Static Type Determination for C++*

Heman D. Pande†
Tata Research Development and
Design Centre
1 Mangalas Road, Pune-411050, India

Barbara G. Ryder
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903
ryder@cs.rutgers.edu

LCSR-TR-197-A

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 methods and pointers. We highlight the significance of type determination in improving code efficiency and precision of other static analyses. We present a safe, approximate algorithm for C++ programs with single level pointers, using the conditional analysis technique [LR91]. We discuss the generalization of our approach to analyze programs with multiple levels of pointer dereferencing.

1 Introduction

Recent emphasis in the static analysis community has been on expanding compile time analysis to include interprocedural information [Bur90, Cal88, CBC93, MLR+93, CK88, CK89, HS90, HRB90, LRZ93, Mey81]. Historically, compile time analysis has been used in intraprocedural context for code optimizations. The emphasis is shifting towards including the use of interprocedural static analysis in all phases of the software life cycle including debugging, integration and testing [FW85, HS89, HRB90, Lak91, OW91, RW85, Wei84, YHR90]. However, until recently, software analysis 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 and introduced a new technique for interprocedural analysis of C programs [LR91]. They have also developed a safe, approximate algorithm to solve the aliasing problem for a restricted subset of C which excludes pointers to functions, casting*, 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 [PRL91, PLR94], 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 employ the static analysis techniques beneficially to C++ programs. We have concentrated our efforts on developing new techniques to handle most of the features distinguishing C++ from C such as inheritance and virtual methods (object-orientedness), subtyping and overloading (polymorphism). The most significant C++ feature affecting compile time analysis is *virtual methods*. With virtual methods, it is the type of the receiver at an invocation site which dynamically determines the method to be invoked. With static *type determination*, such a late binding may be replaced by a function call to an appropriate method, or inlined code in suitable circumstances, thereby eliminating the overhead of late binding and improving the execution efficiency. Recent empirical studies of dynamic behavior

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

†Author's current address: Department of Computer Science, Rutgers University, New Brunswick, NJ 08903.

email: pandeh@cs.rutgers.edu

Although simple casting for `p = malloc()` is handled.
of actual C++ programs indicate there is an opportunity to avoid late bindings in many cases [CG94]. Additionally, a statically determined list of possible types for a receiver would focus further analyses only on selected methods, rather than the entire pool of methods with the same name. Exclusion of the statically invocable methods from analysis would eliminate their spurious side effects, thereby improving the precision of subsequent analyses.

This paper describes initial results of our research in type determination. Ours is the first algorithm for type determination which uses the technique of data flow analysis without making gross approximations for the distinguishing C++ features mentioned above. The contributions of our work can be seen at two levels: (i) increased efficiency and precision of other compile-time analyses and (ii) improved run-time performance of the programs analyzed. In general, type determination cannot be done separately due to its interaction with aliasing. We present a type determination algorithm for the restricted case of single level pointers where the two problems are separable. Details of the generalized version appear in [PR94].

It is true that all C++ programs can be source-to-source transformed into C programs. Thus, 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 method mechanism can be expressed in terms of function calls through arrays of function pointers. Algorithms which attempt precise analysis in the presence function pointers and procedure variables handle only a limited usage of such constructs or resort to possibly worst case exponential analyses [Ghi92, HK92, Lok93, Ryd79]. This motivated us to develop new techniques to analyze virtual methods 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.

Overview: In Section 2, we mention related work, especially for analysis of C++. We introduce the program representation, terminology and theoretical problem complexity in Section 3. We show that the problem is NP-hard in the presence of single level pointers; the intractability of the problem is inherent to this restricted case without generalizing to multiple level pointers. In Section 4, we describe a polynomial time algorithm to determine points-to information, i.e. the class 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. In Section 5 we briefly describe the interaction between type determination and aliasing in the general case. Finally, we conclude by summarizing our results.

