Static Type Determination and Aliasing for C++*

Hemant D. Pande† Barbara G. Ryder

Department of Computer Science
Rutgers University
Hill Center, Busch Campus
Piscataway, NJ 08855
email: {pande,ryder}@cs.rutgers.edu

LCSR-TR-236

Abstract

Static type determination involves compile time calculation of the type of object a pointer may point to, at a particular program point during some execution. We show that the problem of precise interprocedural type determination is NP-hard in the presence of inheritance, virtual functions and pointers. We highlight the significance of type determination in improving code efficiency and precision of other static analyses. First we present a safe, approximate type determination algorithm for C++ programs with single level pointers, using the conditional analysis technique [LR91]. Then we present a generalization of this approach to simultaneously solve the aliasing and type determination problems for C++ programs with multiple levels of pointer dereferencing, and explain why this is a more complicated analysis.

Keywords : C++, Interprocedural static analysis, Type determination, Aliasing, Virtual functions, Pointers.

1 Introduction

In the past decade, significant research by the static analysis community has concentrated on expanding compile time analysis to include interprocedural information [BCC+94, Bur90, Cal88, CBC93, CK88, CK89, HS94, HRB90, LRZ93, MLR+93, Mey81]. Historically, compile time analysis has been used in intraprocedural context for code optimizations. The emphasis is shifting towards the use of interprocedural static analysis in compiling as well as all phases of software life cycle including debugging, integration and testing [FW85, HS89, HRB90, OW91, RW85, Wei84, YHR92]. However, until recently software tools either have not performed interprocedural static analysis or have employed grossly approximate techniques for languages with pointers. Landi and Ryder have shown the theoretical difficulty of static analysis in the presence of pointers [LR91, Lan92]

*This research was supported, in part, by funds from NSF grants CCR90-23628, CCR-92-08632 and Siemens Corporate Research.

†On study leave from Tata Research Development and Design Centre (a division of Tata Consultancy Services), 1 Mangaldas Road, Pune 411050, India.

and introduced a new technique for interprocedural analysis of C programs. They have also developed a safe, approximate algorithm to solve the *may aliasing* problem for a restricted subset of C which excludes pointers to functions, casting[1], union types, exception handling, *setjump* and *longjump* [LR92]. Arrays are treated as a single aggregate without distinguishing the individual elements. Our recent analysis of C programs [PLR94, PRL91], based on this work, represents one of the first attempts to obtain highly precise static interprocedural information for C programs and to apply it successfully in a software tool, the Test Analysis and Coverage Tool (TACTIC) [OW91].

Encouraged by the results obtained from analyzing C, we are now concentrating on how to apply the static analysis techniques beneficially to C++ programs. We have focussed our efforts on developing new techniques to handle most of the features distinguishing C++ from C such as inheritance and virtual functions (i.e. object-orientedness), subtyping and overloading (i.e. polymorphism). The most significant C++ feature affecting compile time analysis is virtual functions, because the type of receiver at a virtual call site dynamically determines the function to be invoked. With static *type determination*, such a late binding may be replaced by a function call to an appropriate function, or inlined code in suitable circumstances, thereby eliminating the overhead of late binding and improving the execution efficiency. Recent empirical studies of dynamic behavior of C++ programs indicate there is opportunity to avoid late bindings in many cases, which is particularly significant for architectures which employ deep pipelining [CG94]. Additionally, a statically determined list of possible types for a receiver would focus further analyses only on selected functions, rather than the entire pool of functions with the same name, potentially saving analysis time. Also, exclusion of the statically un-invocable functions from analysis would eliminate their spurious side effects, thereby improving the precision of subsequent analyses.

Static type determination is the first step in analyzing C++ since it represents important semantic information whose precision can greatly affect the quality and utility of various other static analyses. We demonstrate the significance of this information for other analysis problems by solving the problem of *aliasing*, which determines when a single memory location may be accessed with at least two distinct names at a program point. These two problems, aliasing and type determination, are fundamental for C++ since their solutions are virtually indispensable for any further analyses. For ease of explanation, we first present a type determination algorithm for the restricted case when the C++ programs under analysis use only single level pointers. When the pointer usage is so restricted, type determination can be solved independent of aliasing. Our C++ aliasing algorithm in this case is a slight modification of the aliasing algorithm for C as described in [LR91]. In the presence of multiple level pointers, however, these problems cannot be solved separately, due to their interdependence; we explain this while presenting a single algorithm that solves type determination and aliasing together. Ours is the first such combined analysis for these two problems. Our work contributes to: (i) increased efficiency and precision of other compile time analyses and (ii) improved run time performance of the programs analyzed.

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.

[1] The algorithm has now been modified to handle casting.

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 [HK92, Lak93, MGH94, Ryd79]. 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. However, when there is no increase in generality, we reduce a C^{++} construct to a semantically equivalent C construct. For example, we transform a *class constructor* to a *malloc* followed by appropriate initializations and we express the principle of encapsulation using the concepts of scope and visibility in C. The C^{++} programs we analyze have the same restrictions as our earlier C analyses [LR92, PLR94, PRL91].

Overview : In Section 2, we mention related work. We introduce the program representation, define the terminology and establish the theoretical problem complexity in Section 3. We show that the problem is NP-hard in the presence of single level pointers. Thus the intractability of the problem is demonstrable in this restricted case without generalizing to multiple level pointers. In Section 4, we describe a polynomial algorithm to determine the program-point-specific *points-to* information (i.e. the type of object pointed to by a pointer at a program point). We provide a running example to derive *points-to* values at some key program points and use it to bring out the significance of type determination. Most of the algorithm description in this section originally appears in our earlier work on type determination for C^{++} program with single level pointers [PR94]. In Section 5, we describe the combined algorithm for type determination and aliasing in the general case of multiple level pointers. Section 6 includes our preliminary implementation results. Finally, we conclude by summarizing our contributions.

2 Related Work[2]

Program-point-specific type determination for object oriented languages has been attempted with varying degrees of success. Suzuki's algorithm [Suz81] handles languages like Smalltalk where objects serve as receivers of functions, but the problem is alleviated by the significant absence of pointers to objects. The algorithm by Palsberg and Schwartzbach [PS91] infers types of expressions in an object oriented language with inheritance, assignments and late bindings. 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. Recent work on improving run-time efficiency of SELF, a dynamically typed language, uses customization, iterative type analysis and inline caches to replace dynamic binding with procedure calls or inlined code [CU89, CU90, HCU91]. More recently, the algorithms using run-time type feedback [HCU94] and type inference [APS93] for optimizing SELF programs have shown promise of adaptability to C^{++}. The algorithm by Larcheveque [Lar92] is rendered imprecise by the fact that it factors out the side effects of function invocations and aliasing due to parameter bindings as well as pointers. The suggested algorithms for these problems [CK89, Wei84] are grossly approximate and unsuitable in a C^{++} context. We show that aliasing and type determination are inseparable in the general case, therefore a factored approach

[2]For related work on pointer-induced aliasing, see [LR92].

is not desirable. Ramesh 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 uses the precalculated alias information for type determination. Suedholt and Steigner [SS92] use a concept of *representant* virtual function to keep information about all the virtual functions with the same name. This approach leads to the loss of context which distinguishes one virtual function from others. Vitek *et al* [VHU92] 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.

