2, <rec \Rightarrow E>) \in S_c(0)$ for some $APT2$ holding on Υe.

   (b) Using induction hypothesis, $(APT_{e'}, <p \Rightarrow B>) \in S_m(0)$ for some $APT_{e'}$ holding on $\Upsilon e \Theta c e'$.

   (c) Since $APT_{e'}$ holds on $\Upsilon e \Theta c e'$, either $APT_{e'} \in (\textit{type-bind}_{c \to e'}(\emptyset) \cup \{\emptyset\})$ or $APT_{e'} \in \textit{type-bind}_{c \to e'}(PT1)$, using Lemma B.1.1.

   (d) If $APT_{e'} \in \textit{type-bind}_{c \to e'}(PT1)$, using induction hypothesis $(APT1, PT1) \in S_c(0)$, where $APT1$ holds on Υe.

   Given the above, $f_{m \to c}(S_m)$ ensures that $(APT_{e'}, <p \Rightarrow B>) \in S_c(d_m)$. Subsequently, $f_{c \to n}(S_c)$ makes

$$(\emptyset, <p \Rightarrow B>) \in S_n(0) \text{ or } (APT1, <p \Rightarrow B>) \in S_n(0)$$

   as

$$APT_{e'} \in (\textit{type-bind}_{c \to e'}(\emptyset) \cup \{\emptyset\}) \text{ or } APT_{e'} \in \textit{type-bind}_{c \to e'}(PT1)$$

   respectively. In the former case, $\emptyset$ always holds at Υe. In the latter, we already saw that $APT1$ holds at Υe.

8. n is none of the above and $<p \Rightarrow B>$ holds on the subpath of Σ from ρ to m : Using induction hypothesis, $(APT, <p \Rightarrow B>) \in S_m(0)$ where APT holds on Υe. $f_{m \to n}(S_m)$ ensures that

$$(APT, <p \Rightarrow B>) \in S_n(0)$$

where APT holds on Υe.

We have accounted for all situations which result in $<p \Rightarrow B>$ holding at Σ of length k, and hence proved Theorem B.1.3. □

In Section 5.3.2, we calculated type information of finer granularity (in terms of the *points-to-type* predicates) than the sets S_n associated with each ICFG node. We now show that the fixed point solutions calculated by the algorithm in Section 5.3.2 (see Figures 5.4 on p. 63 and 5.5 on p. 64 for a high level view of the algorithm) and the algorithm in Figure B.2 are identical. As we have already seen, once (APT, PT) is added to $S_n(0)$, it is never taken out of the set, and once *points-to-type*(n, APT, PT) becomes *true*, it stays *true*, preserving *monotonicity* of information calculated at each ICFG node by both algorithms. Therefore, the equality of the fixed point solution is tantamount to showing that for each ICFG node n, *points-to-type*(n, APT, PT) = *true* iff $(APT, PT) \in S_n(0)$ during some iteration of the algorithms. Let's call the two algorithms $\mathcal{A}_{PTT}$ and $\mathcal{A}_S$ respectively. Each iteration of the `while` loop in Figures 5.5 and B.2 represents an *iteration* of the respective algorithm. $\mathcal{A}_{PTT}$ and $\mathcal{A}_S$ only differ in the order of evaluation at ICFG nodes, owing to the following differences in the way information is processed by them.

1. In $\mathcal{A}_S$, an edge function $f_{m \to n}$ maps the entire set S_m at an ICFG node m to its successor n. This would be equivalent to propagating all the *points-to-type* predicates at node m through successor n in consecutive iterations of $\mathcal{A}_{PTT}$. Since $\mathcal{A}_{PTT}$ actually propagates one *points-to-type* predicate at m through n at a time, the iterations propagating these predicates are staggered rather than necessarily being consecutive.

2. The $\mathcal{A}_S$ edge function $f_{x \to c}$ from exit node x to call node c propagates, in one iteration, the set $S_x(0)$ of x to the set $S_c(d_x)$ of c (where d_x is the index of exit node x). Effectively, this action copies the information at x in the lattice value for c. In a later iteration, the edge function $f_{c \to r}$ uses the type information from $S_c(0)$ and $S_c(d_x)$ together to derive information at the corresponding return node r. On the other hand, $\mathcal{A}_{PTT}$ propagates a predicate from c to r directly, using

appropriate predicates at x in a *single* iteration.

Nevertheless, given that the information grows monotonically, the order of evaluation does not matter, and the two algorithms converge to the same fixed point. We now informally prove the equality of the solutions of $\mathcal{A}_{PTT}$ and $\mathcal{A}_S$.

In order to compare the solutions of the two algorithms, we define the following quantities.

$$PTT_n = \left\{ (APT, PT) \;\middle|\; \begin{array}{l} \textit{points-to-type}(n,\ APT,\ PT) \text{ belongs to} \\ \text{the MFP solution of } \mathcal{A}_{PTT} \text{ at node } n \end{array} \right\}$$

$$PTT_n^i = \left\{ (APT, PT) \;\middle|\; \begin{array}{l} \textit{points-to-type}(n,\ APT,\ PT) \text{ is } \textit{true} \text{ after} \\ \text{the } i^{th} \text{ iteration of } \mathcal{A}_{PTT} \end{array} \right\}$$

$$\begin{aligned} S_n &= \text{MFP solution of } \mathcal{A}_S \text{ at node } n \\ S_n^i &= \text{solution of } \mathcal{A}_S \text{ at node } n \text{ after the } i^{th} \text{ iteration} \end{aligned}$$

Let PTT_n^0 and S_n^0 be defined as the solutions for node n just after the introduction phase of the respective algorithms.

Lemma B.1.2 *For each node $n \in \mathcal{N}$, $PTT_n^0 = S_n^0(0)$.*

By inspection. See the definition of *type-bind*$_{c\to e}$ on p. 139, Figure B.2 and Figure 5.4. □

Lemma B.1.3 *For each node $n \in \mathcal{N}$, any iteration j*

$$PTT_n^j \subseteq PTT_n \text{ and } S_n^j(0) \subseteq S_n(0)$$

Obvious, using monotonicity of operations in both the algorithms. □

Lemma B.1.4 *For each node $n \in \mathcal{N}$, any iteration j of $\mathcal{A}_{PTT}$*

$$PTT_n^j \subseteq S_n(0)$$

We prove Lemma B.1.4 by induction on the number of iterations of $\mathcal{A}_{PTT}$.

basis: Let $j = 0$. For each node n, using Lemma B.1.2

$$PTT_n^0 = S_n^0(0)$$

Since $S_n^0(0) \subseteq S_n(0)$, using Lemma B.1.3

$$PTT_n^0 \subseteq S_n(0)$$

induction hypothesis: Let Lemma B.1.4 be true for $i < j$.

induction step: For those nodes whose solutions are not affected by the j^{th} iteration, $PTT_n^j \subseteq S_n(0)$ using induction hypothesis. We show that for each node n whose solution changes (i.e., grows) after the j^{th} iteration, $PTT_n^j \subseteq S_n(0)$. We achieve it by showing that for each new (APT, PT) added to PTT_n^j,

$$(APT, PT) \in S_n(0)$$

We explore all cases which add such a tuple to PTT_n^j during the j^{th} iteration. In the following cases, without loss of generality, let $<rec \Rightarrow E>$ be the pointer-type pair which makes a virtual call m with receiver rec invoke the function with entry e.

1. n is not an entry or return node, and the predicate *points-to-type*(n, APT, PT) is added by **type-implies-type-through-other**($m, APT, PT1$) at a predecessor m (using **case 1**, **case 2** or **case 3**) : Using induction hypothesis,

   $$(APT, PT1) \in S_m(0)$$

   The edge function $f_{m \to n}(S_m)$ uses the tuple $(APT, PT1)$ in $S_m(0)$ resulting in

   $$(APT, PT) \in S_n(0)$$

2. *points-to-type*(e, APT_e, APT_e) is added by **type-implies-type-from-call** at an entry node e. This may happen in one of the following circumstances.

