Laboratory of Computer Science Research Technical Report LCSR-TR-250-A, October 1995

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 phone: (908)445-{4070,3699}

Abstract

Determining the type of an object to which a pointer may point at each statement during execution is the goal of static type determination. We prove NP-hardness of type determination and aliasing for C++. We show the interdependence of the two problems for general-purpose pointers and present a polynomial approximation algorithm to solve the combined problem. We include empirical results to demonstrate the feasibility of our analysis.

1 Introduction

Recently, researchers have concentrated on developing practical static analyses that handle languages with general-purpose pointer usage, such as C [BCC+94, CBC93, MLR+93, EGH94, LRZ93, PLR94, Ruf95, WL95]. Our efforts are focused on developing new techniques to handle those features distinguishing C++ from C, such as inheritance and virtual functions (i.e., object-orientedness), subtyping and overloading (i.e., polymorphism). Most significant are virtual functions, because the type of receiver at a virtual call site dynamically determines the function to be invoked. Static type determination enables us to replace late binding with a direct call to an appropriate function, or with inlined code in suitable circumstances. (*In the full paper we will supply an example of optimizations enabled by our analysis.*)

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]. A uniquely resolved call site would eliminate pipeline delays, as the target of the call is unambiguously known. Short-listing the functions may allow the compiler to replace the late binding mechanism of virtual call with appropriate function calls within a decision statement. Branch prediction techniques may then be applied to improve the execution performance of the program. Additionally, short-listing can focus further analyses on selected functions, rather than on the entire pool of functions with the same name, potentially saving analysis time. Also, exclusion of the statically non-invocable functions from analysis can eliminate spurious side effects, thereby improving the precision of subsequent analyses.

Last year, we presented a static type determination algorithm for C++ programs restricted to using only single level pointers [PR94]. In this context, type determination can be solved independent of aliasing. However, in the presence of multiple level pointers, these two problems cannot be solved separately, but are interdependent. We have designed a combined approximation algorithm that solves type determination and aliasing simultaneously for C++ programs. Ours is the first data flow technique for these problems in an object-oriented language.

Our major contributions are

1. the first polynomial time combined approximation algorithm for aliasing and type determination in an object-oriented language,

*This research was supported, in part, by funds from NSF grants GER90-23628, CCR-92-08632 and Siemens Corporate Research.

2. empirical results on actual C^{++} programs to validate our analysis approach, and
3. a theoretical proof of the NP-hardness of these problems.

Related Work[1] Recent research in optimizations for object-oriented languages has mainly concentrated on two analysis techniques: type feedback (run time) and type inference (compile time). A comparison of the two techniques appears in [AH95]. (*The following abbreviated discussion of related work will be expanded in the full paper.*)

Notable recent research on type feedback includes work by Calder and Grunwald [CG94], Hölzle and Ungar [HU94], and Dean *et al.* [DCG95].

We briefly discuss the type inference techniques based on class hierarchy, type constraints, function pointer analysis and control flow analysis of higher order languages. The type determination aspect of our work can be viewed as type inference using data flow analysis.

Class hierarchy information is used at link time by Fernandez to replace method invocations with direct calls in Modula-3 programs [Fer95]. Diwan *et al.* apply hierarchy information and a limited form of static analysis to optimize Modula-3 programs [DMM95].

Agesen *et al.* present a polynomial time constraint based type inferencing algorithm for SELF [APS93]. Kumar *et al.* improve on this technique by utilizing the *Static Single Assignment* form of object-oriented programs [KAI95]. Plevyak and Chien describe an incremental constraint based type inference technique for Concurrent Aggregates, a concurrent object-oriented language [PC94]. While constraint based inferencing is most suitable for purely dynamic, untyped languages like SELF and ours for typed languages like C^{++}, the two approaches may supplement each other for languages which combine these separate domains.

Since virtual function calls in C^{++} can be modeled using function pointers in C, algorithms which handle them [BCC+94, EGH94, WL95, Wei80] may be applied towards analysis of C^{++}. Nevertheless, these approaches i) have a different emphasis and are ill-tuned to function pointer analysis and/or ii) have impractical worst case complexity, and are unsuitable in C^{++}context where virtual functions are ubiquitous.

Techniques for flow analysis of higher order languages like Scheme [Har89, JM79, Shi90] may be adapted for analyzing function pointers (and hence C^{++}); however, they do not have acceptable complexity for reasonable precision on real programs. Jagannathan and Wright compare various existing techniques for analysis of such languages and motivate the need for alternative data flow techniques [JW95].

Earlier work on dynamically dispatched methods includes [CU89, CU90, HCU91, Lar92, PS91, Par92, SS92, Suz81, VHU92].