3 Problem Definition

Program Representation

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 union of CFGs for the individual functions comprising the program. 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* 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 function has returned[3]. For a non-virtual function call, we represent the control flow into the called function by an interprocedural edge from *call* 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. However, virtual function 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 virtual function invocation to appropriate *entry* node(s) and from *exit* node(s) to the *return* node is part of the algorithm presented in this paper.

Terminology

- An ICFG path is *realizable* if, whenever a function on this path returns, it returns to the call site which invoked it [LR91]. Not all paths in the ICFG are realizable.
- 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[4]. In other words, the first and the last node on a balanced path belong to the same invocation of the function containing them both.
- *Objects* correspond to locations that can store information, and *object names* provide ways to refer to them. We associate names with static memory locations or with heap locations created by *new*. For static storage, the name-storage association is created through a declaration

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

[4] We defined the terms *realizable* and *balanced* paths independently, and have only recently found that the ideas already existed in literature. They are referred to as *valid* and *complete* respectively in [SP81].

statement. For heap storage, we create names for the creation site, new_{pp} where pp is the program point where the location is created. An object name is a variable name or a name of a heap location, and a possibly empty sequence of dereferences and member accesses.

- 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[5]. We represent aliases by unordered pairs of object names (e.g. $<v, *p>$). The order is unimportant since the alias relation is symmetric.
- *Type Determination* involves calculating the type of the object pointed to by a pointer at a program point as a result of some execution to that program point. We represent this information by pairs of a pointer and an associated type (e.g. $<p \Rightarrow C>$).
- 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*. Our analysis tries to restrict itself to consistent paths since non-consistent paths do not correspond to any possible execution sequence.
- The *precise*[6] solution for static type determination at a program point is a set of pointer-type pairs, each of which is a result of an execution on some consistent path to that program point.

Theoretical Complexity of the Problem

Theorem 1 *In the presence of single level pointers and virtual functions in C++, precise program-point-specific type determination 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 and show that the cnf is satisfiable iff we can solve the point-specific type determination problem precisely on the program segment, thereby proving that the problem is NP-hard. We represent each logic variable v_i as a program variable with the name `v_i` and its negation with an explicit program variable `nv_i`. Also, each `l_jk` in the program segment corresponds to the literal l_{jk} from the cnf and represents a program variable `v_i` or `nv_i`. Figure 1 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.

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

[6]Under the standard assumption of static analysis that all intraprocedural paths are executable.

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)
   {
       return arg;
   }
}

True *False::and (True *arg)
   {
        return &false;
   }
}

True *v_1, *nv_1, *v_2, *nv_2, ..., *v_m, *nv_m;
True *c;

L1:
   if (-) {v_1 = &true; nv_1 = &false;} else {v_1 = &false; nv_1= &true;}
    .....
   if (-) {v_m = &true; nv_m = &false;} else {v_m = &false; nv_m = &true;}
L2:
   if (-) c = l_11; else if (-) c = l_12; else c = l_13;
   if (-) c = c->and(l_21); else if (-) c = c->and(l_22); else c = c->and(l_23);
   ...
   if (-) c = c->and(l_n1); else if (-) c = c->and(l_n2); else c = c->and(l_n3);
L3:
```

Figure 1: Reducing 3-SAT to Static Type Determination

Claim: cnf is satisfiable iff `c` points 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 `l_jk` points to *true*, defining an all-*true* path from L2 to L3. By construction, the variable `c` points to `true` at the end of this 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 `l_jk` points to `false`. This row will be responsible for the variable `c` to point to `false` from that row onwards. Thus at statement L3, `c` will never point to `true`. □

An easy corollary follows, since the theorem involves a subproblem of the following problem.

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

4 Approximate Type Determination Algorithm for Single Level Pointers

Apart from the restrictions mentioned in Section 1, the algorithm described here applies to C++ programs which only use a single level of dereferencing with pointers. In the following description,

we consider the receiver of a function call as the first actual parameter and denote the corresponding formal as *this*.

Our algorithm uses the idea of *conditional analysis* [LR91]. Execution flow in a function is analyzed by assuming information that can hold at the entry of the function. The resulting analysis is *conditional* on the assumed information at entry. The algorithm is rendered practical by doing computation only for those assumptions which actually reach the entry node on some execution path.

- 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 a set S of 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 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 $\mathcal{P}_{e,S}$.
- We define the *conditional type determination problem* at node n as:

  There exists a conditionally consistent path with respect to a set S from *entry* to node n on which a pointer-type pair PT holds, **and** there exists a consistent path from ρ to *entry* on which every pointer-type pair in the set S holds.

Since this formulation is computationally intractable, we approximate it by considering assumption sets containing at most one pointer-type pair. If the set S contains multiple pairs, we use any one i) to approximate a conditionally consistent path from *entry* to n, and ii) as the only necessary assumption for type determination calculation. This approximation leads to an overestimate of the type determination solution [Pan94]. An underestimate of the solution would lead to missed pointer-type pairs, which may subsequently make a virtual function uninvocable at a call site, rendering the calculation unsafe. However, an overestimate preserves the safety of calculation; if there exists a path to a node on which a pointer-type pair holds during some execution, an overestimate will include the pair in the solution for that node.

We define a predicate *points-to* with the following interpretation reflecting the approximation: *points-to*$(n, APT, <p \Rightarrow C>) == true$ if (i) there exists a consistent path from ρ to the *entry* node of the procedure containing node n, on which the pointer-type pair APT (if any) holds; and (ii) given that (i) is true, there exists a conditionally consistent path $\mathcal{P}_{entry,APT}$ to n on which $<p \Rightarrow C>$ holds.

4.1 A Running Example

Before discussing the algorithm, we examine a program segment in Figure 2. We will use it in Section 4.2 to illustrate the significance of type determination in practical issues of run-time efficiency and benefits to other optimizations. Throughout Section 4.3, we will use it as an illustration for our algorithm description and in Section 4.5, to illustrate the sources of approximation for our algorithm. We assume that the boolean conditions in *if* statements are side effect free and hence inconsequential to the analysis.

```
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 2: Example of Type Determination Algorithm

4.2 Practical Issues

At node $n13$ in Figure 2, 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 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].