   (a) m is a non-virtual call and *points-to-type*(m, $APT1$, $PT1$) contributes to this addition (using **case 1** of **type-implies-type-from-call**) : Using induction hypothesis,

   $$(APT1, PT1) \in S_m(0)$$

The edge function $f_{m \to e}(S_m)$ uses *type-bind*$_{m \to e}(PT1)$ and the above tuple in $S_m(0)$ to ensure that

$$(APT_e, APT_e) \in S_e(0)$$

(b) m is a virtual call, *points-to-type*$(m,\ APT1,\ PT1)$ and *points-to-type*$(m,\ APT2,\ {<}rec \Rightarrow E{>})$, where $PT1 \neq {<}rec \Rightarrow E{>}$, together contribute to this addition (using **case 2.1** or **case 2.2** of **type-implies-type-from-call**) : Using induction hypothesis,

$$\{(APT1, PT1), (APT2, {<}rec \Rightarrow E{>})\} \subseteq S_m(0)$$

The edge function $f_{m \to e}(S_m)$ uses *type-bind*$_{m \to e}(PT1)$ and the above subset of $S_m(0)$ to ensure that

$$(APT_e, APT_e) \in S_e(0)$$

(c) m is a virtual call and *points-to-type*$(m,\ APT2,\ {<}rec \Rightarrow E{>})$ contributes to this addition (using **case 2.1** of **type-implies-type-from-call**) : Using induction hypothesis,

$$(APT2, {<}rec \Rightarrow E{>}) \in S_m(0)$$

The edge function $f_{m \to e}(S_m)$ uses *type-bind*$_{m \to e}(\emptyset)$ and the above tuple in $S_m(0)$ resulting in

$$(APT_e, APT_e) \in S_e(0)$$

3. **type-implies-type-from-call**(m, APT, PT) adds *points-to-type*$(r,\ APT,\ PT)$ to the solution at the corresponding return node r where PT involves a local variable of the calling function and the call node m is virtual or non-virtual : By induction hypothesis

$$(APT, PT) \in S_m(0)$$

$f_{m \to r}(S_m)$ uses this tuple in $S_m(0)$, resulting in

$$(APT, PT) \in S_r(0)$$

4. *points-to-type*(r, APT, PT) is added to the solution at a return node r of non-virtual call m by either **type-implies-type-from-call** or **type-implies-type-from-exit**, where PT involves a global variable. This may happen using one of the following cases. Let e be the entry of function invoked by m and x be the corresponding exit node for e.

   (a) *points-to-type*(m, APT, $PT1$) exist at call node m, $PT1$ imposes APT_e at entry node e, and *points-to-type*(x, APT_e, PT) is present at x : Using induction hypothesis,

   $$(APT, PT1) \in S_m(0)$$

   Also, $APT_e \in \textit{type-bind}_{m \to e}(PT1)$ using Lemma B.1.1. Applying induction hypothesis again,

   $$(APT_e, PT) \in S_x(0)$$

   Using $f_{x \to m}(S_x)$,

   $$(APT_e, PT) \in S_m(d_x)$$

   Using $f_{m \to r}(S_m)$,

   $$(APT, PT) \in S_r(0)$$

   (b) APT_e is either $\emptyset$ or imposed at entry node e simply because of actual-formal parameter bindings, and *points-to-type*(x, APT_e, PT) is present at x : $APT_e \in \textit{type-bind}_{m \to e}(\emptyset)$ using Lemma B.1.1 or $APT_e = \emptyset$. Applying induction hypothesis,

   $$(APT_e, PT) \in S_x(0)$$

   Using $f_{x \to m}(S_x)$,

   $$(APT_e, PT) \in S_m(d_x)$$

   Using $f_{m \to r}(S_m)$,

   $$(\emptyset, PT) \in S_r(0)$$

5. *points-to-type*(r, APT, PT) is added to the solution at a return node r of virtual call m by either **type-implies-type-from-call** or **type-implies-type-from-exit**, where PT involves a global variable. This may happen using one of the

following cases. Let e be the entry of function invoked by m and x be the corresponding exit node for e.

(a) *points-to-type*(m, $APT2$, $<rec \Rightarrow E>$) and *points-to-type*(m, APT, $PT1$) exist at call node m, $PT1$ imposes APT_e at e, and *points-to-type*(x, APT_e, PT) is present at x : Using induction hypothesis,

$$\{(APT, PT1), (APT2, <rec \Rightarrow E>)\} \subseteq S_m(0)$$

Also, $APT_e \in$ *type-bind*$_{m \to e}(PT1)$ using Lemma B.1.1. Applying induction hypothesis again,

$$(APT_e, PT) \in S_x(0)$$

Using $f_{x \to m}(S_x)$,

$$(APT_e, PT) \in S_m(d_x)$$

Using $f_{m \to r}(S_m)$,

$$(APT, PT) \in S_r(0)$$

(b) *points-to-type*(m, $APT2$, $<rec \Rightarrow E>$) exists at call node m, APT_e is either $\emptyset$ or imposed at entry node e simply because of actual-formal parameter bindings, and *points-to-type*(x, APT_e, PT) is present at x : Using induction hypothesis,

$$(APT2, <rec \Rightarrow E>) \in S_m(0)$$

Also, $APT_e \in$ *type-bind*$_{m \to e}(\emptyset)$ using Lemma B.1.1 or $APT_e = \emptyset$. Applying induction hypothesis again,

$$(APT_e, PT) \in S_x(0)$$

Using $f_{x \to m}(S_x)$,

$$(APT_e, PT) \in S_m(d_x)$$

Using $f_{m \to r}(S_m)$,

$$(\emptyset, PT) \in S_r(0)$$

$\square$

Lemma B.1.5 *For each node $n \in \mathcal{N}$, any iteration j of $\mathcal{A}_S$*

$$S_n^j(0) \subseteq PTT_n$$

We prove Lemma B.1.5 by induction on the number of iterations of $\mathcal{A}_S$.

basis: Let $j = 0$. For each node n, using Lemma B.1.2

$$S_n^0(0) = PTT_n^0$$

But $PTT_n^0 \subseteq PTT_n$ using Lemma B.1.3. Therefore,

$$S_n^0(0) \subseteq PTT_n$$

induction hypothesis: Let Lemma B.1.5 be true for $i < j$.

induction step: For those nodes whose solutions are not affected by the j^{th} iteration, $S_n^j(0) \subseteq PTT_n$ using induction hypothesis. We prove that for each node n whose solution changes (i.e., grows) after the j^{th} iteration, $S_n^j(0) \subseteq PTT_n$. For this purpose, we show that for each new tuple (APT, PT) added to $S_n^j(0)$,

$$(APT, PT) \in PTT_n$$

which is equivalent to stating that *points-to-type*(n, APT, PT) is *true* at node n in the fixed point solution of $\mathcal{A}_{PTT}$. We explore all cases which add such a tuple to $S_n^j(0)$ during the j^{th} iteration. Like in the proof for Lemma B.1.4, let $<rec \Rightarrow E>$ be the pointer-type pair which makes a virtual call m with receiver rec invoke the function with entry e.

1. n is not an entry or return node, and the tuple (APT, PT) is added to $S_n^j(0)$ by $f_{m \to n}(S_m^{j-1})$ because $(APT, PT1) \in S_m^{j-1}(0)$. Using induction hypothesis,

   $$(APT, PT1) \in PTT_m$$

   But **type-implies-type-through-other** must use *points-to-type*($m, APT, PT1$) resulting in **make-true**(*points-to-type*(n, APT, PT)).

2. (APT_e, APT_e) is added by $f_{m\to e}(S_m^{j-1})$ at an entry node e. This may happen in one of the following circumstances.

   (a) m is a non-virtual call, $(APT1, PT1) \in S_m^{j-1}(0)$ and APT_e belongs to $\textit{type-bind}_{m\to e}(PT1)$: Using induction hypothesis, *points-to-type*(m, $APT1$, $PT1$) is *true* at m and must contribute to the addition of *points-to-type*(e, APT_e, APT_e) in the solution for node e (using **case 1** of **type-implies-type-from-call**).