2 Related Work

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 methods, 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 the dynamically typed language SELF uses customization, iterative type analysis and inline caching to replace dynamic binding by procedure calls or inline code [CU89, CU90, HCU91]. The algorithm by Lacheveque [Lar92] is rendered imprecise by the fact that it factors out the side effects of method 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 C++ context. We show that aliasing and type determination are inseparable in the general case, therefore a factored approach 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 methods are invocable [Par92]. He uses the precalculated alias information for type determination. Suedholt and Steigner [SS92] use a concept of representant virtual method to keep information about all the virtual methods with the same name. This approach leads to the loss of context which distinguishes one virtual method from others. Vitek et al [VH92] 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 method 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 methods comprising the program [LR92]. Formally, an ICFG is a triple $\langle N, E, \rho \rangle$ where $N$ is the set of nodes, $E$ is the set of edges and $\rho$ is the entry node for main. $N$ contains a node for each simple statement in the program, an entry and exit for each method, 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 method has returned. (We will use the terms call and invocation interchangeably.) For a non-virtual method call, we represent the control flow into the called method by an interprocedural edge from call to the corresponding entry node. Similarly, we represent the return of control from the called method by an interprocedural edge from the exit node to the return node. However, virtual method invocation makes it impossible to determine before analysis the correspondence between a call and entry since the method invoked depends on the type of the receiver at the call site. Establishing the interprocedural edge(s) from a call node representing virtual method 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

- We define an ICFG path from $\rho$ as realizable if, whenever a method on this path returns, it returns to the call site which invoked it. Not all paths in the ICFG are realizable. Our analysis tries to restrict itself to realizable paths since unrealizable paths do not correspond to any possible execution sequence.
- Objects are locations that can store information, and object names provide ways to refer to them. An object name is a variable name and a possibly empty sequence of dereferences and member accesses.
- An alias occurs when there exists a realizable path to a program point, such that two or more names exist for the same location at that program point. We represent aliases by unordered pairs of object names (e.g. $<v, *p>$). The order is unimportant since aliases are symmetric.
- The Type Determination Problem involves calculating the type of the object pointed to by a pointer at a program point as a result of some execution that ends at that program point.
- A pointer-type pair $<p \Rightarrow C>$ holds on the realizable path $\rho n_1 n_2 \ldots n_i$ if $p$ points to an object of class $C$ after execution of program point $n_i$, whenever the execution defined by that path occurs.

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.

We prove the theorem by a polynomial reduction of the NP-Complete problem of 3-Satisfiability to type determination in the presence of single level pointers and virtual functions in C++ [PR94]. □

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 An Approximate Type Determination Algorithm

Our algorithm uses the idea of conditional analysis as found in [LR91]. Execution flow in a method is analyzed by assuming information that can hold at the entry of the method. Thus, in a sense, 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. The algorithm described here is applied only to C++ programs allowing a single level of dereferencing with pointers. We assume in the following description that the receiver of a method call is the first actual parameter and the corresponding formal is denoted by this.

We define a predicate points-to with the following interpretation: points-to(n, assumption, fact) == true if (i) there exists a realizable path to the entry node of the procedure containing node n, on which assumption holds; and (ii) given that (i) is true, there exists a realizable path from the entry node to n on which fact holds. The assumption can either be 0 or a pointer-type pair while fact is a pointer-type pair.

4.1 A Running Example

Before discussing the algorithm, we list a program segment in Figure 1. 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 Figure 1 as a running example for the algorithm description.

4.2 Practical Issues

At node n9 in Figure 1, the pointer q is made to point to an object of class Base, and then immediately used at node n10 as the receiver for a virtual method 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 assignment at node n2 and printing hello world at n3 will not appear as possible side effects of the invocation at node n10. As another significant implication, our algorithm will be able to determine that n11 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 n11 can be considered non-virtual. Given the potential disparity in side effects of virtual methods which share the same name, type determination can significantly improve the precision of analysis.