2 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 [LR91, PR94]. Formally, an ICFG is a triple ($\mathcal{N}$, $\mathcal{E}$, ρ) where $\mathcal{N}$ is the set of nodes, $\mathcal{E}$ is the set of edges and ρ is the *entry* node for `main`. $\mathcal{N}$ contains a node for each simple statement in the program, an *entry* and *exit* node for each function, and a *call* and *return* node for each invocation site. An intraprocedural edge into a *call* node represents the execution flow into an invocation site, while an intraprocedural edge out of a *return* node represents control flow from an invocation site once the invoked function has returned[2]. For a non-virtual function call, we represent the control flow into the called function by an interprocedural edge from a *call* node to the corresponding *entry* node. Similarly, we represent the return of control from the called function by an interprocedural edge from the *exit* node to the *return* node. However, virtual function invocation makes it impossible to determine the correspondence

[1] For related work on pointer-induced aliasing for imperative languages, see [LR92].
[2] We use the terms *call* and *invocation* interchangeably.

between a *call* and *entry* before analysis, since the function invoked depends on the type of the receiver at the call site. Establishing the interprocedural edge(s) from a *call* node representing a virtual function invocation to appropriate *entry* node(s) and from *exit* node(s) to the corresponding *return* node(s), (i.e., resolving virtual functions), is a major goal of the algorithm presented in this paper.

Terminology

- An ICFG path is *realizable* if, whenever a called function on that path returns, it returns to the corresponding *return* node of the call site which invoked it [LR91]. Not all ICFG paths 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[3]. Intuitively, the first and the last node on a balanced path belong to the same function. Moreover, they are in the same incarnation of that function since every called function on the path (perhaps recursively) must return control before the path terminates.
- *Objects* correspond to locations that can store information; *object names* provide ways to refer to them. We associate names with static memory locations and dynamic (i.e., heap) locations (created by *new*). For static storage, the name-storage association is created through a variable declaration statement. For each heap location, we create a name new_{pp} where pp is the program point representing the creation site for the location. An *object name* is a variable name or a heap location name, and a possibly empty sequence of dereferences.
- 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. 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 an execution to that program point. We represent this information by a pair consisting of a pointer and an associated type (e.g. $<p \Rightarrow C>$), called a *pointer-type* pair.
- A 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`. Non-consistent paths do not correspond to any possible execution sequence.
- The *precise*[4] solution for static type determination and aliasing at a program point is a set of pointer-type and alias pairs, each of which is a result of an execution on 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 and aliasing is NP-hard.*

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

(*The theorem proof works by reducing the 3-SAT problem to type determination and aliasing, and will appear in the full paper. An easy corollary follows, since the theorem involves a subproblem of the general type determination and aliasing problem.*)

---

[3] We defined the terms *realizable* and *balanced* paths independently [LR91, PR94a], and have only recently found that the ideas already existed in literature, referred to as *valid* and *complete* respectively in [SP81]. A more recent paper [RHS95] also addresses these issues.

[4] Under the standard assumption of static analysis that all intraprocedural paths are executable [Bar78].

3 Approximate Type Determination and Aliasing Algorithm

Aliasing and Type Determination In programs restricted to single level pointers, one pointer cannot be aliased to another, as this requires multiple levels of indirection [LR91]. As a result, when a pointer changes its value (to point to an object of another type), it does not affect the value of any other pointers [PR94]. In this context, type determination impinges on aliasing since the receiver types decide which virtual function is invoked at a call site, and the invoked function can affect aliasing. Nevertheless, aliasing plays no part in type determination.

Such a separation does not occur when we allow multiple level pointers. As an example, the ICFG node "`m : p = &q;`" creates alias $<*p, q>$. Suppose subsequently on an execution path, "`n : *p = &r;`" creates pointer-type pair $<*p \Rightarrow \mathit{type}(r)>$. In the absence of information that the alias pair $<*p, q>$ holds before node `n`, we would not be able to infer $<q \Rightarrow \mathit{type}(r)>$ at `n` and the type determination would be rendered incorrect and unsafe. Thus, aliasing affects type determination and *vice versa.* In this section, we formulate the combined problem and state our tractable approximation of it. Then, we describe our algorithm at a high level, aided by examples.

Problem Formulation *Conditional analysis* [LR91] involves analyzing execution flow in a function, assuming certain information holds at the entry of the function. Formally,

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

  > Given that all the alias and pointer-type pairs in S hold at `e`, the execution defined by the subpath from `e` to `call` implies a pointer-type pair $<rec \Rightarrow C>$ at `call` such that the virtual function represented by `entry` is invocable from `call`.

  We denote such a path by $\mathcal{P}_{n,S}$.
- We define the *conditional type determination problem* at node `n`: There exists a conditionally consistent path with respect to a set S from `e` to node `n` on which a pointer-type pair PT holds, **and** there exists a consistent path from ρ to `e` on which every pair in the set S holds.
- Similarly, we define the *conditional aliasing problem* at node `n`: There exists a conditionally consistent path with respect to a set S from `e` to node `n` on which an alias AP holds, **and** there exists a consistent path from ρ to `e` on which every pair in the set S holds.

Approximation Formulation Since the above formulation is computationally intractable, we approximate the joint solution of these two related problems by considering an approximate assumption set S' to contain at most one alias or pointer-type from S, chosen arbitrarily. We use the pair in S' (i) to approximate a conditionally consistent path to `n`, and (ii) as the only necessary assumption for conditional analysis[5]. This approximation leads to a safe overestimate of consistent paths and conditional analysis solution [Pan95].