4.3 Algorithm Description

To determine the type of an object a pointer variable may point to at a given program point, we perform a fixed point computation of the equations describing the C^{++} statement side effects on the predicate *points-to*, as described below. Underlying this analysis, we have a data flow framework defined on the simple *true/false* lattice. The elements of the lattice describe the values of *points-to* predicates at each program point. 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* predicate *points-to*$(n, APT, <ptr \Rightarrow C>)$ for some APT, guaranteeing the safety of calculation.

We use a worklist for the fixed point computation. Whenever a predicate *points-to*(n, APT, PT) becomes *true* for the first time, it is placed on the worklist. Once marked *true*, a predicate stays *true*. Thus a *true* predicate goes on the worklist exactly once, guaranteeing the termination of our

for each node n in the ICFG
If n is
1. `n : p = new t :`
   make-true ($points\text{-}to(n, \emptyset, <p \Rightarrow t>)$)
2. `n : p = &r :`
   make-true ($points\text{-}to(n, \emptyset, <p \Rightarrow type(r)>)$)
   where $type(r)$ returns the type of object name r.
3. `n : foo(param`$_1$`, ..., param`$_k$`) :`
   make-true ($points\text{-}to(entry_{foo}, <p \Rightarrow type(r)>, <p \Rightarrow type(r)>)$)
   for each `param`$_i$ of the form `&r` with pointer variable `p` as the corresponding formal, and the call is non-virtual.

Figure 3: Introduction Phase

algorithm. We refer to this action as **make-true** and denote it in the algorithm by "**make-true** ($points\text{-}to(n, APT, PT)$)".

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* predicates are assumed *false* initially. For efficiency, we have designed the algorithm in such a way that work is performed only for $points\text{-}to(n, APT, <ptr \Rightarrow C>)$ which become *true*. Given the solution for *points-to* at node n, the information about pointer-type pairs at n can be easily computed as follows:

$$pointer\text{-}type\text{-}info(n) = \left\{ <ptr \Rightarrow C> \;\middle|\; (\exists APT) points\text{-}to(n, APT, <ptr \Rightarrow C>) == true \right\}.$$

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

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 3 lists the nodes examined in the introduction phase and their associated actions. Note that in item 3 we restrict ourselves to non-virtual function calls, because without the knowledge of the receiver type, we can make no educated guesses about the function invoked. We handle virtual function calls during the propagation phase.

Using the program segment in Figure 2, we list the following examples of type introduction. Since there exists a path $entry_{main}.n4$ at the end of which $<p \Rightarrow Base>$ holds without assuming any information at $entry_{main}$, using item 1,

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

Since there exists a path $entry_{main}.n7$ at the end of which $<p \Rightarrow Derived>$ holds without assuming any information at $entry_{main}$, using item 1 we also have

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

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

Figure 4: Propagation Phase

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

make-true (*points-to*$(n9, \emptyset, <s \Rightarrow Derived>)$)

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 n given the assumption APT, to all the successors of the node n, 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 4 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$(n, APT, <ptr \Rightarrow C>)$
This function captures the intraprocedural aspects of type propagation as described in the following cases.

case 1: If successor is an assignment to *ptr*, "`m : ptr = ...`", the given *points-to* does not propagate through m. Whatever *ptr* pointed to before node m was encountered is immaterial.

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

make-true (*points-to*$(m, APT, <aptr \Rightarrow C>)$) and **make-true** (*points-to*$(m, APT, <ptr \Rightarrow C>)$).

Type casting appears in the latter node so that the assignment is type-correct, but it is unimportant for our analysis since *aptr* points to an object of class C irrespective of the cast type.

case 3: If successor node m neither defines nor uses the pointer variable *ptr*, then the type of *ptr* is preserved: **make-true** (*points-to*$(m, APT, <ptr \Rightarrow C>)$). This is a case of simple propagation of information without any change. In the example for type introduction, we inferred *true* values for *points-to*$(n4, \emptyset, <p \Rightarrow Base>)$ and *points-to*$(n7, \emptyset, <p \Rightarrow Derived>)$. Propagating this information through the type preserving successors up to node $n9$, we **make-true**

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

Using further applications of **case 3**, the information at $n9$ propagates to its successors as

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

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, $\textit{points-to}(call_{n10}, \emptyset, <s \Rightarrow Derived>)$ is *true*. Since s is visible in the called function $Derived :: bar()$, we **make-true**

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

At the call site $n10$, s is the first actual parameter and corresponds to the formal *this* of $Derived :: bar()$. Since $\textit{points-to}(call_{n10}, \emptyset, <s \Rightarrow Derived>)$ is *true*, we **make-true**

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

If *ptr* is not visible in the called function, the type pointed to by *ptr* cannot change[7]. In this case we propagate the predicate $\textit{points-to}(call, APT, <ptr \Rightarrow C>)$ directly to the corresponding *return* node as $\textit{points-to}(return, APT, <ptr \Rightarrow C>)$.

case 2: Call is virtual. Suppose the call node is: "`n : rec→fun ( )`".
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* needs to be propagated only to the function invocable when the receiver points to an object of class C. In the example, $\textit{points-to}(call_{n11}, \emptyset, <p \Rightarrow Base>)$ propagates to $entry_{Base::foo}$ as

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

   but not to $entry_{Derived::foo}$. On the other hand, $\textit{points-to}(call_{n11}, \emptyset, <p \Rightarrow Derived>)$ propagates to $entry_{Derived::foo}$ as

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

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

[7] 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** ($points\text{-}to(entry, <f \Rightarrow type(r)>, <f \Rightarrow type(r)>)$). Note that this case cannot be handled in the introduction phase, as the invocability of this function from call node n would not be known then.

   b) For each $points\text{-}to(call, APT', <ptr' \Rightarrow E>)$ where $ptr' \neq rec$:
   We determine the corresponding entry and perform actions as in **case 1**. Thus while propagating $points\text{-}to(call_{n11}, \emptyset, <p \Rightarrow Base>)$, we propagate the *true* predicate $points\text{-}to(call_{n11}, \emptyset, <b \Rightarrow Base>)$ to $entry_{Base::foo}$ with

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

case 2.2: *ptr* and *rec* are distinct variables:
Suppose the *points-to* information currently available about the receiver *rec* at the given call node is:

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

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 $points\text{-}to(call_{n11}, \emptyset, <s \Rightarrow Derived>)$ is the candidate for propagation at $call_{n11}$. We have also seen that $points\text{-}to(call_{n11}, \emptyset, <p \Rightarrow Base>)$ and $points\text{-}to(call_{n11}, \emptyset, <p \Rightarrow Derived>)$ 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 $points\text{-}to(call_{n11}, \emptyset, <s \Rightarrow Derived>)$ to the corresponding *entry* nodes using:

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

If the pointer variable *ptr* is not visible in any one (or more) of these invocable functions, the predicate on the worklist propagates directly to the *return* node by **make-true** ($points\text{-}to(return, APT, <ptr \Rightarrow C>)$).