Resolving a virtual method 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 n12, 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 method baz() from class Base without redefining it, n12 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 method, 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 method 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
class Base {
public:
    virtual foo ()
    virtual bar ()
    virtual baz ()
    } *a, *b, *p, *q;
Base::foo () {
    n1: a = new Base;
}
Base::bar () {
}
Base::baz () {
}
}...

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

Figure 1: Example of Type Determination Algorithm

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. If there exists a path to node n on which <ptr ⇒ C> holds during some execution, our algorithm will report a true predicate points-to(n, APT, <ptr ⇒ C>) for some APT, guaranteeing the safety of calculation. However, owing to the intractability of the problem, our polynomial time algorithm is justifiably approximate, reporting an overestimate of the actual solution.

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 algorithm. We refer to this action as make-true and denote it in the algorithm by “make-true (points-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 the work is performed only for points-to(n, APT, <ptr ⇒ C>) which are to 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:
for each node \( n \) in the ICFG

1. \( n : p = \text{new } t : \)
   - \textit{make-true} \( \text{points-to}(n, \emptyset, <p \Rightarrow t>) \)

2. \( n : p = &r : \)
   - \textit{make-true} \( \text{points-to}(n, \emptyset, <p \Rightarrow \text{type}(r)>) \)
   - where \( \text{type}(r) \) returns the type of object name \( r \).

3. \( n : \text{foo}(\text{param}_1, \ldots, \text{param}_m) : \)
   - \textit{make-true} \( \text{points-to}(\text{entry}_{\text{foo}}, <p \Rightarrow \text{type}(r)>, <p \Rightarrow \text{type}(r)>) \)
   - where \( \text{param}_i \) is of the form \&\( r \) with pointer variable \( p \) as the corresponding formal, and the call is non-virtual.

Figure 2: Introduction Phase

while worklist is not EMTPY

remove \((n, APT, <\text{ptr} \Rightarrow C>)\) from worklist

if \( n \) is a call node
   - \textit{type-implies-type-from-call} \( \text{call}, APT, <\text{ptr} \Rightarrow C> \)
else if \( n \) is an exit node
   - \textit{type-implies-type-from-exit} \( \text{exit}, APT, <\text{ptr} \Rightarrow C> \)
else
   - \textit{type-implies-type-through-other} \( n, APT, <\text{ptr} \Rightarrow C> \)

Figure 3: Propagation Phase

\[
\text{pointer-type-info}(n) = \{ <\text{ptr} \Rightarrow C> \mid \exists APT \text{points-to}(n, APT, <\text{ptr} \Rightarrow C>) = \text{true} \}. 
\]

Conceptually, we start with no information at any of the ICFG nodes by initializing each possible \textit{points-to} predicate to \textit{false}. We also initialize the worklist to EMTPY. The time complexity of the initialization of the entire \textit{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, PLR94].

The first entries in the worklist come from the introduction phase. During this phase we make \textit{true} certain predicates at a node by looking at the local information available in the node itself. Figure 2 lists the nodes examined in the introduction phase and their associated actions. Note that in item 3 we restrict ourselves to non-virtual method calls. Without the knowledge of the receiver type, we can make no educated guesses about the method invoked. We handle virtual method calls during the propagation phase.

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

\[\textit{make-true} \text{points-to}(n4, \emptyset, <p \Rightarrow \text{Base}>)\]

Since there exists a path \textit{entry}_{\text{main}}, n5 at the end of which \(<p \Rightarrow \text{Derived}>\) holds without assuming any information at \textit{entry}_{\text{main}}, using item 1 we also have

\[\textit{make-true} \text{points-to}(n5, \emptyset, <p \Rightarrow \text{Derived}>)\]

At node \( n6 \), using item 2 and the fact that \( r \) is an object of class \textit{Derived},

\[\textit{make-true} \text{points-to}(n6, \emptyset, <s \Rightarrow \text{Derived}>)\]