We define two predicates, *points-to-type*[6] and *may-hold*, with the following interpretations reflecting this approximation. *points-to-type*$(n, AAPT, <p \Rightarrow C>)$ if there exists a consistent path from ρ to the *entry* node of the procedure containing node `n`, on which an alias or pointer-type pair $AAPT$ (if any)[7] holds **and** there exists a conditionally consistent path $\mathcal{P}_{n,\{AAPT\}}$ to `n` on which $<p \Rightarrow C>$ holds. Similarly, *may-hold*$(n, AAPT, <a,b>)$ if there exists a consistent path from ρ to the *entry* node of the procedure containing node `n`, on which $AAPT$ (if any) holds **and** there exists a conditionally consistent path $\mathcal{P}_{n,\{AAPT\}}$

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

[6] In our previous papers [PR94, PR94a], *points-to-type* was called *points-to.*

[7] AAPT may be $\emptyset$.

to `n` on which $<a, b>$ holds. For efficiency, we have designed our approximation algorithm so that work is performed only for *true*-valued *may-hold* and *points-to-type* predicates.

3.1 Algorithm Overview

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

The worst case complexity of our algorithm is polynomial in the number of ICFG nodes [Pan95]. However, we can show that in practice the algorithm runs in time linear with respect to the size of *may-hold* and *points-to-type* solution; we also have empirical corroboration of this claim. (*Further details will appear in the full paper.*)

A high level description of our algorithm appears in Figure 1. The algorithm has three main phases which are discussed using examples in Sections 3.3–3.5. Firstly, the predicates *points-to-type* and *may-hold* and the worklist are *initialized* (see Section 3.3). Secondly, we *introduce* certain *true*-valued predicates at pointer assignments (using **intra-alias-type-introduction**) and at parameter binding sites (using **inter-alias-type-introduction**) (see Section 3.3). These initial predicates are placed on the worklist. Thirdly, the algorithm performs the usual fixed point iteration, until the worklist is empty. That is, a predicate is removed from the worklist and *propagated* through successor nodes using appropriate functions determined by the node type. This propagation of information occurs in the `while` loop of Figure 1. Intraprocedural propagation is explained in Section 3.4; the interprocedural propagation functions (e.g., **alias-type-propagation-from-call**) are explained in Section 3.5. The propagation functions make additional predicates *true* and put them on the worklist. (*Sections 3.3–3.5 will be expanded in the full paper.*)

The calculation of a fixed point for *points-to-type* and *may-hold* is tantamount to the solution of a monotone data flow framework defined on a lattice whose elements are sets of [*assumption, alias/pointer-type*] tuples [Pan95]. Whenever a predicate becomes *true* for the first time, it is placed on the worklist. We refer to this action as **make-true**. Once marked *true*, a predicate stays *true*. A predicate goes on the worklist at most once, guaranteeing the termination of our algorithm.

3.2 Calculation of Approximate Assumption Sets

Before providing algorithm details, we briefly describe some auxiliary functions used to capture type and aliasing effects on entry of an invoked function from the types and aliases present at the invocation site. These functions calculate the approximate assumption sets described earlier. The first two functions, **bind0** and **type-bind0**, are used during the introduction phase (Section 3.3) and the rest during interprocedural propagation (Section 3.5). In these descriptions, `call` and `entry` represent ICFG nodes whereas *alias* and *pointer-type* are specific pairs.

bind0(`call`,`entry`) : This function calculates those *aliasing effects* from `call` to `entry` requiring no information from the predecessor(s) of `call`. For example, if $\&a$ is passed as an actual to the formal f, $<*f, a>$ is created at `entry` regardless of any aliases a may have at `call`; so $<*f, a> \in$ **bind0**(`call`,`entry`).

type-bind0(`call`,`entry`) : This function calculates those *type effects* from `call` to `entry` requiring no information from the predecessor(s) of `call`. 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-bind0**(`call`, `entry`).

bind(`call`,`entry`,*alias*) : This function represents the propagation of *alias* reaching the `call` to the corresponding `entry`. Depending on the actual-formal associations, *alias* may manifest itself at `entry` and/or

```
// initialization of information (Section 3.3)
lazily set all possible predicates to false;
set worklist to empty;
// introduction of aliases and pointer-type pairs (Section 3.3)
intra-alias-type-introduction ( ); // (Figure 3)
for each non-virtual call to entry
       inter-alias-type-introduction (call, entry);
// propagation of aliases and pointer-type pairs
while worklist is not empty
       remove (M, AAPT, APT) from worklist
       if M is a call node // (Section 3.5)
            alias-type-propagation-from-call (M, AAPT, APT);
       elseif M is an exit node // (Section 3.5)
            if APT is an alias pair
                  alias-implies-alias-from-exit (M, AAPT, APT);
            if APT is a pointer-type pair
                  type-implies-type-from-exit (M, AAPT, APT);
       else // intraprocedural propagation (Section 3.4)
            for each N ∈ successor (M)
                  if N is a pointer assignment // (Figure 4)
                       if APT is an alias pair
                            alias-implies-alias-thru-assign (N, M, AAPT, APT);
                            alias-implies-type-thru-assign (N, M, AAPT, APT);
                       if APT is a pointer-type pair
                            type-implies-type-thru-assign (N, M, AAPT, APT);
                            type-implies-alias-thru-assign (N, M, AAPT, APT);
                  // propagate directly through N
                  elseif APT is an alias pair
                       make-true(may-hold (N, AAPT, APT))
                  else
                       make-true(points-to-type (N, AAPT, APT))
```