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 node we propagate the effects of this call from the exit of the called function to the return node corresponding to the call node [**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* predicate at the call node c_i (i.e. *points-to*$(entry, APT, APT)$ was made *true* during introduction phase), we simply propagate $<ptr \Rightarrow C>$ to r_i. In this case: **make-true** (*points-to*$(r_i, \emptyset, <ptr \Rightarrow C>)$). On the other hand, suppose it took *points-to*(c_i, APT'', APT') to impose APT at the *entry*; then we have: **make-true** (*points-to*$(r_i, APT'', <ptr \Rightarrow C>)$).

In our example, suppose we are propagating *points-to*$(exit_{Base::foo}, <p \Rightarrow Base>, <p \Rightarrow Base>)$. We have two return nodes *viz.* $return_{n11}$ and $return_{n14}$. Since it takes *points-to*$(call_{n11}, \emptyset, <p \Rightarrow Base>)$ to impose $<p \Rightarrow Base>$ at $entry_{Base::foo}$, using the information thus available at $call_{n11}$ and $exit_{Base::foo}$:

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

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

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

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*$(r_i, \emptyset, <ptr \Rightarrow C>)$).

In either of the above cases, there is an opportunity for improving the precision: Suppose a function $E :: foo()$ is invocable from a virtual call site with receiver l due to the presence of a *true* valued *points-to*$(c_i, APT''', <l \Rightarrow E>)$. 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*$(r_i, \emptyset, <p \Rightarrow C>)$. Since APT''' 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 APT''' to obtain a more precise *points-to*$(r_i, APT''', <p \Rightarrow C>)$.

4.4 Algorithm Complexity

The following considerations are significant while determining the worst case complexity of our algorithm.

1. The values of *points-to* are initialized in unit time (representation dependent).
2. The value of a predicate changes at most once, from *false* to *true*, and then stays *true*. A *true* predicate is only added to the worklist once, when its value has just been changed from *false* to *true*.
3. The total time complexity of actions performed for introductions and intraprocedural propagation is of the order of the number of ICFG edges, (or the number of ICFG nodes.)

4. For each ICFG node, the relevant solution is the third argument of the *points-to* predicate. For example, $\textit{points-to}(n, APT1, <p \Rightarrow C>)$, $\textit{points-to}(n, APT2, <p \Rightarrow C>)$ ...all yield the same inference that p may point to an object of class C at node n.

Assuming the above and further that the maximum number of assumptions ($APT's$) for each pointer-type pair derived at a node is bounded by a constant, the algorithm is linear in the solution size.

4.5 Sources of Approximation

We illustrate the sources of approximation in our type determination algorithm for single level pointers using two examples from Figure 2.

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

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

   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

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

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

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

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

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

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

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

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

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

   at $call_{n12}$ implies **make-true** $(\textit{points-to}(return_{n12}, \emptyset, <b \Rightarrow Base>))$. 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 Approximate Type Determination and Aliasing Algorithm for Multiple Level Pointers

In programs restricted to single level pointers, one pointer cannot be aliased to another, as this requires multiple levels of indirection. As a result, when a pointer changes its type (to point to an object of another type), it does not affect the type of any other pointers. 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. As an example, the node: "`m : p = &q;`" creates alias $<*p, q>$. Suppose subsequently on an execution path, "`n : *p = &r;`" creates pointer-type pair $<*p \Rightarrow type(r)>$. In the absence of information that the alias pair $<*p, q>$ holds before node n, we would not be able to infer $\textit{points-to}(n, \emptyset, <q \Rightarrow type(r)>)$, and the type determination would be rendered incorrect and unsafe. (Recall, it is unsafe to underestimate the set of possible types of a receiver object.)

We now present an algorithm to perform type determination and aliasing calculation together. Unlike previous section, we will use the *points-to* calculation in the solution of the aliasing problem, rather than performing type determination alone. This interleaved calculation is facilitated by two new predicates: $\textit{may-hold}(n, AAPT, <a, b>)$ and $\textit{points-to}(n, AAPT, <q \Rightarrow C>)$. The former is related to the *may-hold* in [LR92, PRL91, PLR94]. The latter is a variant of our *points-to* predicate from the previous section. We use $\textit{may-hold}(n, AAPT, <a, b>)$ to encode the approximate answer to the Conditional May Alias question: $\textit{may-hold}(n, AAPT, <a, b>)$ is *true* if $<a, b>$ holds on a conditionally consistent path $\mathcal{P}_{entry, AAPT}$ from *entry* of the procedure containing n to n **and** there is a consistent path from ρ to *entry* on which $AAPT$ holds. Similarly the new encoding of *points-to* is: $\textit{points-to}(n, AAPT, <p \Rightarrow C>)$ is true if p points to type C on a conditionally consistent path $\mathcal{P}_{entry, AAPT}$ to n **and** there is a consistent path from ρ to *entry* on which $AAPT$ holds. To account for the interaction between type determination and aliasing, $AAPT$ may be an alias pair, a pointer-type pair or $\emptyset$.

For efficiency reasons, we have designed the algorithm in such a way that work is performed only for *true* valued *may-hold* and *points-to* predicates, similar to the approach taken in Section 4. From the definitions of the predicates, may alias at a node is easily computable:

$$\textit{may-alias}(n) = \left\{ <a, b> \;\middle|\; \exists AAPT(\textit{may-hold}(n, AAPT, <a, b>) == \text{true}) \right\}$$

If we required information about pointer-type pairs at node, it would also be easily computable given the solution for *points-to*:

$$\textit{pointer-type-info}(n) = \left\{ <p \Rightarrow C> \;\middle|\; \exists AAPT(\textit{points-to}(n, AAPT, <p \Rightarrow C>) == \textit{true}) \right\}$$

As a significant departure from the aliasing algorithm in [LR92], we do not dereference a pointer until we can (statically) determine that it points to an object. For example, the assignment "`n : p = q`" does not result in the creation of an alias pair $<*p, *q>$ unless q is known to point to some object (i.e. it is neither uninitialized nor does it point to NULL at n). This information is obtained from a *true* valued *points-to* predicate involving q at a predecessor of n.

Although our algorithm handles multiple level pointer dereferencing, we omit the treatment of pointers to recursive structures (potentially yielding infinite levels of dereferences) from this description. The visibility of automatic variables across function boundaries complicates the analysis

significantly. We leave out the visibility issue from the following description. The details of the complete algorithm appear in [Pan94].