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 3 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 \( ptr, m : ptr = \ldots \), 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): \( m : ptr' = ptr; \) or \( m : ptr' = (\text{Class E})* ptr; \)

\[
\text{make-true} \ (\text{points-to}(m, APT, <ptr' \Rightarrow C>)) \quad \text{and} \quad \text{make-true} \ (\text{points-to}(m, APT, <ptr \Rightarrow C>)).
\]

Type casting appears in the latter node so that the assignment is type-correct, but it is unimportant for our analysis since \( ptr' \) 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: \text{make-true} \ (\text{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 \text{points-to}(n6, \emptyset, <p \Rightarrow \text{Base}>) and \text{points-to}(n5, \emptyset, <p \Rightarrow \text{Derived}>). Propagating this information to the successor node \( n6 \) which preserves the type of \( p \) we \text{make-true} both \( \text{points-to}(n6, \emptyset, <p \Rightarrow \text{Base}>) \) and \( \text{points-to}(n6, \emptyset, <p \Rightarrow \text{Derived}>) \)

Using further applications of **case 3**, the information at \( n6 \) propagates to its successors as

\[
\text{points-to}(\text{call}, 0, <p \Rightarrow \text{Base}>) \quad \text{points-to}(\text{call}, 0, <p \Rightarrow \text{Base}>) \\
\text{points-to}(\text{call}, 0, <p \Rightarrow \text{Derived}>) \quad \text{points-to}(\text{call}, 0, <p \Rightarrow \text{Derived}>) \\
\text{points-to}(\text{call}, 0, <s \Rightarrow \text{Derived}>) \quad \text{points-to}(\text{call}, 0, <s \Rightarrow \text{Derived}>)
\]

**type-implies-type-from-call** (\( \text{call}, APT, <ptr \Rightarrow C> \))

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 \( \text{call} \) represents a non-virtual method invocation. As we already saw, \text{points-to}(\text{call}, 0, <s \Rightarrow \text{Derived}>)

Since \( s \) is visible in the called method \( \text{Derived} :: \text{bar}() \), we \text{make-true}

\[
\text{points-to}(\text{entry} \text{Derived} \text{:: bar}, <s \Rightarrow \text{Derived}>, <s \Rightarrow \text{Derived}>)
\]

At the call site \( n7 \), \( s \) is the first actual parameter and corresponds to the formal this of \( \text{Derived} :: \text{bar}() \). Since \text{points-to}(\text{call}, 0, <s \Rightarrow \text{Derived}>) is true, we \text{make-true}

\[
\text{points-to}(\text{entry} \text{Derived} \text{:: bar}, <\text{this} \Rightarrow \text{Derived}>, <\text{this} \Rightarrow \text{Derived}>)
\]

If \( ptr \) is not visible in the called method, the type pointed to by \( ptr \) cannot change\(^2\). In this case we propagate the predicate \text{points-to}(\text{call}, APT, <ptr \Rightarrow C>) directly to the corresponding return node as \text{points-to}(\text{return}, APT, <ptr \Rightarrow C>)

\(^2\) This is true because we only have single level pointers.
**case 2:** Call is virtual. Suppose the call node is: \( n : \text{rec} \rightarrow \text{fun}(\ ) \).

The entry nodes to which the effects of the given worklist entry \( \text{fun} \) have to be propagated depend on the type(s) of objects the receiver \( \text{rec} \) 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 method to be invocable (case 2.1), and

(ii) a method 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:** \( \text{ptr} == \text{rec} \) (i.e. \( \text{ptr} \) is the same variable as the receiver \( \text{rec} \))

1. The effect of this points-to needs to be propagated only to the method invocable when the receiver points to an object of class \( C \). In the example, points-to(call_{as}, \emptyset, <p ⇒ Base>) propagates to entry_{Base::foo} as

   \[
   \text{make-true}(\text{points-to(entry}_{Base::foo}, <p ⇒ Base>, <p ⇒ Base>))
   \]

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

   \[
   \text{make-true}(\text{points-to(entry}_{Derived::foo}, <p ⇒ Derived>, <p ⇒ Derived>))
   \]

   but not to entry_{Base::foo}.