   (b) m is a virtual call, $\{(APT1, PT1), (APT2, <rec \Rightarrow E>)\} \subseteq S_m^{j-1}(0)$ and $APT_e \in \textit{type-bind}_{m\to e}(PT1)$: Using induction hypothesis, *points-to-type*(m, $APT1$, $PT1$) and *points-to-type*(m, $APT2$, $<rec \Rightarrow E>$) exist at m. Then it must be that *points-to-type*(e, APT_e, APT_e) is *true* (using either **case 2.1** or **case 2.2** of **type-implies-type-from-call**).

   (c) m is a virtual call, $(APT2, <rec \Rightarrow E>) \in S_m^{j-1}(0)$ and APT_e belongs to $\textit{type-bind}_{m\to e}(\emptyset)$: Using induction hypothesis, *points-to-type*(m, $APT2$, $<rec \Rightarrow E>$) exists at m. Then it must be that *points-to-type*(e, APT_e, APT_e) is *true* (using **case 2.1** of **type-implies-type-from-call**).

3. m is a virtual or non-virtual call, PT involves a local variable of the calling function and $f_{m\to r}(S_m^{j-1})$ adds (APT, PT) to the solution at the corresponding return node r : By induction hypothesis, $(APT, PT) \in PTT_m$. Then **type-implies-type-from-call**(m, APT, PT) must **make-true**(*points-to-type*(r, APT, PT)).

4. (APT, PT) is added to the solution at a return node r of non-virtual call m by $f_{m\to r}(S_m^{j-1})$ and PT involves a global variable. This can happen in one of the following circumstances. Let e be the entry of function invoked by m and x be the corresponding exit node for e.

   (a) $(APT, PT1) \in S_m^{j-1}(0)$, $APT_e \in \textit{type-bind}_{m\to e}(PT1)$, and $(APT_e, PT) \in S_m^{j-1}(d_x)$: Using induction hypothesis, *points-to-type*(m, APT, $PT1$) exists at call node m. Since the tuple (APT_e, PT) belongs to $S_m^{j-1}(d_x)$, it must have been added in some earlier iteration i by $f_{x\to c}(S_x^{i-1})$, implying that $(APT_e, PT) \in S_x^{i-1}(0)$. Using induction hypothesis again, *points-to-type*(x, APT_e, PT) is *true* at x. Therefore, *points-to-type*(r, APT, PT) using either **type-implies-type-from-call** or **type-implies-type-from-exit**.

(b) $APT = \emptyset$, $APT_e \in (\textit{type-bind}_{m\to e}(\emptyset) \cup \{\emptyset\})$ and $(APT_e, PT) \in S_m^{j-1}(d_x)$: Since the tuple (APT_e, PT) belongs to $S_m^{j-1}(d_x)$, it must have been added in some earlier iteration i by $f_{x\to c}(S_x^{i-1})$, implying that $(APT_e, PT) \in S_x^{i-1}(0)$. Using induction hypothesis, *points-to-type*(x, APT_e, PT) is *true* at x. Therefore, *points-to-type*(r, $\emptyset$, PT) must be *true* using **type-implies-type-from-exit**.

5. (APT, PT) is added to the solution at a return node r of virtual call m by $f_{m\to r}(S_m^{j-1})$ and PT involves a global variable. This can happen in one of the following circumstances. Let e be the entry of a function invoked by m and x be the corresponding exit node for e.

(a) $\{(APT, PT1), (APT2, <rec \Rightarrow E>)\} \subseteq S_m^{j-1}(0)$, $(APT_e, PT) \in S_m^{j-1}(d_x)$ where $APT_e \in \textit{type-bind}_{m\to e}(PT1)$: *points-to-type*(m, $APT2$, $<rec \Rightarrow E>$) and *points-to-type*(m, APT, $PT1$) exist at call node m by induction hypothesis. Since the tuple (APT_e, PT) belongs to $S_m^{j-1}(d_x)$, it must have been added in some earlier iteration i by $f_{x\to c}(S_x^{i-1})$, implying that $(APT_e, PT) \in S_x^{i-1}(0)$. Using induction hypothesis again, *points-to-type*(x, APT_e, PT) is *true* at x. Therefore, *points-to-type*(r, APT, PT) is *true* using either **type-implies-type-from-call** or **type-implies-type-from-exit**.

(b) $APT = \emptyset$, $(APT2, <rec \Rightarrow E>) \in S_m^{j-1}(0)$, $(APT_e, PT) \in S_m^{j-1}(d_x)$ where $APT_e \in (\textit{type-bind}_{m\to e}(\emptyset) \cup \{\emptyset\})$: *points-to-type*(m, $APT2$, $<rec \Rightarrow E>$) exists at call node m using induction hypothesis. Since the tuple (APT_e, PT) belongs to $S_m^{j-1}(d_x)$, it must have been added in some earlier iteration i by $f_{x\to c}(S_x^{i-1})$, implying that $(APT_e, PT) \in S_x^{i-1}(0)$. By induction hypothesis, *points-to-type*(x, APT_e, PT) is *true* at x. Therefore, *points-to-type*(r, $\emptyset$, PT)

is *true* using either **type-implies-type-from-call** or **type-implies-type-from-exit**.

□

Theorem B.1.4 *For each node n, the MFP solutions PTT_n of $\mathcal{A}_{PTT}$ and $S_n(0)$ of $\mathcal{A}_S$ are identical.*

Using Lemma B.1.4, after j^{th} iteration of $\mathcal{A}_{PTT}$

$$PTT_n^j \subseteq S_n(0)$$

Since $\mathcal{A}_{PTT}$ terminates after finite number of iterations,

$$PTT_n \subseteq S_n(0)$$

Using Lemma B.1.5, after j^{th} iteration of $\mathcal{A}_S$

$$S_n^j(0) \subseteq PTT_n$$

Since $\mathcal{A}_S$ terminates after finite number of iterations,

$$S_n(0) \subseteq PTT_n$$

Clearly for each node n,

$$S_n(0) = PTT_n$$

□

Algorithm $\mathcal{A}_S$ is based on concrete lattice theoretical foundations which makes it amenable to theoretical analyses such as the safety of calculation, but unsuitable for implementation. On the other hand, efficiency of implementation is an important motivation underlying the design of algorithm $\mathcal{A}_{PTT}$. Our prototype implementation described in Chapter 6 is directly based on this algorithm[40]. Due to various efficiency-oriented optimizations, it is difficult to reason about the theoretical properties of algorithm $\mathcal{A}_{PTT}$. By proving that they calculate the same solution (Theorem B.1.4), we have effectively shown that $\mathcal{A}_{PTT}$ is a safe and efficient algorithm for type determination of C^{++} programs restricted to the use of single level pointers.

[40]The inefficiencies of our prototype are due to factors external to the algorithm (such as limitations of the front end) and aspects common to both algorithms (such as poor representation of object names).

B.2 Type Determination and Aliasing for Multiple Level Pointers

In order to define a monotone data flow framework for the combined problem of type determination and aliasing in the presence of multiple, but finitely many, levels of pointer dereferencing (without k-limiting), we need a new list of set definitions. Note that these definitions override the definitions for identically named sets in Appendix B.1. Again, we assume the same restrictions on the language constructs as mentioned in Chapter 1 (p. 5).

$\mathcal{X}$: set of exit nodes of ICFG with each node in the set indexed by a unique number between 1 and $|\mathcal{X}|$. Let $\tilde{x} = |\mathcal{X}|$. Let d_x be the index of exit node x.

$\mathcal{EN}$: set of entry nodes of ICFG. Since there is a unique *exit* for each *entry*, each node e in $\mathcal{EN}$ is indexed by the same number d_e, between 1 and $\tilde{x}$, as its corresponding *exit*.

$\mathcal{T}$: Set of types in the program.

$\mathcal{NV}$: set of object names derived from nv_T, where nv_T is a global variable associated with each type $T \in \mathcal{T}$[41].

$\mathcal{LC}_n$: set of object names derived from local variables of the function containing ICFG node n.

$\mathcal{GB}$: set of object names derived from global variables in the program (including heap variables, static members of classes, and nv's).

$\mathcal{GB}_v = \mathcal{GB} - \mathcal{NV}$: set of global object names not involving nv.