Figure 1: A High Level Description of the Algorithm

may give rise to additional alias pairs. In Figure 2, $<*p, *q>$ is created at `n2` and reaches $call_{c1}$, resulting in:

$$\textbf{bind}(call_{c1}, entry_{C::foo}, <*p, *q>) = \{<*foo1, *foo2>, <*foo1, *q>, <*foo2, *p>, <*p, *q>\}$$

alias-bind(`call`,`entry`,*pointer-type*) : This function calculates the *alias effects* of *pointer-type* present at `call` which fall into the following two categories: aliases between appropriate dereferences of two formals (i.e., the first pair below) and aliases between the dereference of an actual and the corresponding formal (i.e., the last pair). In Figure 2, **alias-bind**($call_{c2}, entry_{C::bar}, <r \rightarrow d1 \Rightarrow B>$) includes:

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

type-bind(`call`,`entry`,*pointer-type*) : This function calculates the *type effects* of *pointer-type* present at `call` on `entry`. Depending on the actual-formal bindings at `call`, *pointer-type* may simply propagate to `entry` and/or may create a pointer-type pair involving the corresponding formal. In Figure 2,

```
class B { public:          class C { public:                          class D { public:
  int b1;                    int foo (int *foo1, int *foo2);            B *d1;
};                           int bar (D *bar1; B *bar2);              };
                           };

                           C *s;    D *r;
                           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);
                           }
```

Figure 2: Example for Binding Functions

`for each node n in the ICFG`
`If n is`
`n :` `p = new t;`
make-true($points\text{-}to\text{-}type(n, \emptyset, <p \Rightarrow t>)$)
make-true($may\text{-}hold(n, \emptyset, <p{\rightarrow}mem_k, new_n.mem_k>)$)[†]
`n :` `p = &r;`
make-true($points\text{-}to\text{-}type(n, \emptyset, <p \Rightarrow type(r)>)$)
make-true($may\text{-}hold(n, \emptyset, <p{\rightarrow}mem_k, r.mem_k>)$)

[†]We use $<p{\rightarrow}mem_k, new_n.mem_k>$ to denote all aliases involving corresponding members of the class.

Figure 3: Intraprocedural Introduction Phase : **intra-alias-type-introduction**

type-bind($call_{c2}, entry_{C::bar}, <r{\rightarrow}d1 \Rightarrow B>$) includes:

$$\{<r{\rightarrow}d1 \Rightarrow B>, <bar2 \Rightarrow B>\}$$

3.3 Initialization and Introduction Phases

The algorithm starts by lazily initializing all the *points-to-type* and *may-hold* predicates to *false*; this enables us to perform initialization of all the predicates in constant time [LR92]. We also initialize the worklist to empty. The intraprocedural aspects of the introduction phase are summarized in Figure 3. This introduces pointer-type and alias pairs generated locally at a pointer assignment ICFG node.

The function **inter-alias-type-introduction(**`call,entry`**)** has the following task: for each AP in **bind0**(`call,entry`), **make-true**($may\text{-}hold(\texttt{entry}, AP, AP)$); and for each PT in **type-bind0**(`call,entry`), **make-true**($points\text{-}to\text{-}type(\texttt{entry}, PT, PT)$).

alias-implies-alias-thru-assign

$may\text{-}hold(m, \emptyset, <*q, a>)$

`n :   p = q;`

$may\text{-}hold(n, \emptyset, <*p, a>)$

alias-implies-type-thru-assign

$may\text{-}hold(m, \emptyset, <*u, p>) \qquad points\text{-}to\text{-}type(m, \emptyset, <q \Rightarrow B>)$

`n :   p = q;`

$points\text{-}to\text{-}type(n, \emptyset, <*u \Rightarrow B>)$

type-implies-type-thru-assign

$points\text{-}to\text{-}type(m, \emptyset, <q \Rightarrow B>)$

`n :   p = q;`

$points\text{-}to\text{-}type(n, \emptyset, <p \Rightarrow B>)$

type-implies-alias-thru-assign

$points\text{-}to\text{-}type(m, \emptyset, <q \Rightarrow B>)$

`n :   p = q;`

$may\text{-}hold(n, \emptyset, <p \rightarrow mem_k, q \rightarrow mem_k>)$

Figure 4: Example for Intraprocedural Propagation

3.4 Intraprocedural Propagation

We present the salient features of intraprocedural propagation in Figure 4, referring to appropriate functions from Figure 1. Accompanying each function we list the predicate(s) being propagated from node `m`, the pointer assignment successor `n` through which they propagate, and finally the resulting predicate. We concentrate on pointer assignment nodes, calculating the semantic effects (i.e., transfer functions) of the code at these nodes on the information reaching from an ICFG predecessor. Propagation is trivial through a node which is not a pointer assignment; such a node can neither create nor destroy aliases or pointer-types.

3.5 Interprocedural Propagation

Propagation from `call` node For non-virtual function calls, the corresponding `entry` node is easily determined. However if `call` represents a virtual call site, the *points-to-type* predicates involving the receiver at `call` determine the possible functions invoked. Each class associated with the receiver may correspond to a virtual function. For each `entry` so determined, we propagate information from `call` to `entry`. Having described the parameter binding functions already in Section 3.2, we can describe **alias-type-propagation-from-call** as setting *true* at `entry` those *may-hold* and *points-to-type* predicates corresponding to each element of the appropriate **bind, alias-bind** and **type-bind** sets. For example in Figure 2, $<r \rightarrow d1 \rightarrow b1, bar2 \rightarrow b1> \in$ **alias-bind**$(call_{c2}, entry_{C::bar}, <r \rightarrow d1 \Rightarrow B>)$ results in **make-true**$(may\text{-}hold(entry_{C::bar}, <r \rightarrow d1 \rightarrow b1, bar2 \rightarrow b1>, <r \rightarrow d1 \rightarrow b1, bar2 \rightarrow b1>))$.