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

   a) If the method call involves passing of an object address \( \text{ptr} \) as an actual to a pointer formal \( f \): make-true (points-to(entry_{p}, <f ⇒ type(r)>, <f ⇒ type(r)>)). Note that this case could not be handled in the introduction phase, as the invocability of this method from call node \( n \) was not known then.

   b) For each points-to(call_{APT'}, <ptr' ⇒ E>) where \( \text{ptr'} \neq \text{rec} \):

   Determine the corresponding entry and perform actions as in case 1. Thus while propagating points-to(call_{as}, \emptyset, <q ⇒ Base>), we also propagate the predicate points-to(call_{as}, \emptyset, s ⇒ Derived>, already true at call_{as}, with

   \[
   \text{make-true}(\text{points-to(entry}_{Base::foo}, s ⇒ Derived>, s ⇒ Derived>))
   \]

**case 2.2:** \( \text{ptr} \) and \( \text{rec} \) are distinct variables:

Suppose the points-to information currently available about the receiver \( \text{rec} \) at the given call node is:

\[
\text{points-to(call}_{APT1}, <\text{rec} ⇒ C1>) = \text{true} \quad \text{and} \quad \text{points-to(call}_{APT2}, <\text{rec} ⇒ C2>) = \text{true}
\]

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

   \[
   \text{make-true}(\text{points-to(entry}_{Base::foo}, s ⇒ Derived>, s ⇒ Derived>))
   \]

   \[
   \text{make-true}(\text{points-to(entry}_{Derived::foo}, s ⇒ Derived>, s ⇒ Derived>))
   \]

If the pointer variable \( \text{ptr} \) is not visible in any one (or more) of these invocable methods, the predicate on the worklist propagates directly to the return node by make-true (points-to(return, APT, \text{ptr} ⇒ C>)).
type-implies-type-from-exit \((exit, APT, <pfr \Rightarrow C>)\)
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 method \(\text{fun}\)\) and the return nodes corresponding to exit be \(r_1, r_2, \ldots, r_k\) at the instant of processing this worklist entry. New return nodes may be added later, when the method is determined to be invocable from other virtual method call sites. We do not consider them at this time. As explained earlier, when a new virtual method is determined to be invocable from a call node we propagate the effects of this call from the exit of the called method 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 \(c_1, c_2, \ldots, c_k\). We do the following for each return node \(r_i\):

If \(pfr\) is not visible in the method 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 \(pfr\) is visible in the method containing the return node \(r_i\) we have the following cases:

case 1: If APT is non-\(\emptyset\), implying that APT holds at entry in order that \(pfr\) points to an object of class \(C\) at exit. Each call node \(c_i\) responsible for imposing APT at entry in turn leads to \(pfr \Rightarrow C\) holding at its corresponding return node \(r_i\).

If APT is imposed at entry of the invoked method without requiring any points-to predicate at the call node \(c_i\) (i.e. \(\text{points-to}(\text{entry}, APT, APT)\) was made \(\text{true}\) during introduction phase), we simply propagate \(<pfr \Rightarrow C>\) to \(r_i\). In this case: make-true (points-to\((r_i, \emptyset, <pfr \Rightarrow C>)\)). On the other hand, suppose it took points-to\((c_i, APT''\), APT\) to impose APT at the entry, we have: make-true (points-to\((r_i, APT'', <pfr \Rightarrow C>)\))

In our example, suppose we are propagating points-to\((exit_{Base;foo}, <p \Rightarrow Base>, <p \Rightarrow Base>)\). We have two return nodes viz. \(\text{return}_{n_8}\) and \(\text{return}_{n_{10}}\). Since it takes points-to\((\text{call}_{n_8}, \emptyset, <p \Rightarrow Base>)\) to impose \(<p \Rightarrow Base>\) at entry\(Base;foo\). Using the information thus available at \(\text{call}_{n_8}\) and \(exit_{\text{Base;foo}}\):

make-true (points-to\((\text{return}_{n_8}, \emptyset, <p \Rightarrow Base>)\))