5.1 Parameter Bindings

Before providing the algorithm details, we describe some auxiliary functions to capture type and aliasing effects on the entry of the invoked function by the types and aliases present at the invocation site.

bind0(call,entry) : This function calculates the aliasing effects from *call* to *entry* without requiring any information from the predecessor(s) of *call*. In other words, only the information local to *call* is utilized. For example, if $\&a$ passed as actual to the formal $f1$, $<*f1, a>$ 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, $bind0(call, entry) =$

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

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

type-bind0(call,entry) : This function calculates the type effects from *call* onto *entry* without requiring any 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 and $\&a$ is passed as actual to the formal f; then $<f \Rightarrow B> \in type\text{-}bind0(call, entry)$.

We examine the details of the remaining functions with respect to the example in Figure 5.

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* or may also give rise to additional alias pairs. In the example, $<*p, *q>$ reaches $call_{c1}$. The aliases created at $entry_{C::foo}$ by this pair fall into the following three categories:

1. between dereferences of formals: Since the dereferences of two actuals are aliased at $call_{c1}$, the dereferences of corresponding formals are aliased at $entry_{C::foo}$. So, $<*foo1, *foo2> \in bind(call_{c1}, entry_{C::foo}, <*p, *q>)$.

2. between an alias of a dereference of an actual and appropriate dereference of the corresponding formal: Since $*p$ is aliased to $*q$ at $call_{c1}$, $*foo1$ is aliased to $*q$ at $entry_{C::foo}$. Similarly, $*foo2$ is aliased to $*p$ at $entry_{C::foo}$.

3. *alias* itself propagating to *entry*: $<*p, *q>$ is itself aliased at $entry_{C::foo}$.

For this example, $bind(call_{c1}, entry_{C::foo}, <*p, *q>) =$

$$\{<*foo1, *foo2>, <*foo1, *q>, <*foo2, *p>, <*p, *q>\}$$

alias-bind(call,entry,pointer-type) : The aliases created by a *pointer-type* pair fall into the following two categories:

```
class B { public:
  int b1;
};

class C { public:
  int foo (int *foo1, *foo2);
  int bar (D *bar1; B *bar2);
} *s;

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

class D { public:
   B *d1;
} *r;
```

Figure 5: Example for Parameter Bindings

1. alias 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* pair 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 $\mathit{alias\text{-}bind}(call_{c2}, entry_{C::bar}, <r{\rightarrow}d1 \Rightarrow B>)$.

2. alias between dereference of actual and the corresponding formal: Given that *pointer-type* pair is $<r{\rightarrow}d1 \Rightarrow B>$ and $r{\rightarrow}d1$ is passed to formal $bar2$ at $call_{c2}$,
$$\{<{*}r{\rightarrow}d1, {*}bar2>, <r{\rightarrow}d1{\rightarrow}b1, bar2{\rightarrow}b1>\}$$
also forms a subset of $\mathit{alias\text{-}bind}(call_{c2}, entry_{C::bar}, <r{\rightarrow}d1 \Rightarrow B>)$.

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* pair may simply propagate to *entry* or may also make a dereference of the corresponding formal point to an instance of some class. In our example,
$$\mathit{type\text{-}bind}(call_{c2}, entry_{C::bar}, <r{\rightarrow}d1 \Rightarrow B>) = \{<r{\rightarrow}d1 \Rightarrow B>, <bar2 \Rightarrow B>\}$$

5.2 Algorithm Overview

We perform a fixed point computation of the equations describing the C^{++} statement side effects on the predicates *may-hold* and *points-to*, as described below. Underlying this analysis, we have a

true/false lattice based data flow framework similar to the one described in Section 4. If there exists an execution path to node n on which a pointer p points to an object of type C, our algorithm will report a *true* predicate *points-to*$(n, AAPT, <p \Rightarrow C>)$ for some entry assumption $AAPT$. Similarly, if there exists an execution path to node n on which $<a, b>$ holds, our algorithm will report a true predicate *may-hold*$(n, AAPT, <a,b>)$ for some $AAPT$. We thus maintain the safety of our calculation. Our algorithm is justifiably approximate for the same reasons as mentioned in Section 4, reporting an overestimate of the aliases and pointer-type pairs.

Like Section 4, we compute the predicate values in three phases: i) *initialization* ii) *introduction* and iii) *propagation*. In this section, we use **make-true** on both *points-to* and *may-hold* predicates to set each one to *true* when appropriate and add it to the worklist exactly once. A high level description of the algorithm appears in Figure 6.

Our algorithm has a high-level structure that corresponds to a *lazy* evaluation of the interactions between *points-to* 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 5.2.3) and interprocedural (Sections 5.2.4 and 5.2.5) propagations.

5.2.1 Initialization Phase

We start with initializing all the *points-to* and *may-hold* predicates to *false*. Similar to the approach in Section 4, we use a constant time initialization for the predicates. We also initialize the worklist to empty.

5.2.2 Introduction Phase

We first concentrate on alias and type introduction by providing details of the relevant functions from Figure 6. The first two functions describe the intraprocedural aspects while the remaining two functions describe the interprocedural aspects of the introduction phase.

aliases-intro-by-assignment(n) : If an address of a variable q is assigned to another variable p, the dereference of p is aliased to q. For example if q is an object of class B, the assignment "`n : p = &q`" results in

make-true (*may-hold*$(n, \emptyset, <*p, q>)$) and **make-true** (*may-hold*$(n, \emptyset, <p \rightarrow mem_k, q.mem_k>)$)[8] .

types-intro-by-assignment(n) : If an address of a variable of class B is assigned to another variable, the destination clearly points to an instance of class B after n. For example if q is an object of class B, "`n : p= &q`" causes **make-true** (*points-to*$(n, \emptyset, <p \Rightarrow B>)$).

[8] We use the alias pairs like $<p \rightarrow mem_k, q.mem_k>$ to denote all aliases involving corresponding members of the class.

```
find-aliases ( )
        // initialization of information
        set all possible predicates to false;
        set worklist to empty;
        // introduction of aliases and pointer-type pairs
        for each node N in the ICFG
                if N is an assignment to a pointer
                        alias-intros-by-assignment (N)
                        type-intros-by-assignment (N)
                if N is a non-virtual call node with corresponding ENTRY
                        alias-intros-by-call (N,ENTRY)
                        type-intros-by-call (N,ENTRY)
        // propagation of aliases and pointer-type pairs
        while worklist is not empty
                remove (N, AAPT, APT) from worklist
                if N is a call node
                        if APT is an alias pair
                                alias-implies-alias-from-call (N, AAPT, APT)
                                alias-implies-type-from-call (N, AAPT, APT)
                        if APT is a pointer-type pair
                                type-implies-type-from-call (N, AAPT, APT)
                                type-implies-alias-from-call (N, AAPT, APT)
                else if N is an exit node
                        if APT is an alias pair
                                alias-implies-alias-from-exit (N, AAPT, APT)
                        if APT is a pointer-type pair
                                type-implies-type-from-exit (N, AAPT, APT)
                else
                        if APT is an alias pair
                                alias-implies-alias-thru-other(N, AAPT, APT)
                                alias-implies-type-thru-other (N, AAPT, APT)
                        if APT is a pointer-type pair
                                type-implies-type-thru-other (N, AAPT, APT)
                                type-implies-alias-thru-other (N, AAPT, APT)
```