Propagation from `exit` node Suppose a pair APT holds at `exit` with assumption $AAPT$ at `entry`. Using the parameter binding functions described in Section 3.2 we determine the `call`(s) responsible for imposing $AAPT$ at `entry`, and propagate APT only to the corresponding `return`(s). This pivotal role played by the binding functions allows us to propagate information along a good approximation of consistent paths.

alias-implies-alias-from-exit(`exit`,$AAPT$,*alias*): (i) 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, $may\text{-}hold(\texttt{return}, \emptyset, alias)$ is made *true* for all `returns` corresponding to virtual or non-virtual `calls` invoking this function.

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

type-implies-type-from-exit(`exit`,$AAPT$,*pointer-type*): There are two cases similar to the previous function:

(i) If $AAPT = \emptyset$, **make-true**(*points-to-type*(`return`, $\emptyset$, *pointer-type*)),

(ii) For non-$\emptyset$ $AAPT$, **make-true**(*points-to-type*(`return`, $AAPT'$, *pointer-type*)).

3.6 Handling recursive structures

Recursive structures give rise to potentially infinite object names. To reduce the number and length of object names (and subsequently the number of aliases and pointer-types) to a finite number, we use the notion of *k-limiting*, similar to that introduced by Jones and Muchnick [JM79]. Intuitively, k-limiting implies that up to k explicit dereferences are maintained in an object name and further dereferences are abstracted into a special # dereference. For example, we represent $p{\rightarrow}next{\rightarrow}next{\rightarrow}next{\rightarrow}i$ by $p{\rightarrow}next{\rightarrow}next\#$, where k is 2. Note that there is inherent loss of information in this representation; the same k-limited object name also represents another object name $p{\rightarrow}next{\rightarrow}next{\rightarrow}j$ which has different dereference structure both in the length and member names. Unfortunately, this inability to distinguish between two object names leads to further imprecision in analysis. As we show in Section 4, the loss of precision is within an acceptable range in terms of the quality of analysis. Details of the algorithm in the presence of k-limited object names and the impact of k-limiting on analysis precision are beyond the purview of this paper, and appear in [Pan95].

4 Implementation Results

The results presented in this section represent our efforts to empirically demonstrate the contributions of the algorithm and assess its practicality using a prototype implementation. The prototype is written in C and runs on a Sun SPARC-20. We are using the MasterCraft C^{++} system of Tata Consultancy Services as the front end C^{++} parser for the implementation. Our aliasing and type determination algorithm reuses some code from Landi-Ryder aliasing algorithm [LR92] with suitable modifications. We present empirical results of analyzing 19 C^{++} programs obtained from various (publicly available) sources such as textbooks, demonstration programs accompanying a C^{++} compiler and undergraduate projects.

The test suite of C^{++} programs Table 1 lists some characteristics of programs we analyzed, such as the lines of code (LOC), number of functions, number of virtual functions and number of virtual calls. Also, we list the number of virtual call sites present in those functions which were found unreachable, number of aliases per ICFG node, number of pointer-types per ICFG node and finally, the program analysis time. Some programs have high analysis times because we have not optimized our prototype for time performance. In the remainder of this section, unless otherwise stated, we normalize the data with respect to virtual call sites in reachable functions. In the charts appearing in this section, we will list the programs by number (from Table 1) instead of by name.

Virtual function resolution Based on the distinct classes of objects pointed to by the receiver at a virtual call site, the algorithm determines which virtual functions in the inheritance hierarchy may be invoked from the call site. In Figure 5 we classify the reachable virtual call sites in terms of the number of virtual functions found invocable. Our results corroborate the observation by Calder and Grunwald that although object-oriented libraries support polymorphism through virtual functions, the target of most indirect function calls can be accurately predicted [CG94]. While their observation is based on execution profiles of programs, our results eliminate the dependence on profile data by using compile time analysis which accounts for *all* possible executions of the program. We also examined the data and ascertained that the calls for which unique resolution was not possible, indeed demonstrated polymorphism at run-time, and that the non-uniqueness was never due to approximations in analysis. We further classified uniquely resolved virtual calls into those resolved without employing data flow analysis techniques (where the class hierarchy contained only one implementation or the receiver was address of an object), and those which required the entire functionality of our algorithm (where unique resolution was out of two or more methods). Results in Figure 6 suggest that data flow analysis is necessary to obtain good results.

Invocable virtual functions at a call site and call graph construction Even when unique resolution is not possible, limiting the number of virtual functions invoked at a call site can help subsequent analyses, in that the side effects of the non-invocable functions can be safely eliminated in the context of that call. Given the potential disparity in various implementations of a virtual function, this can translate into improved precision and efficiency of analysis. Limiting the number of invocable virtual functions at a virtual call is relevant to tools which use branch prediction. In Figure 7, we report the average percentage of virtual functions found invocable out of those in the class hierarchy of the receiver type at a (reachable or unreachable) virtual call site. We create edges from a virtual call site only to the functions found invocable using resolution information, implicitly building a smaller and more precise call graph than the one built without such information. Therefore, the values in Figure 7 can also be interpreted as the size of our call graph *vis à vis* that of a naive approach.