$\mathcal{O}$: Set of object names, as defined in Table 2.1 (p. 12). Clearly, $\mathcal{O} = \mathcal{GB} \cup (\bigcup_{e \in \mathcal{EN}} \mathcal{LC}_e)$.

$\mathcal{P}$: Set of object names which have a pointer type. Clearly, $\mathcal{P} \subseteq \mathcal{O}$.

$\mathcal{PT} = (\mathcal{P} \times \mathcal{T})$: set of all possible pointer-type pairs (e.g., $<p \Rightarrow C>$).

---

[41] As mentioned in Section 5.4.2, we omit the explicit mention of T and simply denote the variable as nv.

$$\mathcal{RT}_{c \to e} = \left\{ <rec \Rightarrow T> \;\middle|\; \begin{array}{l} rec \in \mathcal{P} \text{ is the receiver of call node } c \text{, and} \\ c \text{ calls the method with entry node } e \text{ if } rec \\ \text{points to an object of } T \in \mathcal{T} \text{ at } c \end{array} \right\}.$$

$\mathcal{A} = (\mathcal{O} \times \mathcal{O})$: set of all possible ordered alias pairs (e.g., $<p, q>$ and $<q, p>$).

$\mathcal{APT} = \mathcal{A} \cup \mathcal{PT}$: the set of all possible alias and pointer-type pairs.

$\mathcal{APT}_\emptyset = (\mathcal{APT} \cup \{\emptyset\})$: set of all possible assumptions at entry nodes.

$\mathcal{S} = (\mathcal{APT}_\emptyset \times \mathcal{APT}_\emptyset \times \mathcal{APT})$.

$\mathcal{S}^{\tilde{x}}$: $(\tilde{x} + 1)$-ary cross product of $\mathcal{S}$.

$\emptyset^{\tilde{x}}$: $(\tilde{x} + 1)$-ary tuple of $\emptyset$s.

We have redefined the set $\mathcal{S}$ from Appendix B.1 to have the following interpretation in the context of the new framework. $\mathcal{S}$ is the set of all tuples of the form (*assumption*, *assumption*, *pointer-type/alias*). We define the instance of a monotone data flow framework for this general problem as $D = (ICFG_D, L, F, M)$, similar to that defined in Appendix B.1. However, we need to update the definitions of the following functions:

$$\pi_1(s) \;\; : 2^{\mathcal{S}} \rightarrow 2^{(\mathcal{APT}_\emptyset \times \mathcal{APT}_\emptyset)} = \left\{ (AAP1, AAP2) \;\middle|\; (AAP1, AAP2, AP) \in s \right\}$$

$$\pi_2(s) \;\; : 2^{\mathcal{S}} \rightarrow 2^{\mathcal{APT}} = \left\{ AP \;\middle|\; (AAP1, AAP2, AP) \in s \right\}$$

The mapping function $M : \mathcal{E}_D \rightarrow F$ must account for more complex edge functions reflecting the interdependence of type determination and aliasing. Having defined the framework for single level pointers in detail, we claim that the monotone function space F is powerful enough to include all the edge functions in the general framework. To substantiate this claim, we list the edge functions for some representative classes of edges. The description is incomplete, but we believe it provides the salient features of the framework sufficient to corroborate our claim.

As in Appendix B.1, we begin by describing the auxiliary functions responsible for parameter bindings at the call sites. Let c be a call node, $e \in \mathcal{EN}$ be possibly invocable from c, and $AP \in \mathcal{APT}$.

$$b_{c \to e}(\emptyset) = \textbf{type-bind0}(c,e) \cup \textbf{bind0}(c,e)$$

$$b_{c \to e}(AP) = \textbf{type-bind}(c,e,AP) \cup \textbf{alias-bind}(c,e,AP) \cup \textbf{bind}(c,e,AP)$$

The functions comprising $b_{c \to e}(\emptyset)$ and $b_{c \to e}(AP)$ have been described in Section 5.4.2 (p. 74).

Before we provide details of the selected edge functions, we must mention a few significant facts about our treatment of alias and pointer-type pairs which involve non-visible object names. As we saw in Section 5.4.4, when two incoming predicates propagate through a pointer assignment to derive a single predicate, we may choose the entry assumption of either incoming predicate as the assumption for the resulting predicate. Using the $\oplus$ operation, we prefer the non-$\emptyset$ assumption, arbitrarily picking one when both are non-$\emptyset$. Unfortunately, such a simple choice is not sufficient when object names with nv are involved. If the resulting predicate contains an object name p involving nv, we must use the entry assumption from that incoming predicate which contributes to the presence of p in the resulting predicate. As we saw in Section 5.4.2 (See p. 77), the alias or pointer-type pair which imposes the entry assumption AAP_e from call node c enables us to map back p to the original non-visible object name at c. In deference to this crucial role played by parameter binding, we ensure that AAP_e containing nv appears as entry assumption whenever the resulting alias or pointer-type pair contains an object name with the same nv. This situation is further complicated by the fact that sometimes two *may-hold* predicates, each involving an object name with nv, give rise to an alias, say AP, between those two object names. In Figure 5.11 (p. 83), suppose the predicates (i1) and (i2) result in (o2), and both *lhs_alias* and *objname* contain nv. In this special case, both $AAPT1$ and $AAPT2$ are essential; each enables us to retrieve the corresponding object name in the calling function when the alias propagates from the exit node of the called function. This is the only case where both the assumptions are non-$\emptyset$. Fortunately, such aliases do not interact with other nodes in the called function and can be propagated directly to the exit node.

The following terms facilitate the description of edge functions.

- *null_object*: A special object name $\notin \mathcal{O}$; a comparison of this object name with any object name in $\mathcal{O}$ returns *false*.

- *is_equal*(p, q): Returns *true* iff p is the access path representation of the object name q.

- *is_prefix*(p, q): Returns *true* iff q may be obtained by applying a (possibly empty) dereference sequence to an object name r where *is_equal*(p, r).

- *is_strict_prefix*(p, q): Returns *true* iff q may be obtained by applying a non-empty dereference sequence to an object name r where *is_equal*(p, r).

- *apply_deref*(p, q, u): Returns *null_object* if !*is_prefix*(p, q), otherwise returns the object name obtained by applying to u the dereference sequence which would transform an object name r into q, where *is_equal*(p, r).

- *pair_contain_nv*(*pair*): Returns *true* iff the alias or pointer-type *pair* contains an object name involving nv.

- $map\_nv1_{c\to e}(AP, AAP_e, <p,q>)$: Let $\ll c, e \gg$ be the interprocedural edge from a call node c to an entry node e, $<p,q>$ be such that at most one of p and q involve nv, and $AAP_e \in b_{c\to e}(AP)$. Then this function returns the alias pair $<a,b>$ where a and b are obtained in the following manner. If p involves nv, p is mapped to the corresponding object name a in the calling function, else $a = p$. This mapping is achieved using $AAP_e \in b_{c\to e}(AP)$. For example, if g is the corresponding actual associated with the formal f for the edge $\ll c, e \gg$, and l is a local variable at the call site c, $map\_nv1_{c\to e}(<*g, *l>, <*f, *nv>, <*nv, q>)$ will return $<*l, q>$. Similarly, if q involves nv, q is mapped to the corresponding object name b in the calling function, else $b = q$.

- $map\_nv2_{c\to e}(AP1, AAP1_e, AP2, AAP2_e, <p,q>)$: Let $\ll c, e \gg$ be the interprocedural edge from a call node c to an entry node e, $<p,q>$ be such that both p and q involve nv, $AAP1_e \in b_{c\to e}(AP1)$, and $AAP2_e \in b_{c\to e}(AP2)$. Then this function returns the alias pair $<a,b>$ where a and b are obtained in the following manner. p is mapped to the corresponding object name a in the calling function using $AAP1_e \in b_{c\to e}(AP1)$. Similarly, q is mapped to the corresponding object name b in the calling function using $AAP2_e \in b_{c\to e}(AP2)$.