Figure 6: A High Level Description of the Aliasing Algorithm

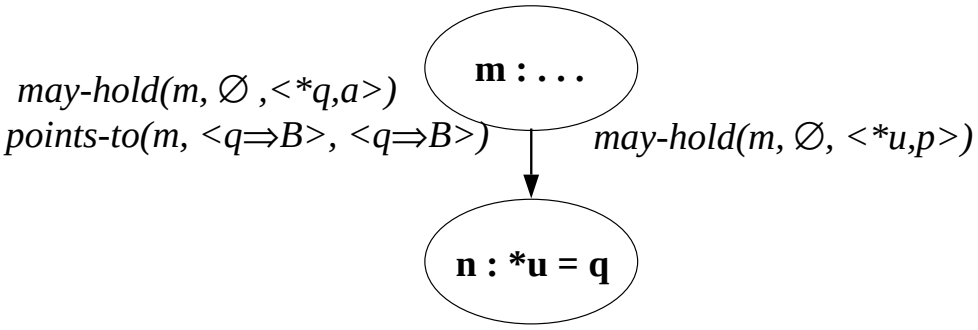

Figure 7: Example for Intraprocedural Propagation

aliases-intro-by-call(call,entry) : This function has the following trivial task: for each alias $<a, b>$ from the pre-computed $bind0(call, entry)$, **make-true** ($may\text{-}hold_{(entry, <a,b>, <a,b>)}$).

types-intro-by-call(call,entry) : This function has the following trivial task: for each $<p \Rightarrow B>$ in $type\text{-}bind0(call, entry)$, **make-true** ($points\text{-}to_{(entry, <p \Rightarrow B>, <p \Rightarrow B>)}$).

5.2.3 Intraprocedural Propagation

We concentrate on deriving information only at assignment nodes, using the semantic impact of code at the node itself and the information at its ICFG predecessors.

Propagation is trivial through an intraprocedural successor which is not a pointer assignment node. An assignment involving no pointers cannot create or destroy aliases, nor can it change the type pointed to by a pointer. A predicate $points\text{-}to(m, AAPT, PT)$ propagates through a non-pointer assignment n as $points\text{-}to(n, AAPT, PT)$. Similarly a predicate $may\text{-}hold(m, AAPT, PA)$ propagates through a non-pointer assignment n as $may\text{-}hold(n, AAPT, PA)$.

However when the successor is a pointer assignment node, propagation depends on the interaction between the incoming predicate and the object name(s) participating in the assignment. An incoming *may-hold* may interact with the assignment to create *may-hold* or *points-to* predicates after the assignment. Similarly, an incoming *points-to* may interact with the assignment to create *points-to* or *may-hold* predicates after the assignment. We present the salient features of intraprocedural propagation using the predicates from the example shown in Figure 7. We use appropriate functions in Figure 6 to propagate the individual predicates.

alias-implies-alias-thru-other(node,AAPT,AP) : Suppose the predicate under consideration is $may\text{-}hold(m, \emptyset, <*u, p>)$.

1. Since an assignment to $*u$ at n cannot destroy any of its aliases, we simply propagate $may\text{-}hold(m, \emptyset, <*u, p>)$ to n with **make-true** ($may\text{-}hold_{(n, \emptyset, <*u,p>)}$).

2. Information regarding the type of the right hand side of an assignment enables us to create aliases between the dereference of the object name aliased to the left hand side and that of the right hand side. Since q is known to point to an instance of class B at m, p would

also point to the same instance because $*u$ and p are aliased before n. Thus the presence of *points-to*$(m, <q \Rightarrow B>, <q \Rightarrow B>)$ in conjunction with *may-hold*$(m, \emptyset, <*u, p>)$ causes

make-true (*may-hold*$(n, <q \Rightarrow B>, <*p, *q>)$)[9]
make-true (*may-hold*$(n, <q \Rightarrow B>, <p \rightarrow mem_k, q \rightarrow mem_k>)$)

Note that the predicates *may-hold*$(m, \emptyset, <*u, p>)$ and *points-to*$(m, <q \Rightarrow B>, <q \Rightarrow B>)$, although *true* at the same node, need not be the result of executing the same consistent path. However, for the safety of calculation the algorithm must account for the case when they actually are.

As a result of the assignment, $*u$ and p both point to the same instance of class B, implying

make-true (*may-hold*$(n, <q \Rightarrow B>, <**u, *p>)$)
make-true (*may-hold*$(n, <q \Rightarrow B>, <(*u) \rightarrow mem_k, p \rightarrow mem_k>)$)

3. Since $*q$ is known to be aliased to a at m, $*p$ would also be aliased to a at n. Thus the presence of *may-hold*$(m, \emptyset, <*q, a>)$ implies **make-true** (*may-hold*$(n, \emptyset, <*p, a>)$). Like in the previous case, this is a safe but conservative implication of two independent *true* predicates at m.

Suppose the predicate under consideration is *may-hold*$(m, \emptyset, <*q, a>)$. Immediately after the assignment at n, the dereference of the left hand side becomes aliased to any alias of $*q$. As a result, we **make-true** (*may-hold*$(n, \emptyset, <**u, a>)$). Also, since *may-hold*$(m, \emptyset, <*u, p>)$ is *true*, we **make-true** (*may-hold*$(n, \emptyset, <*p, a>)$). Note that this is the dual of case 3 above. Either of the two will take place depending on which predicates become candidate for propagation earlier during the worklist algorithm.

Since the assignment cannot break the alias $<*q, a>$, we simply propagate $<*q, a>$ to node n using **make-true** (*may-hold*$(n, \emptyset, <*q, a>)$).

alias-implies-type-thru-other(node,AAPT,AP) : Suppose *may-hold*$(m, \emptyset, <*u, p>)$ is the candidate for propagation to node n. Using the information that $<q \Rightarrow B>$ holds at m (possibly on a distinct path), we safely infer **make-true** (*points-to*$(n, <q \Rightarrow B>, <p \Rightarrow B>)$).

type-implies-type-thru-other(node,AAPT,PT) : Suppose *points-to*$(m, <q \Rightarrow B>, <q \Rightarrow B>)$ is being propagated to the successor n. Using another *true* valued predicate *may-hold*$(m, \emptyset, <*u, p>)$, we determine that the left hand side of the assignment is aliased to p, thus p will point to an instance of class B immediately after the assignment, implying **make-true** (*points-to*$(n, <q \Rightarrow B>, <p \Rightarrow B>)$).