Parameterization of k To study the effect of k-limiting on algorithm performance, we parameterized the analysis on the value of k. For each program we picked a minimum value of k, called min_k, such that the object names appearing in the source code would not be k-limited. We analyzed five programs – *deriv1, deriv2, chess, ocean* and *primes* – with three values of k, *viz.* $(min_k + 2)$, $(min_k + 1)$ and min_k. In most cases, we found that virtual function resolution was insensitive to the value of k. This is probably because no receiver of a virtual function call was ever k-limited, owing to the selection criterion for min_k.

We also observed that the size of alias and pointer-type solution usually decreased with lower values of k. This was the net result of two opposite effects of k-limiting [Pan95]: On the one hand, multiple object names may map to a single k-limited object name for a lower value of k, leading to reduction in the number of object names and correspondingly alias and pointer-type solution size. On the other hand, further loss of precision caused by a lower value of k leads to increased solution size due to spurious pairs. It was heartening to see that any possible size increase due to loss of precision was almost always overtaken by the reduction due to many-to-one object name mapping. Table 2 summarizes our observations. Note that for *deriv2*, the size of alias solution and precision of function resolution suffered for a lower k.

Effect of inlining base class constructors In C^{++} the derived class constructor calls the constructor of each base class to initialize the inherited base class members. While analyzing the body of the base constructor, we realized that the algorithm could not distinguish the calling contexts of distinct derived constructors. As a result, the information due to one derived constructor went to *all* derived constructors on return from the base constructor. In other words, our approximations led to propagation of information over non-realizable paths. In order to regain the calling context, we inlined the base constructor in each derived constructor. The dramatic improvements in time requirement and precision of analysis are reported in Table 3. They underline the need for context sensitive analysis for precision and also suggest a potential technique to improve efficacy of other analyses of object-oriented programs.

5 Conclusions

We have presented the first polynomial time combined algorithm 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 approximate solution in virtual function resolution using a prototype implementation. Currently, we are continuing to gather empirical data. We plan to extend our work to be able to analyze larger programs and to solve other analysis problems useful for applications like debugging and testing in a C^{++} programming environment.

name	LOC	functions	virtual functions	virtual calls	%unreachable virtual calls	aliases per node	ptr-types per node	time min:sec
1. office	213	12	4	4	0	49	10	0:04
2. greed	968	47	9	17	35	2	2	0:31
3. family	109	22	3	3	0	4	3	0:02
4. FSM	98	15	2	1	0	6	2	0:01
5. garage	143	19	3	10	0	304	34	0:56
6. vcircle	142	16	4	5	0	0	1	0:01
7. deriv2	313	34	16	66	45	29	3	1:04
8. shapes	267	32	12	31	19	54	5	2:19
9. deriv1	192	31	13	28	29	22	4	0:15
10. objects	465	59	31	39	85	6	1	0:08
11. simul	339	54	12	7	15	3	1	0:03
12. primes	46	11	4	3	0	78	11	0:26
13. ocean	444	64	10	5	0	81	8	1:58
14. NP	31	7	2	6	0	0	9	0:01
15. city	519	67	2	1	0	322	12	13:27
16. tree	217	26	8	3	33	726	34	11:14
17. employ	894	58	25	4	0	213	14	7:42
18. life	178	21	8	2	0	49	3	0:26
19. chess	392	43	12	1	0	58	16	0:54