- $map\_nv1_{c\to e}(AP, AAP_e, <p \Rightarrow C>)$: Let $\ll c, e \gg$ be the interprocedural edge from a call node c to an entry node e, and $AAP_e \in b_{c\to e}(AP)$. Then this function returns the pointer-type pair $<a \Rightarrow C>$ in the following manner. If p involves nv, p is mapped to the corresponding object name a in the calling function using $AAP_e \in b_{c\to e}(AP1)$, else $a = p$.

As in Appendix B.1, we first list some component functions which facilitate defining an edge function, describe these component functions intuitively, and then define the edge function using them. In all the following definitions, given a tuple $(AAP1, AAP2, pair)$, both $AAP1$ and $AAP2$ are non-$\emptyset$ iff $pair$ is an alias between two object names involving nv. In all other cases, $AAP2$ is $\emptyset$.

1. $\ll m, n \gg$ where n is a pointer assignment "`lhs = rhs`" and m is an intraprocedural predecessor. Assume that `lhs` as well as `rhs` contain dereferences.

$$h0(S) = S(0) - \left(\begin{array}{l} \left\{ (AAP, \emptyset, <p \Rightarrow C>) \;\middle|\; \begin{array}{l} AAP \in \mathcal{APT}_\emptyset \wedge C \in \mathcal{T} \wedge \\ is\_prefix(\texttt{lhs}, p) \end{array} \right\} \cup \\ \\ \left\{ (AAP, \emptyset, <p, q>) \;\middle|\; \begin{array}{l} AAP \in \mathcal{APT}_\emptyset \wedge p, q \in \mathcal{O} \wedge \\ \left(\begin{array}{l} is\_strict\_prefix(\texttt{lhs}, p) \vee \\ is\_strict\_prefix(\texttt{lhs}, q) \end{array} \right) \end{array} \right\} \end{array} \right)$$

$h0(S)$ removes all tuples $(AAP, \emptyset, AP)$ where AP involves an object name which can be obtained by applying a dereference sequence to `lhs`.

$$h1_{AAP1,\, AAP2,\, p,\, C,\, q,\, r}(S) = \left\{ \begin{array}{l} (AAP1, \emptyset, <p \Rightarrow C>), \\ (AAP2, \emptyset, <q, r>) \end{array} \right\} \subseteq S(0) \;?\; s_{AAP1,\, AAP2,\, p,\, C,\, q,\, r} : \emptyset$$

$$h2_{AAP1,\, AAP2,\, p,\, C,\, q,\, r}(S) = \left\{ \begin{array}{l} (AAP1, \emptyset, <p \Rightarrow C>), \\ (AAP2, \emptyset, <r, q>) \end{array} \right\} \subseteq S(0) \;?\; s_{AAP1,\, AAP2,\, p,\, C,\, q,\, r} : \emptyset$$

where $s_{AAP1,\, AAP2,\, p,\, C,\, q,\, r} =$

$$\left(\begin{array}{l} \left\{ \begin{array}{l|l} (AAP1 \oplus AAP2, \emptyset, <\!t \Rightarrow C\!>), & t == \mathit{apply\_deref}(\mathtt{rhs}, p, q) \wedge \\ (AAP1 \oplus AAP2, \emptyset, <\!u \Rightarrow C\!>) & u == \mathit{apply\_deref}(\mathtt{rhs}, p, r) \wedge \\ & !\mathit{pair\_contain\_nv}(AAP2) \end{array} \right\} \cup \\ \\ \left\{ \begin{array}{l|l} (AAP1, \emptyset, <\!t \Rightarrow C\!>), & t == \mathit{apply\_deref}(\mathtt{rhs}, p, q) \wedge \\ (AAP2, \emptyset, <\!u \Rightarrow C\!>) & u == \mathit{apply\_deref}(\mathtt{rhs}, p, r) \wedge \\ & \mathit{pair\_contain\_nv}(AAP2) \end{array} \right\} \end{array} \right)$$

$$h3(S) = \bigcup_{\substack{AAP1, AAP2 \in \mathcal{APT}_{\emptyset} \\ p \in \mathcal{O} \wedge \mathit{is\_prefix}(\mathtt{rhs}, p) \\ C \in \mathcal{T} \\ q \in \mathcal{O} \wedge \mathit{is\_equal}(\mathtt{lhs}, q) \\ r \in \mathcal{O}}} \left(\begin{array}{l} h1_{AAP1,\, AAP2,\, p,\, C,\, q,\, r}(S) \cup \\ h2_{AAP1,\, AAP2,\, p,\, C,\, q,\, r}(S) \end{array} \right)$$

$h3(S)$ is a comprehensive version of propagation described by Figure 5.9, accounting for cases when the incoming predicates may involve object names containing nv and the result is a *points-to-type* predicate. Note that the condition $p \in \mathcal{O} \wedge \mathit{is\_prefix}(\mathtt{rhs}, p)$ in the definition of $h3(S)$ above ensures that *apply_deref* never assigns *null_object* to t or u in the set $s_{AAP1,\, AAP2,\, p,\, C,\, q,\, r}$.

$$h4_{AAP1,\, AAP2,\, p,\, q,\, v,\, w}(S) = \left\{ \begin{array}{l} (AAP1, \emptyset, <\!p, q\!>), \\ (AAP2, \emptyset, <\!v, w\!>) \end{array} \right\} \subseteq S(0) \;?\; s_{AAP1,\, AAP2,\, p,\, q,\, v,\, w} : \emptyset$$

$$h5_{AAP1,\, AAP2,\, p,\, q,\, v,\, w}(S) = \left\{ \begin{array}{l} (AAP1, \emptyset, <\!q, p\!>), \\ (AAP2, \emptyset, <\!v, w\!>) \end{array} \right\} \subseteq S(0) \;?\; s_{AAP1,\, AAP2,\, p,\, q,\, v,\, w} : \emptyset$$

$$h6_{AAP1,\, AAP2,\, p,\, q,\, v,\, w}(S) = \left\{ \begin{array}{l} (AAP1, \emptyset, <\!p, q\!>), \\ (AAP2, \emptyset, <\!w, v\!>) \end{array} \right\} \subseteq S(0) \;?\; s_{AAP1,\, AAP2,\, p,\, q,\, v,\, w} : \emptyset$$

$$h7_{AAP1,\, AAP2,\, p,\, q,\, v,\, w}(S) = \left\{ \begin{array}{l} (AAP1, \emptyset, <\!q, p\!>), \\ (AAP2, \emptyset, <\!w, v\!>) \end{array} \right\} \subseteq S(0) \;?\; s_{AAP1,\, AAP2,\, p,\, q,\, v,\, w} : \emptyset$$

where $s_{AAP1,\,AAP2,\,p,\,q,\,v,\,w} =$

$$\left(\begin{array}{l}
\left\{\begin{array}{l|l}
(AAP1 \oplus AAP2, \emptyset, <t,q>), & t == apply_deref(\texttt{rhs}, p, q) \wedge \\
(AAP1 \oplus AAP2, \emptyset, <q,t>), & u == apply_deref(\texttt{rhs}, p, r) \wedge \\
(AAP1 \oplus AAP2, \emptyset, <u,q>), & !pair_contain_nv(AAP1) \wedge \\
(AAP1 \oplus AAP2, \emptyset, <q,u>) & !pair_contain_nv(AAP2)
\end{array}\right\} \cup \\
\\
\left\{\begin{array}{l|l}
(AAP1, \emptyset, <t,q>), & t == apply_deref(\texttt{rhs}, p, q) \wedge \\
(AAP1, \emptyset, <q,t>), & u == apply_deref(\texttt{rhs}, p, r) \wedge \\
(AAP1, \emptyset, <u,q>), & !pair_contain_nv(AAP1) \wedge \\
(AAP1, \emptyset, <q,u>) & pair_contain_nv(AAP2)
\end{array}\right\} \cup \\
\\
\left\{\begin{array}{l|l}
(AAP1, \emptyset, <t,q>), & t == apply_deref(\texttt{rhs}, p, q) \wedge \\
(AAP1, \emptyset, <q,t>), & u == apply_deref(\texttt{rhs}, p, r) \wedge \\
(AAP2, \emptyset, <u,q>), & !pair_contain_nv(AAP1) \wedge \\
(AAP2, \emptyset, <q,u>) & pair_contain_nv(AAP2)
\end{array}\right\} \cup \\
\\
\left\{\begin{array}{l|l}
(AAP1, \emptyset, <t,q>), & t == apply_deref(\texttt{rhs}, p, q) \wedge \\
(AAP1, \emptyset, <q,t>), & u == apply_deref(\texttt{rhs}, p, r) \wedge \\
(AAP1, AAP2, <q,u>), & pair_contain_nv(AAP1) \wedge \\
(AAP2, AAP1, <u,q>) & pair_contain_nv(AAP2)
\end{array}\right\}
\end{array}\right)$$