As there is no assignment to \(p\) on any path from \(\text{call}_{n_8}\) to \(\text{call}_{n_{10}}\). points-to\((\text{call}_{n_{10}}, \emptyset, <p \Rightarrow Base>)\) is \(\text{true}\). This predicate also imposes \(<p \Rightarrow Base>\) at entry\(\text{Base;foo}\). Using this information available at \(\text{call}_{n_{10}}\) while propagating points-to\((exit_{\text{Base;foo}}, <p \Rightarrow Base>, <p \Rightarrow Base>)\):

make-true (points-to\((\text{return}_{n_{10}}, \emptyset, <p \Rightarrow Base>)\))

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

4.4 Algorithm Complexity
The following considerations are significant while determining the complexity of our algorithm.

1. The values of \(\text{points-to}\) 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 inraprocedural 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 \(\text{points-to}\) predicate. For example, \(\text{points-to}(n, APT_1, <p \Rightarrow C>)\), \(\text{points-to}(n, APT_2, <p \Rightarrow C>)\) 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 average 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.
alias-implies-alias
\<q, x\>
\ n : p = q;
\<p, x\>

alias-implies-type
\<p, r\>
\ n : p = \$obj;
\<p \rightarrow type(obj)\>

type-implies-type
\<q \Rightarrow C\>
\ n : p = q;
\<p \Rightarrow C\>

1

5 Extension for multiple level pointers

In the presence of only single level pointers, a pointer cannot be aliased to another pointer. As a result, when a pointer changes its type (to point to an object of another type), it is only this pointer and nothing else which changes type. Type determination impinges on aliasing since the receiver types decide which virtual method is invoked at a call site, and the invoked method can affect aliasing. Aliasing plays no part in type determination. However such a separation does not occur in case of 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 type pair \( \<p \Rightarrow type(r)\> \). In the absence of information that the alias pair \( \<p, q\> \) holds at node \( n \), we would not be able to infer \( points-to(n, o, \<q \Rightarrow type(r)\>) \), 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.) In Figure 4, we illustrate some intraprocedural aspects of the interaction between type determination and aliasing. The fact to be propagated appears first, followed by node \( n \), and then we list an appropriate resulting fact.

Our algorithm described in Section 4.3 can extend to handle the general case of multiple level pointers, however it involves interleaved type determination and aliasing calculations. We have implemented a prototype for the general algorithm to perform type determination and aliasing together for C++ programs with multiple level pointer dereferencing. A detailed description of the general algorithm and preliminary implementation results can be found in [PR94]. Although current results are encouraging, we are extending the implementation to analyze a broader range of larger C++ programs in order to make a more definitive empirical assessment of our algorithm.

6 Conclusions

We have presented a polynomial-time approximate technique to perform program-point-specific, interprocedural type determination for C++. We have shown the theoretical difficulty of this problem and demonstrated the utility of its solution in virtual method name resolution. This is the first static analysis algorithm for type determination which accounts for pointers and virtual methods without gross approximations. For ease of explanation we have restricted the problem domain to C++ programs with only single level pointer dereferencing, where the virtual name resolution is separable from other analyses. We are currently gathering data from our implementation of the general algorithm to determine its practicality. We also plan to extend our work to solve other analysis problems useful for applications such as debugging and testing in a C++ programming environment.
Acknowledgements

This research benefited greatly from discussions with William Landi of Siemens Corporate Research. We also thank Rakesh Ghiya for his input in the design of this algorithm. We are indebted to Ashok Sreenivas, R. Venkatesh and Ulka Shroti of TRDDC for their feedback on the algorithm and invaluable help in the implementation. Finally we thank Tata Consultancy Services, Pune for letting us use the MasterCraft C++ source code as front end for our prototype implementation.

References