Table 1: Some Characteristics of C^{++} Programs Analyzed

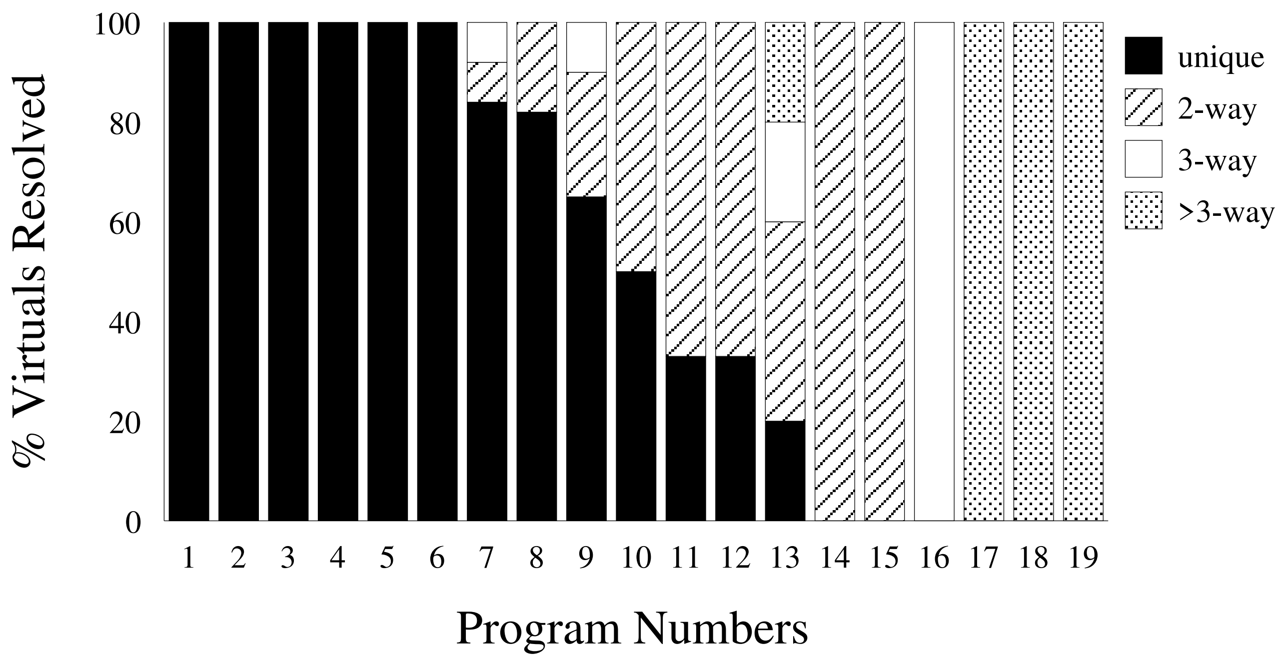

Figure 5: Percentage of Virtual Calls with 1,2,3,>3 Invocable Functions

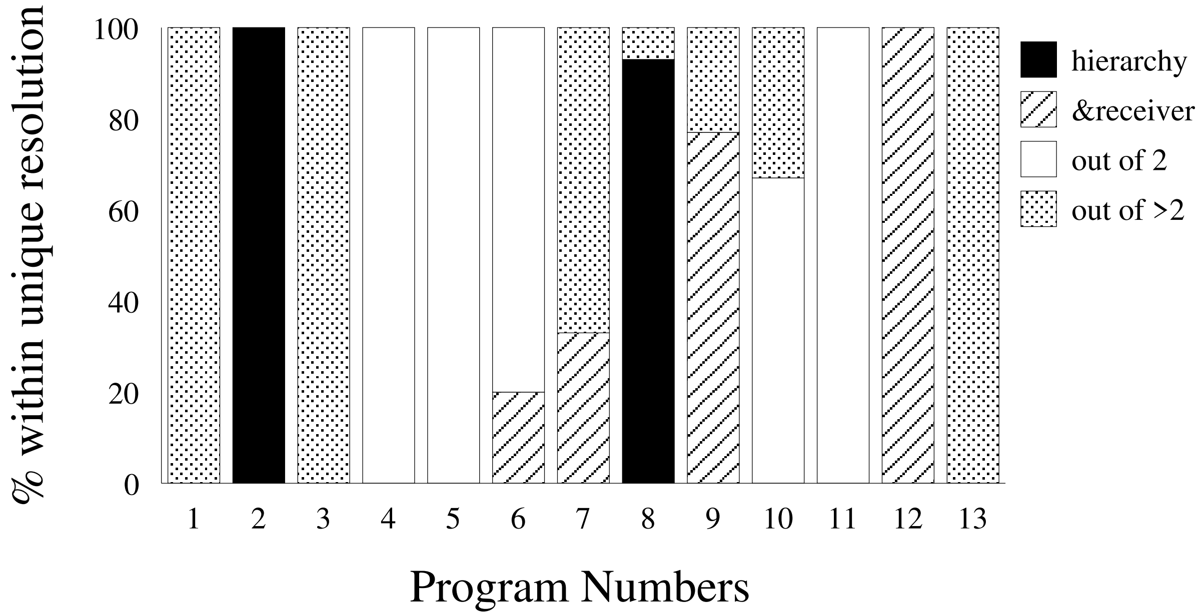

Figure 6: Further Classification of Unique Resolution from Programs 1-13 in Figure 5