$$h8(S) = \bigcup_{\substack{AAP1, AAP2 \in \mathcal{APT}_\emptyset \\ p \in \mathcal{O} \wedge is_strict_prefix(\texttt{rhs}, p) \\ q, w \in \mathcal{O} \\ v \in \mathcal{O} \wedge is_equal(\texttt{lhs}, q)}} \left(\begin{array}{l}
h4_{AAP1,\,AAP2,\,p,\,q,\,v,\,w}(S) \cup \\
h5_{AAP1,\,AAP2,\,p,\,q,\,v,\,w}(S) \cup \\
h6_{AAP1,\,AAP2,\,p,\,q,\,v,\,w}(S) \cup \\
h7_{AAP1,\,AAP2,\,p,\,q,\,v,\,w}(S)
\end{array}\right)$$

$h8(S)$ is a comprehensive version of propagation described by Figure 5.11, accounting for cases when the incoming predicates may involve object names containing nv and the result is a *may-hold* predicate. Note that the condition $p \in \mathcal{O} \wedge is\_strict\_prefix(\texttt{rhs}, p)$ in the definition of $h8(S)$ above ensures that $apply\_deref$ never assigns $null\_object$ to

t or u in the set $s_{AAP1,\, AAP2,\, p,\, q,\, v,\, w}$.

We can similarly define a component function representing the aspects of propagation outlined in Figure 5.10. Let $h9(S)$ denote the function responsible for propagating a *may-hold* predicate to n using *points-to-type* and *may-hold* predicates at m. We omit the details of $h9(S)$. Given the above functions, the edge function can now be defined as

$$f_{m \to n}(S) = [h1(S) \cup h3(S) \cup h8(S) \cup h9(S), g^{1}_{0,0}(S), \ldots, g^{\tilde{x}}_{0,0}(S)]$$

2. $\ll c, e \gg$ where c is a non-virtual call node and e is the corresponding entry node. Let

$$h0(S) \;\; = \;\; \left\{ \; (AAP_e, \emptyset, AAP_e) \;\middle|\; AAP_e \in b_{c \to e}(\emptyset) \; \right\}$$

$h0(S)$ returns all tuples $(APT_e, \emptyset, APT_e)$ where $AAP_e \in b_{c \to e}(\emptyset)$.

$$\begin{aligned} h1_{AAP,\, AP}(S) \;\; &= \;\; \{(AAP, \emptyset, AP)\} \subseteq S(0) \\ &\quad\;\; ? \left\{ \; (AAP_e, \emptyset, AAP_e) \;\middle|\; AAP_e \in b_{c \to e}(AP) \; \right\} : \emptyset \end{aligned}$$

$$h2(S) \;\; = \;\; \bigcup_{\substack{AAP \in \mathcal{APT}_{\emptyset} \\ AP \in \mathcal{APT}}} h0_{AAP,\, AP}(S)$$

$h2(S)$ returns all tuples $(AAP_e, \emptyset, AAP_e)$ where (i) $AAP_e \in b_{c \to e}(AP)$ and (ii) AP holds at c.

$$f_{c \to e}(S) = [h0(S) \cup h2(S), g^{1}_{0,0}(S), \ldots, g^{\tilde{x}}_{0,0}(S)]$$

The edge function for $\ll c, e \gg$ where c is a virtual call and $e \in \mathcal{EN}$ can be defined as an extension of the above edge function. In Appendix B.1, we have already described a similar extension in the context of single level pointers.

3. $\ll c, r \gg$ where c is a non-virtual call node with corresponding entry node e and r is the corresponding return node.

$$h0(S) \;\; = \;\; S(d_e) \cap ((b_{c \to e}(\emptyset) \cup \{\emptyset\}) \times \{\emptyset\} \times ((\mathcal{GB}_v \times \mathcal{T}) \cup (\mathcal{GB}_v \times \mathcal{GB}_v)))$$

$$h1(S) \;\; = \;\; \{(\emptyset, \emptyset)\} \times \pi_2(h0(S))$$

$h1(S)$ returns tuples $(\emptyset, \emptyset, AP)$ such that (i) $AAP_e \in b_{c \to e}(\emptyset)$ or $AAP_e = \emptyset$, (ii) AP holds at corresponding exit node of e with assumption $(AAP_e, \emptyset)$, and (iii) AP only contains global object names not involving nv.

$$h2_{AP,\, AAP_e,\, AP_x}(S) \;=\; \{(AAP_e, \emptyset, AP_x)\} \subseteq S(d_e)$$

$$?\; \left\{ (AAP_e, \emptyset, AP1_x) \;\middle|\; \begin{array}{l} AAP_e \in b_{c \to e}(AP) \wedge \\ AP_x \in ((\mathcal{GB} \times \mathcal{GB}) \cup (\mathcal{GB} \times \mathcal{T})) \wedge \\ AP1_x == map\_nv1_{c \to e}(AP, AAP_e, AP_x) \end{array} \right\} : \emptyset$$

$$h3_{AP}(S) \;=\; \bigcup_{AAP_e, AP_x \,\in\, \mathcal{APT}} h2_{AP,\, AAP_e,\, AP_x}(S)$$

$$h4_{AP}(S) \;=\; \bigcup_{AAP \,\in\, \mathcal{APT}_\emptyset} \{(AAP, \emptyset, AP)\} \subseteq S(0) \;?\; \{(AAP, \emptyset, AP)\} : \emptyset$$

$$h5(S) \;=\; \bigcup_{AP \,\in\, \mathcal{APT}} \pi_1(h4_{AP}(S)) \times \pi_2(h3_{AP}(S))$$

$h3(S)$ returns tuples $(AAP, \emptyset, AP1_x)$ such that (i) AP holds at c with assumption $(AAP, \emptyset)$, (ii) $AAP_e \in b_{c \to e}(\mathrm{AP})$, (iii) AP_x holds at corresponding exit node of e with assumption $(AAP_e, \emptyset)$, (iv) AP_x contains only global object names with at most one object name involving nv, and (v) $AP1_x$ is the pair with appropriate object names mapped back using AP at c.

$$h6_{AAP1,\, AAP2,\, AP}(S) \;=\; \{(AAP1, AAP2, AP)\} \subseteq S(0) \;?\; \{(AAP1, AAP2, AP)\} : \emptyset$$

$$h7(S) \;=\; \bigcup_{\substack{AAP1, AAP2 \,\in\, \mathcal{APT}_\emptyset \\ AAP2 == \emptyset \\ AP \,\in\, ((\mathcal{LC}_c \times \mathcal{T}) \cup (\mathcal{LC}_c \times \mathcal{LC}_c))}} h6_{AAP1,\, AAP2,\, AP}(S)$$

$h7(S)$ returns tuples $(AAP1, \emptyset, AP) \in S(0)$ where the pair AP only consists of object names local to the function containing call node c. Such pairs cannot be destroyed by the called function, and hence directly propagate to the return node r.

$$h8(S) = \bigcup_{\substack{AAP1, AAP2 \in \mathcal{APT}_\emptyset \\ AP \in ((\mathcal{NV} \times \mathcal{T}) \cup (\mathcal{NV} \times \mathcal{NV}))}} h6_{AAP1,\, AAP2,\, AP}(S)$$

$h8(S)$ returns tuples $(AAP1, AAP2, AP) \in S(0)$ where the pair AP only consists of object names with nv. Such pairs cannot be destroyed by the called function, and hence directly propagate to the return node r.