Since the assignment does not alter that value of q, we simply propagate $<q \Rightarrow B>$ to node n with **make-true** (*points-to*$(n, <q \Rightarrow B>, <q \Rightarrow B>)$).

type-implies-alias-thru-other(node,AAPT,PT) : Let the predicate under propagation be *points-to*$(m, <q \Rightarrow B>, <q \Rightarrow B>)$. As a result of the assignment, $*u$ and q point to the same instance of class B, implying

[9]Since we can accommodate at most one entry assumption in our approximate formulation, we pick the non-$\emptyset$ assumption from the former predicate.

make-true ($may\text{-}hold(n, <q \Rightarrow B>, <**u, *q>)$)
make-true ($may\text{-}hold(n, <q \Rightarrow B>, <(*u)\rightarrow mem_k, q\rightarrow mem_k>)$)

If $may\text{-}hold(m, \emptyset, <*u, p>)$ is already *true* when $points\text{-}to(m, <q \Rightarrow B>, <q \Rightarrow B>)$ is removed from the worklist for propagation (dual of the case 2 in **alias-implies-alias-thru-other**), we use the two predicates to infer:

make-true ($may\text{-}hold(n, <q \Rightarrow B>, <*p, *q>)$)
make-true ($may\text{-}hold(n, <q \Rightarrow B>, <p\rightarrow mem_k, q\rightarrow mem_k>)$)
make-true ($may\text{-}hold(n, <q \Rightarrow B>, <**u, *p>)$)
make-true ($may\text{-}hold(n, <q \Rightarrow B>, <(*u)\rightarrow mem_k, p\rightarrow mem_k>)$)

5.2.4 Propagation from *call* nodes

For non-virtual function calls, the corresponding entry node is easily determined. However if *call* represents a virtual call site, the *points-to* predicates involving the receiver at *call* determine the possible functions invoked. 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-alias-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** ($may\text{-}hold(entry, AA, AA)$). 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 $may\text{-}hold(exit, AA, PA)$. Associating this information at *call* and *exit*, we **make-true** ($may\text{-}hold(return, AAPT, PA)$).

alias-implies-type-from-call(call,AAPT,alias) : The set of aliases imposed at *entry* due to the presence of *alias* at *call* is available as $bind(call, entry, alias)$. In an earlier iteration of the algorithm, if an alias pair AA in this set has already served as an entry assumption to create a *true* predicate $points\text{-}to(exit, AA, <q \Rightarrow D>)$, we **make-true** ($points\text{-}to(return, AAPT, <q \Rightarrow D>)$).

type-implies-alias-from-call(call,AAPT,pointer-type) : 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 $type\text{-}bind(call, entry, <p \Rightarrow B>)$. If some pair PT in this set has already created a *true* predicate of the form $may\text{-}hold(exit, PT, <c, d>)$, we **make-true** ($may\text{-}hold(return, AAPT, <c, d>)$).

Each $AA \in alias\text{-}bind(call, entry, <p \Rightarrow B>)$ causes **make-true** ($may\text{-}hold(entry, AA, AA)$). Some pairs from this set may have already yielded a *true* valued predicate $may\text{-}hold(exit, AA, <a, b>)$ at the exit of called function, which results in **make-true** ($may\text{-}hold(return, AAPT, <a, b>)$).

type-implies-type-from-call(call,AAPT,pointer-type) : Apart from propagating the *pointer-type* pair to appropriate *entry* nodes and the *return* node, this function accounts for the special case of type propagation when *pointer-type* involves the receiver of *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* and *may-hold*) already existing at *call* is propagated to the new *entry*. We have already seen the various kinds of propagation from *call* to *entry* (except the one which immediately follows).

Without loss of generality, let *pointer-type* be $<p \Rightarrow B>$. The pointer-type pairs imposed at *entry* by $<p \Rightarrow B>$ are available through the set *type-bind*$(call, entry, <p \Rightarrow B>)$. Each pair APT in this set creates a *true* predicate *points-to*$(entry, APT, APT)$. Any *points-to*$(exit, APT, <q \Rightarrow D>)$ already existing at *exit* propagates to *return* with **make-true** (*points-to*$(return, AAPT, <q \Rightarrow D>)$).

$<p \Rightarrow B>$ may also create aliases at *entry* as obtained by *alias-bind*$(call, entry, <p \Rightarrow B>)$. An alias AA from this set may have already served as assumption for a *true* predicate of the form *points-to*$(exit, AA, <r \Rightarrow C>)$ at *exit* of the called function, which propagates to *return* with **make-true** (*points-to*$(return, AAPT, <r \Rightarrow C>)$).

5.2.5 Propagation from *exit* to *return* nodes

The following functions from Figure 6 are responsible for propagating the *true* valued predicates at *exit* nodes to the corresponding *return* node(s).

alias-implies-alias-from-exit(exit,AAPT,alias) : 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. As a result, *may-hold*$(return, \emptyset, alias)$ is made true for all *return* nodes corresponding to *call* nodes invoking this function. *call* may either be a function call or virtual invocation.

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

type-implies-type-from-exit(exit,AAPT,pointer-type) : 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 imposed at the entry. As a result, we **make-true** (*points-to*$(return, \emptyset, pointer\text{-}type)$) for all *return* nodes corresponding to *call* nodes invoking this function.

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

5.3 Sources of Approximation

The aliasing calculation of our algorithm is based on the algorithm by Landi and Ryder [LR92] and inherits the approximations made by them. During calculation of *points-to* predicates, we make the same approximations as listed in Section 4.5. The underlying reason behind all our

Program name	Lines	#functions	#virtuals	#virtual calls	%*resolved*	%*eliminated*
vcirc	141	16	4	5	100	44
greed	970	54	16	17	100	68
objects	465	59	35	39	92	92
shapes	265	52	32	21	86	58
chess	185	43	12	1	0	14

Table 1: Preliminary results

approximation is that we cannot tell whether two facts holding at a program point are *true* due to a single consistent path. The factors which lead to approximate calculations during the independent aliasing and typing calculations are also present in the combined algorithm. In course of the algorithm description, we have mentioned such circumstances during the interleaved calculation of aliasing and type determination.

6 Implementation Results and Current Status

The results presented in this section represent the initial efforts to assess the practicality of our algorithm in Section 5 using a prototype implementation. Currently our implementation only analyzes programs with single inheritance and non-recursive data structures. We are using the MasterCraft C^{++} system of Tata Consultancy Services as the front end C^{++} parser for our implementation. Our aliasing and type determination algorithm is similar in design to the aliasing algorithm from Landi and Ryder [LR92]; our prototype reuses some code from their implementation with suitable modifications.