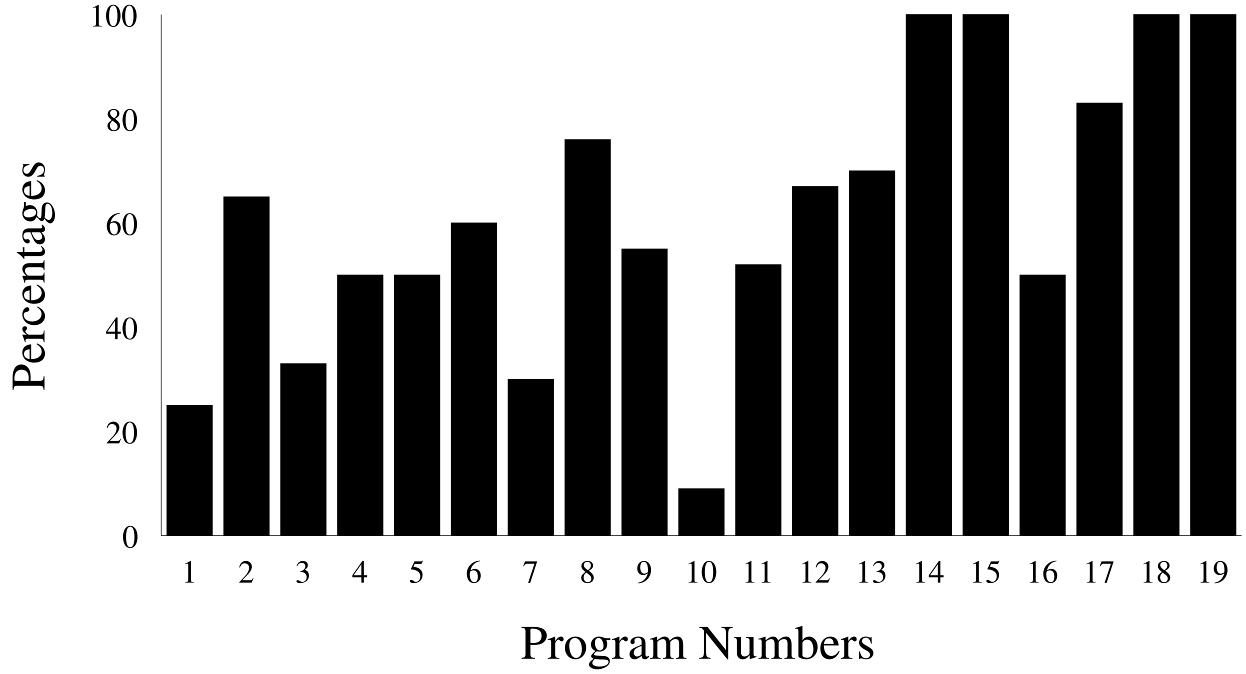

Figure 7: Average Percentage of Invocable Virtual Methods out of those in Class Hierarchy at Virtual Call

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

Table 2: Parameterization of k

name		aliases and pointer-types	*may-hold* per node	*points-to-type* per node	time min:sec
deriv1	original	41430	1823	24	25:01
	inlined	2409	42	4	0:14
deriv2	original‡	>65000	–	–	–
	inlined	7367	60	3	1:05
shapes	original	4508	105	6	0:54
	inlined	4264	88	5	0:56
primes	original	4947	349	12	0:25
	inlined	2291	192	7	0:16

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

Table 3: Inlining Base Class Constructors

References

[AH95] A. Agesen and U. Hölzle. Type feedback vs. concrete type inference: a comparison of optimization techniques for object-oriented languages. *Department of Computer Science Technical Report TRCS 95-04, University of California, Santa Barbara*, March 1995.

[APS93] O. Agesen, J. Palsberg, and M 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.

[Bar78] J. Barth. A practical interprocedural data flow analysis algorithm. In *Communications of the ACM*, 21(9):724–736, September 1978.

[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.

[CU89] C. Chambers and D. Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 146–160, June 1989.

[CU90] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: optimizing dynamically-typed object-oriented programs. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 150–164, June 1990.

[DCG95] J. Dean, C. Chambers and D. Grove. Selective specialization for object-oriented languages. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 93–102, June 1995.

[DMM95] A. Diwan, J. Eliot B. Moss and and K. S. McKinley. Analyzing statically-typed object-oriented programs for modern processors. *Technical Report, Department of Computer Science, University of Massachusetts, Amherst*, July 1995.

[EGH94] M. Emami, R. Ghiya and L. 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.

[Fer95] M. Fernandez. Simple and effective link-time optimization of Modula-3 programs. In Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation, pages 103–115, June 1995.

[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.

[HU94] U. Hölzle 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.

[Har89] L. W. Harrison III. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, 1989.

[JM79] N. Jones and S. Muchnick. Flow analysis and optimization of lisp-like structures. In *Conference Record of the Sixth Annual ACM symposium on Principles of Programming Languages*, pages 244–256, January 1979.

[JW95] S. Jagannathan and A. Wright. Efficient flow analysis for avoiding run-time checks. In *Proceedings of the Second International Static Analysis Symposium (SAS '95).* Also appears as Springer-Verlag LNCS-983.

[KAI95] S. Kumar, D. P. Agrawal and S. P. Iyer. An improved type-inference algorithm to expose parallelism in object-oriented programs. In *Proceedings of the Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Kluwer Academic Publications, Troy, New York, May 22-24, 1995.

[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.

[Lan92] W. Landi. Interprocedural aliasing in the presence of pointers. Ph.D. Thesis, Department of Computer Science, Rutgers University, January 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.

[PC94] J. Plevyak and A. Chien. Precise concrete type inference for object oriented languages. In Proceeding of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '94), pages 324–340, October 1994.

[PLR94] H. D. Pande, W. Landi and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. 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.

[PR94a] H. D. Pande and B. G. Ryder. Static type determination and aliasing for C++. *Laboratory of Computer Science Research Technical Report LCSR-TR-236, Rutgers University*, December 1994. Note: This is an expanded version of [PR94].

[PS91] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91)*, pages 146–161, October 1991.

[Pan95] H. D. Pande. Interprocedural compile time analysis of C and C++ systems. *PhD Thesis, Department of Computer Science, Rutgers University*, in preparation, 1995.

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

[RHS95] T. Reps, S. Horwitz and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Twenty Second Symposium on Principles of Programming Languages*, pages 49–61, January 1995.

[Ruf95] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, Pages 13–22, June 1995.

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

[Shi90] O. Shivers. Data-flow analysis and type recovery in Scheme. In *Topics in Advanced Language Implementation*, MIT Press, 1990.

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

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

[WL95] R. Wilson and M. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, Pages 1–12, June 1995.

[Wei80] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Seventh Symposium on Principles of Programming Languages*, pages 83–94, 1980.