$$h9_{AP1,\, AP2,\, AAP1_e,\, AAP2_e,\, AP_x}(S) = \{(AAP1_e, AAP2_e, AP_x)\} \subseteq S(d_e) \;?\; s_{AP1,\, AP2,\, AAP1_e,\, AAP2_e,\, AP_x} : \emptyset$$

where $s_{AP1,\, AP2,\, AAP1_e,\, AAP2_e,\, AP_x} =$

$$\left\{ (AAP1_e, AAP2_e, AP1_x) \;\middle|\; \begin{array}{l} pair\_contain\_nv(AAP1_e) \wedge AAP1_e \in b_{c\to e}(AP1) \wedge \\ pair\_contain\_nv(AAP2_e) \wedge AAP2_e \in b_{c\to e}(AP2) \wedge \\ AP_x \in (\mathcal{GB} \times \mathcal{GB}) \wedge (PT1_x == \\ map\_nv2_{c\to e}(AP1, AAP1_e, AP2, AAP2_e, AP_x)) \end{array} \right\}$$

$$h10_{AP1,\, AP2}(S) = \bigcup_{AAP1, AAP2, AP_x \in \mathcal{APT}} h9_{AP1,\, AP2,\, AAP1,\, AAP2,\, AP_x}(S)$$

$$h11_{AP1,\, AP2,\, AAP1,\, AAP2}(S) = \left\{ \begin{array}{l} (AAP1, \emptyset, AP1), \\ (AAP2, \emptyset, AP2) \end{array} \right\} \subseteq S(0) \;?\; \{(AAP1, AAP2, AP2)\} : \emptyset$$

$$h12_{AP1,\, AP2}(S) = \bigcup_{\substack{AAP1, AAP2 \in \mathcal{APT} \\ pair\_contain\_nv(AAP1) \\ pair\_contain\_nv(AAP2)}} h11_{AP1,\, AP2,\, AAP1,\, AAP2}(S)$$

$$h13(S) = \bigcup_{AP1, AP2 \in \mathcal{APT}} \pi_1(h12_{AP1,\, AP2}(S)) \times \pi_2(h10_{AP1,\, AP2}(S))$$

Alias pair AP_x at exit of the called function is such that both object names in this alias involve nv, and $h13(S)$ maps them to the corresponding object names in the calling function where each mapped object name again involves nv.

$$h14_{AP1,\,AP2,\,AAP1,\,AAP2}(S) \quad = \quad \left\{ \begin{array}{c} (AAP1, \emptyset, AP1), \\ (AAP2, \emptyset, AP2) \end{array} \right\} \subseteq S(0)\ ?\ \{(AAP1, \emptyset, AP1)\} : \emptyset$$

$$h15_{AP1,\,AP2}(S) \quad = \quad \bigcup_{\substack{AAP1, AAP2 \in \mathcal{APT} \\ pair\_contain\_nv(AAP1) \\ !pair\_contain\_nv(AAP2)}} h14_{AP1,\,AP2,\,AAP1,\,AAP2}(S)$$

$$h16(S) \quad = \quad \bigcup_{AP1, AP2 \in \mathcal{APT}} \pi_1(h15_{AP1,\,AP2}(S)) \times \pi_2(h10_{AP1,\,AP2}(S))$$

Alias pair AP_x at exit of the called function is such that both object names in this alias involve nv, and $h15(S)$ maps them to the corresponding object names in the calling function where at most one of the mapped object names again involves nv.

Given the above component functions,

$$f_{c \to r}(S) = [h1(S) \cup h5(S) \cup h7(S) \cup h8(S) \cup h13(S) \cup h16(S), g^{1}_{0,0}(S), \ldots, g^{\tilde{x}}_{0,0}(S)]$$

References

[AG96] D. C. Atkinson and W. G. Griswold. The design of whole-program analysis tools. In *Proceedings of the Eighteenth International Conference on Software Engineering*, 1996.

[AH90] H. Agrawal and J.R. Horgan. Dynamic program slicing. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–256, June 1990. Also available as SIGPLAN Notices, vol 25, no 6, June 1990.

[AH95] A. Agesen and U. Hölzle. Type feedback vs. concrete type inference: a comparison of optimization techniques for object-oriented languages. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*, pages 91–107, October 1995.

[AL95] R. Z. Altucher and W. Landi. An extended form of must alias analysis for dynamic allocation. In *Conference Record of the Twenty-second Annual ACM Symposium on Principles of Programming Languages*, pages 74–84, January 1995.

[APS93] O. Agesen, J. Palsberg, and M. Schwartzbach. Type inference of SELF: Analysis of objects with dynamic and multiple inheritance. In *European Conference on Object-Oriented Programming (ECOOP '93)*, pages 247–267, July 1993.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Ban79] J. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 29–41, January 1979.

[Bar78] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, 1978.

[BC92] M. Burke and Jong-Deok Choi. Precise and efficient integration of interprocedural alias information into data flow analysis. *ACM Letters on Programming Languages and Systems*, 1(1):14–21, March 1992.

[BCCH94] Michael Burke, Paul Carini, J-D. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, pages 234–250. Springer-Verlag, August 1994.

[Bur90] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[Cal88] D. Callahan. The prgram summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 47–56, June 1988.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[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 Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[CG94] B. Calder and D. Grunwald. Reducing indirect function call overhead in C^{++} programs. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, pages 397–408, January 1994.

[CGZ95] B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioural differences between C and C^{++} programs. *Journal of Programming Languages*, 2:313–351, 1995.

[CHS95] P. R. Carini, M. Hind, and H. Srinivasan. Flow-sensitive type analysis for C^{++}. Research Report RC 20267, IBM T. J. Watson Research Center, 1995.

[CK88] K. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 57–66, June 1988.

[CK89] K. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 49–59, January 1989.

[CU89] C. Chambers and D. Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 146–160, June 1989.

[CU90] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: optimizing dynamically-typed object-oriented programs. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 150–164, June 1990.

[DCG95] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 93–102, June 1995.

[Deu94] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 230–241, June 1994.

[DMM95] A. Diwan, J. E. B. Moss, and K. S. McKinley. Analyzing statically-typed object-oriented programs for modern processors. Technical report, department of computer science, University of Massachusetts, Amherst, July 1995.

[EGH94] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–257, June 1994. Published as SIGPLAN Notices, 29 (6).

[Fer95] M. Fernandez. Simple and effective link-time optimization of modula-3 programs. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 103–115, June 1995.

[FW85] P. G. Frankl and E. J. Weyuker. A data flow testing tool. In *Proceedings of IEEE Softfair II*, December 1985.

[FW88] P. Frankl and E. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[GH96] R. Ghiya and L. 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 Twenty-third Annual ACM Symposium on Principles of Programming Languages*, pages 1–15, January 1996.

[Gor93] T. Goradia. Dynamic impact analysis: A cost effective technique to enforce error propagation. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis*, pages 171–181, June 1993.

[Har89] W. L. Harrison. The interprocedural analysis and automatic parallelization of scheme programs. *LISP and Symbolic Computation*, 2(3-4):179–396, October 1989.

[HCU91] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object Oriented Programming*, July 1991.

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, 1977.

[HFGO94] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the Sixteenth International Conference on Software Engineering*, 1994.

[HK93] M. Hall and K. Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, 1993.

[HPR89] S. Horwitz, J. Prins, and T Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 1989.

[HR94] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 154–163, December 1994.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), January 1990.

[HS89] M. J. Harrold and M. L. Soffa. Interprocedural data flow testing. In *Proceedings of the Third Testing, Analysis, and Verification Symposium*, pages 158–167, December 1989.

[HS90] M. J. Harrold and M. L. Soffa. Computation of interprocedural definition and use dependencies. In *Proceedings of the 1990 International Conference on Computer Languages*, pages 297–306, 1990.

[HS91] M. J. Harrold and M. L. Soffa. Selecting and using data for integration testing. *IEEE Software*, 8(2):58–65, March 1991.

[HS94] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

[HU94] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with tuntime type feedback. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 326–336, June 1994.

[JM79] N. Jones and S. Muchnick. Flow analysis and optimization of lisp-like structures. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 244–256, January 1979.