We are currently working on improving the execution efficiency of the implementation as well as lifting the restriction to non-recursive structures so that the implementation can accept a broader range of C^{++} programs. At present, our testing efforts are at a preliminary stage. In table 1, we present some measurements from our test suite of five C^{++} programs. The programs were written by C^{++} programmers to demonstrate typical object oriented programming style. The last program is incomplete, nevertheless we include the results from analyzing it as it represents a pathological case when the functionality of the program under analysis makes a virtual call potentially invoke all possible virtual functions. The column listed under %*resolved* represents the percentage of virtual function calls which can be converted to simple function calls because the receiver type determined allowed resolution to a unique virtual function. In the column titled %*eliminated*, we present a qualitative measure of the savings obtained by statically determining a list of types for a receiver at a virtual call site, even when the call cannot be resolved to a unique function. Suppose there are n functions invocable from a virtual call site given the declared type of the receiver, and our analysis was responsible for determining that only m would be invocable. There is a percentage saving of $(n-m)*100/n$ at this call site by eliminating the propagation of information to $(n-m)$ statically uninvocable functions. %*eliminated* lists the average percentage savings over all the virtual call sites in the program. In all programs except *chess*, the gains obtained by type determination are

significant. We expect similar results from our continued efforts to obtain empirical data from larger C^{++} programs.

7 Conclusions

We have presented a polynomial-time approximate technique to perform program-point-specific, interprocedural type determination and aliasing for C^{++}. We have shown the theoretical difficulty of this problem and demonstrated the utility of its solution in virtual function name resolution. Ours is the first algorithm which accounts for pointers and virtual functions in C^{++} without making gross approximations. We have implemented a prototype for the algorithm and presented initial empirical results on type determination and alias analysis of C^{++} programs. We plan to extend our work to solve other analysis problems useful for applications such as debugging and testing in a C^{++} programming environment.

Acknowledgements

We thank Rakesh Ghiya for his input in the design of the algorithms presented in this paper. We also thank William Landi of Siemens Corporate Research for his comments on this work, and for his implementation of the C aliasing algorithm at Rutgers University. Thank are due to Tata Consultancy Services, Pune for letting us use the MasterCraft C^{++} source code as front end for out prototype implementation. We are indebted to Ashok Sreenivas, R. Venkatesh and Ulka Shrotri of TRDDC for their feedback on the algorithms and invaluable help in interfacing MasterCraft with our implementation. Finally, we thank Tom Marlowe and Xiang Zhang for ensuring the completeness and clarity of new terminology introduced in this paper.

References

[APS93] Ole Agesen, Jens Palsberg, and Michael Schwartzbach. Type inference of SELF: Analysis of objects with dynamic and multiple inheritance. In *ECOOP '93 Conference Proceedings*, pages 247–267, July 1993.

[BCC+94] M. Burke, P. Carini, J-D. Choi and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings of the Seventh International Workshop on Languages and Compilers for Parallel Computing*, August 19994.

[Bur90] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data flow analysis. In *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[CBC93] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, January 1993.

[CG94] B. Calder and D. Grunwald. Reducing indirect function call overhead in C^{++} programs. In *Proceedings of the Twenty First Annual ACM Symposium on Principles of Programming Languages*, January 1994.

[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.

[Cal88] D. Callahan. The program 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.

[MGH94] M. Emami, R. Ghiya and L. 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 242–256, June 1994.

[FW85] P. G. Frankl and E. J. Weyuker. A data flow testing tool. In *Proceedings of IEEE Softfair II*, December 1985.

[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.

[HCU94] U. Hölzle, C. Chambers and D. Ungar. Optimizing dynamically-dispatched calls with tun-time type feedback. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 326–336, June 1994.

[HK92] Mary Hall and Ken Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, Vol. 1, No. 3, September 1992.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 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.

[HS94] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. In *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

[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 approximate 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*, June 1993.

[Lak93] Arun Lakhotia. Constructing call multigraphs using dependence graphs. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, January 1993.

[Lan92] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[Lar92] J. M. Larcheveque. Interprocedural type propagation for object-oriented languages. In *proceedings of the Fourth European Symposium on Programming (ESOP '92)*, February 1992.

[MLR+93] T. J. Marlowe, W. A. Landi, B. G. Ryder, J. Choi, M. Burke, and P. Carini. Pointer-induce aliasing: A clarification. *ACM SIGPLAN Notices*, 28(9), September 1993.

[Mey81] E. M. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, pages 219-230, January 1981.

[OW91] T. J. Ostrand and E. Weyuker. Data flow based test adequacy analysis for languages with pointers. In *Proceedings of the 1991 Symposium on Software Testing, Analysis and Verification (TAV4)*, October 1991.

[PLR94] H. D. Pande, W. Landi and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. In *IEEE Transactions on Software Engineering*, SE-20(5):385–403, May 1994.

[PR94] H. D. Pande and B. G. Ryder. Static type determination for C^{++}. In *Proceedings of USENIX Sixth C^{++} Technical Conference*, pages 85–97, April 1994.

[PRL91] H. D. Pande, B. G. Ryder and W. Landi. Interprocedural def-use associations in C programs. In *Proceedings of the 1991 Symposium on Software Testing, Analysis and Verification (TAV4)*, October 1991.

[PS91] Jens Palsberg and Michael Schwartzbach. Object-oriented type inference. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 146–161, October 1991.

[Pan94] Hemant D. Pande. Interprocedural compile time analysis of C and C^{++} systems. *PhD Thesis, Department of Computer Science, Rutgers University*, in preparation, 1994.

[Par92] Ramesh Parameswaran. Interprocedural alias and type analysis for pointers. *Masters Thesis, Department of Computer Science, University of Wisconsin - Madison*, 1992.

[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.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, ed. S. S. Muchnick and N. D. Jones, Prentice-Hall, Englewood Cliffs, NJ. Pages 189–233, 1981.

[SS92] Mario Suedholt and Christopher Steigner. On interprocedural data flow analysis for object oriented languages. In *Proceedings of the International Conference on Compiler Construction, Germany*, 1992.

[Suz81] Norihisa Suzuki. Inferring types in smalltalk. In *Eighth Symposium on Principles of Programming Languages*, pages 187–199, January 1981.

[VHU92] Jan Vitek, R. Nigel Harspool and James S. Uhl. Compile-time analysis of object oriented programs. In *Proceedings of the International Conference on Compiler Construction, Germany*, 1992.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[YHR92] W. Yang, S. Horwitz and T. Reps. A program integration algorithm that accommodates semantic preserving transformations. In *ACM Transactions on Software Engineering and Methodology*, 1(3):310–354, July 1992.

[ZR94] X. Zhang and B. G. Ryder. Complexity of interprocedural function pointer aliasing analysis. *Laboratory of Computer Science Research Technical Report LCSR-TR-233, Rutgers University*, October 1994.