[JW95] S. Jagannathan and A. Wright. Efficient flow analysis for avoiding runtime checks. In *Proceedings of the Second International Static Analysis Symposium (SAS '95)*, 1995. Also appears as Springer-Verlag LNCS-983.

[KAI95] S. Kumar, D. P. Agrawal, and S. P. Iyer. An improved type-inference algorithm to expose parallelism in object-oriented programs. In *Proceedings of the Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*. Kluwer Academic Publications, Troy, New York, May 1995.

[KB95] B. Kuhn and D. Binkley. An enabling optimization for C^{++} virtual functions. In *Proceedings of Mid-Atlantic States Graduate Workshop on Programming Languages and Systems, East Stroudsburg, PA*, April 1995.

[KL90] B. Korel and J. Laski. Dynamic slicing of computer programs. *Journal of Systems Software*, 13:187–195, 1990.

[KRS94] J. Knoop, O. Rüthing, and B. Steffen. A tool kit for constructing optimal interprocedural data flow analyses. Technical Report MIP-9413, Universität Passau, November 1994.

[KU77] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[Lak93] A. Lakhotia. Constructing call multigraphs using dependence graphs. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 273–284, January 1993.

[Lan92a] W. Landi. *Interprocedural Aliasing in the Presence of Pointers*. PhD thesis, Rutgers University, January 1992. Available as Laboratory for Computer Science Research Technical Report LCSR-TR-174.

[Lan92b] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[Lar89] J. R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, University of California, Berkeley, May 1989.

[Lar92] J. M. Larcheveque. Interprocedural type propagation for object-oriented languages. In *Proceedings of the Fourth European Symposium on Programming (ESOP '92)*, February 1992.

[LH88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, July 1988. SIGPLAN NOTICES, Vol. 23, No. 7.

[Lom77] D. Lomet. Data flow analysis in the presence of procedure calls. *Journal of Research and Development*, 21(6):559–571, November 1977.

[LR91] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 93–103, January 1991.

[LR92] W. Landi and B. G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

[LRZ93] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.

[LRZA96] W. Landi, B. G. Ryder, S. Zhang, and R. Altucher. Interprocedural modification side effect analysis for C systems. Laboratory for computer science research technical report, Rutgers University, 1996. In preparation.

[LT90] S. S. Liu and A. B. Taha. Interprocedural definition-use dependency analysis for recursive procedures. Software Engineering Research Center SERC-TR-42-F, University of Florida, Gainesville, 1990.

[MLR+93] T. J. Marlowe, W. A. Landi, B. G. Ryder, J. Choi, M. Burke, and P. Carini. Pointer-induced aliasing: A clarification. *ACM SIGPLAN Notices*, 28(9):67–70, September 1993.

[MMR95] Stephen P. Masticola, Thomas J. Marlowe, and Barbara G. Ryder. Lattice frameworks for multi-source and bidirectional data flow analysis problems. *ACM Transactions on Programming Languages and Systems*, 17(5):777–803, September 1995.

[MR90] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: a unified model. *Acta Informatica*, 28:121–163, 1990.

[Mye81] E. M. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 219–230, January 1981.

[OO84] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, May 1984.

[Ost90] Thomas J. Ostrand. Data-flow testing with pointers and function calls. In *Proceedings of the Pacific Northwest Software Quality Conference*, October 1990.

[OW91] T. J. Ostrand and E. Weyuker. Data flow based test adequecy analysis for languages with pointers. In *Proceedings of the 1991 Symposium on Software Testing, Analysis and Verification (TAV4)*, October 1991.

[Par92] R. Parameswaran. Interprocedural alias and type analysis for pointers. Master's thesis, Department of Computer Science, University of Wisconsin - Madison, 1992.

[PC94] J. Plevyak and A. Chien. Precise concrete type inference for object oriented languages. In *Proceeding of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '94)*, pages 324–340, October 1994.

[PLR94] H. D. Pande, W. Landi, and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[PR94a] H. Pande and B. G. Ryder. Static type determination for C^{++}. In *Proceedings of the Sixth USENIX C^{++} Technical Conference*, pages 85–97, April 1994.

[PR94b] H. D. Pande and B. G. Ryder. Static type determination and aliasing for C^{++}. Laboratory for Computer Science Research Technical Report LCSR-TR-236, Rutgers University, December 1994.

[PRL91] H. Pande, B. G. Ryder, and W. Landi. Interprocedural def-use associations for C programs. In *Proceedings of the ACM SIGSOFT Conference on Testing, Analysis and Verification*, pages 139–153, October 1991.

[PS91] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91)*, pages 146–161, October 1991.

[PWC91] Michael Platoff, Michael Wagner, and Joseph Camaratta. An integrated program representation and toolkit for the maintenance of C programs. In *Proceedings of the Conference on Software Maintenance*. IEEE Computer Society Press, October 1991.

[Ram94] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, September 1994.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the Twenty-second Annual ACM Symposium on Principles of Programming Languages*, pages 49–61, January 1995.

[RLP90] B. G. Ryder, W. Landi, and H. Pande. Profiling an incremental data flow analysis algorithm. *IEEE Transactions on Software Engineering*, 16(2):129–140, February 1990.

[RM88] C. Ruggieri and T. Murtagh. Lifetime analysis of dynamically allocated objects. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 285–293, January 1988.

[RW85] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[Ryd79] B. G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–225, May 1979.

[Shi90] O. Shivers. Data-flow analysis and type recovery in scheme. In *Topics in Advanced Language Implementation*. MIT Press, 1990.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[SRW96] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Conference Record of the Twenty-third Annual ACM Symposium on Principles of Programming Languages*, pages 16–31, January 1996.

[SS92] M. Suedholt and C. Steigner. On interprocedural data flow analysis for object oriented languages. In *Proceedings of the International Conference on Compiler Construction, Germany*, 1992.

[Suz81] N. Suzuki. Inferring types in smalltalk. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 187–199, January 1981.

[Ven91] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 107–119, June 1991.

[VHU92] J. Vitek, R. N. Harspool, and J. S. Uhl. Compile-time analysis of object oriented programs. In *Proceedings of the International Conference on Compiler Construction, Germany*, 1992.

[Wei80] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, January 1980.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[WL95] Robert Wilson and Monica Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995. Also available as SIGPLAN Notices, 30(6).

[YHR90] W. Yang, S. Horwitz, and T. Reps. A program integration algorithm that accomodates semantics preserving transformations. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 133–143, December 1990. Also available as SIGSOFT Notes, vol 15, no 6, December 1990.

[YHR92] W. Yang, S. Horwitz, and T. Reps. A program integration algorithm that accomodates semantics preserving transformations. *ACM Transactions on Software Engineering Methodology*, 1(3):310–354, July 1992.

[ZR94] S. Zhang and B.G. Ryder. Complexity of single level function pointer aliasing analysis. Laboratory for Computer Science Research Technical Report LCSR-TR-233, Rutgers University, October 1994.

Vita

Hemant D. Pande

1981–1985	Attended Indian Institute of Technology (IIT), Bombay.
1985	B.Tech. in Computer Science and Engineering, IIT, Bombay.
1985–1986	Teaching assistant, Dept. of Computer Science, Rutgers University.
1985–1989	Graduate work in Computer Science, Rutgers University.
1986–1989	Graduate assistant (Center for Computer Aids for Industrial Productivity grant), Dept. of Computer Science, Rutgers University.
1988	M.S. in Computer Science, Rutgers University.
1989–1991	Member of technical staff, Siemens Corporate Research, Princeton, NJ.
1990	"Profiling an Incremental Data Flow Analysis Algorithm" [RLP90].
1990	M.Phil. in Computer Science, Rutgers University.
1991	"Interprocedural Def-Use Associations for C Programs" [PRL91].
1991–1993	Member of technical staff, Tata Research Development and Design Centre, Pune.
1993–1996	Doctoral student and research assistant (Siemens Corporate Research Fellowship), Dept. of Computer Science, Rutgers University.
1994	"Static Type Determination for C^{++}" [PR94a].
1994	"Interprocedural Def-Use Associations for C Systems with Single Level Pointers" [PLR94].
1996	Ph.D. in Computer Science, Rutgers University.